AGraph: библиотека классов для работы с помеченными графами

§1. Актуальность разработки библиотек для работы с графами

К настоящему времени накоплен большой опыт решения теоретико-графовых задач на ЭВМ. Программы для решения многих задач можно найти в глобальной сети Интернет. В то же время, использование независимо разработанных программ сопряжено с большими трудностями. К их числу относятся как общие, не зависящие от предметной области, технические проблемы (различные языки программирования, несовместимость программных и аппаратных средств), так и проблемы, связанные со спецификой теоретико-графовых задач (использование различных внутренних представлений графов). В связи с этим актуальной задачей является разработка более или менее универсальных библиотек, которые, с одной стороны, предоставляли бы пользователю высокоуровневые средства для работы с графами, а с другой, избавляли его от необходимости ручного программирования рутинных операций ввода-вывода или преобразований между различными внутренними представлениями графов.

Разработка универсальной библиотеки для работы с графами является сложной задачей. Одной из проблем является большое разнообразие задач теории графов. Поскольку теоретические исследования и разработка новых алгоритмов непрерывно продолжаются, очевидно, что никакая библиотека не сможет решать все существующие задачи. Другой проблемой является обеспечение эффективности. Нередко существует несколько алгоритмов для решения одной и той же задачи, причем не всегда можно указать алгоритм, оптимальный во всех случаях: для одних графов более эффективным может быть один алгоритм, для других - другой. Разработчик универсальной библиотеки обычно не может позволить себе реализацию нескольких алгоритмов для решения одной задачи, поэтому ему приходится идти на компромиссы между эффективностью и универсальностью. При разработке библиотек для работы с графами возникают также многочисленные технические трудности. Для приемлемой с точки зрения эффективности реализации многих алгоритмов программисту необходимо иметь в своем распоряжении такие структуры данных, как динамические массивы, списки, стеки, очереди, приоритетные очереди, деревья поиска. Реализация всех необходимых структур данных в рамках одной библиотеки вряд ли возможна и оправдана, поэтому универсальная библиотека для работы с графами требует серьезной программной "инфраструктуры" в виде других библиотек.

Перечисленные проблемы могут вызвать сомнения относительно целесообразности создания универсальных библиотек для работы с графами, однако существуют весомые аргументы в пользу их создания. Во-первых, реализованные в подобной библиотеке базовые алгоритмы могут служить хорошей основой для создания более специализированных алгоритмов и программ, направленных на решение конкретных прикладных задач. Во-вторых, соображения эффективности не всегда являются определяющими - постоянный рост производительности ЭВМ все чаще выводит на первый план технологичность и скорость разработки программного обеспечения (разумеется, это не означает, что программист не должен стремиться к эффективному использованию вычислительных ресурсов). Наряду с "промышленным" программирования, универсальные библиотеки для работы с графами могут применяться в учебных целях, а также для поддержки теоретических исследований, связанных с алгоритмами и программами решения задач теории графов. В обоих случаях универсальность проблемной ориентации библиотеки более важна, чем максимальная эффективность реализованных в ней алгоритмов.

**§2. Объектно-ориентированные библиотеки для работы с графами**

**1. Преимущества ООП при создании библиотек для работы с графами**

При создании "первого поколения" библиотек для работы с графами использовались языки программирования Fortran, Algol, PL\1, затем С [1]. Для решения теоретико-графовых задач использовались и непроцедурные языки, такие, как язык функционального программирования LISP и логического программирования PROLOG, однако из-за недостаточной эффективности и технологических трудностей разработки больших программных систем на этих языках эти языки не подходят для создания универсальных библиотек.

С развитием объектно-ориентированного программирования (ООП) началась разработка объектно-ориентированных библиотек для работы с графами. Использование средств ООП при решении теоретико-графовых задач дает существенные преимущества по сравнению с традиционным структурным подходом, поскольку сам граф, его вершины и ребра являются "готовыми" объектами, данными самой природой задачи. К достоинствам ООП, которые наиболее ярко проявляются при работе с графами, можно отнести следующее:

1. программный код становится более компактным, улучшается его читаемость;
2. при реализации алгоритмов появляется возможность абстрагироваться от деталей внутреннего представления графа;
3. внутреннее представление графа можно менять в широких пределах без влияния на "высокоуровневые" составляющие библиотеки;
4. легко решается проблема "привязки" данных к вершинам и ребрам графа.

2. Обзор существующих ОО-библиотек для работы с графами

В настоящее время существует несколько объектно-ориентированных библиотек, предоставляющих средства для работы с графами. Среди них можно отметить:

* LEDA (Library of Efficient Data types and Algorithms);
* GTL (Graph Template Library, University of Passau);
* GTL (Graph Template Library, Евгений Цыпнятов, Нижний Новгород), далее - GTL (Н-Новгород).

Все эти библиотеки написаны на языке C++.

Библиотека LEDA (последняя версия - 3.8) [2] разрабатывается с 1988 г. в Институте Информатики Макса Планка (Сарабрюккен, ФРГ). Библиотека предлагает различные абстрактные типы данных (стеки, очереди, приоритетные очереди, отображения, списки, множества, разбиения, словари, интервальные множества и др.), специализированные числовые типы данных (рациональные числа, большие вещественные числа, алгебраические числа и др.), графы и вспомогательные структуры данных для работы с графами. В LEDA реализованы алгоритмы решения ряда комбинаторных, алгебраических, геометрических и теоретико-графовых задач, средства графического ввода и вывода. Институт Информатики Макса Планка бесплатно предоставляет библиотеку, включая исходные тексты, по лицензии, которая дает право использовать LEDA для академических исследований и/или обучения, но не допускает коммерческое использование.

Программный интерфейс приложений (API) для работы с графами, реализованный в LEDA, послужил образцом для создания других библиотек, в том числе GTL (University of Passau) (последняя версия - 0.3.1). В отличие от LEDA, GTL базируется на STL (C++ Standard Template Library) - мощной библиотеке классов-контейнеров и стандартных алгоритмов. Существует GTL-Java интерфейс, позволяющий Java-программам использовать графовые структуры данных и алгоритмы, реализованные в GTL. По своим функциональным возможностям и надежности GTL в настоящее время значительно уступает LEDA.

Библиотека GTL (Евгений Цыпнятов, последняя версия - 1.0R8) [3] существенно отличается от других библиотек по своей идеологии. Во-первых, библиотека поддерживает несколько внутренних представлений для графов - в виде массивов вершин и ребер, списков смежности, матрицы смежности. Существует также представление, которое объединяет все три перечисленные выше структуры хранения графов и обеспечивает их автоматическую синхронизацию. Представления реализованы в виде шаблонных классов; выбор нужного представления осуществляется при создании графа. Во-вторых, библиотека использует оригинальный способ придания необходимых "свойств" вершинам и ребрам графа (фактически, "свойства" - это атрибуты вершин и ребер) - механизм классов-"привкусов" (Flavor). Этот способ основан на использовании множественного наследования и параметризуемых (шаблонных) классов графов. Механизм "привкусов" будет рассмотрен при сравнении с аналогичными средствами библиотек LEDA и AGraph. В настоящее время GTL доступна только на платформе Win32, т.к. она существенно зависит от библиотеки MFC (Microsoft Foundation Classes).

§3. Библиотека AGraph

1. Общая характеристика

При разработке библиотеки AGraph были поставлены следующие цели:

* охват широкого круга теоретико-графовых задач;
* простота использования;
* эффективность.

Библиотека AGraph написана на языке Object Pascal [4], который используется в Delphi - среде быстрой разработки приложений (RAD) фирмы Inprise (бывшей Borland), и является, вероятно, единственной развитой библиотекой для работы с графами на Object Pascal. В то же время, используемый язык программирования - не главное отличие AGraph от других библиотек. При необходимости библиотека AGraph может быть переписана с использованием таких объектно-ориентированных языков программирования, как C++, Eiffel или Java. Перенос облегчается тем обстоятельством, что AGraph не использует стандартную библиотеки классов Delphi VCL (Visual Component Library), за исключением классов исключительных ситуаций (Exception).

В пользу выбора языка Object Pascal как средства создания библиотеки для работы с графами можно привести следующие соображения. К настоящему времени разработано немало объектно-ориентированных языков программирования (ООЯП): Smalltalk, C++, Java, Object Pascal, Eiffel, Oberon-2, Modula-3 и другие. Если исходить из достоинств и недостатков самих языков программирования, не принимая во внимание распространенность языка и качество его конкретных реализаций, то одним из лучших "кандидатов", на мой взгляд, является Eiffel. Однако, если учитывать конкретную платформу, которая имеется в распоряжении (персональный компьютер на процессоре семейства Intel 386, работающий под управлением операционных систем Windows или Linux), а также практически доступные системы программирования коммерческого качества, то выбор значительно сужается: остаются языки C++, Java и Object Pascal. Языки Smalltalk и Java не подходят по соображениям эффективности. Наиболее распространенный в настоящее ООЯП, C++, поддерживается на большинстве платформ и является мощным языком программирования. Важное значение имеет существование стандарта языка C++ (к сожалению, многие компиляторы C++ не вполне соответствуют этому стандарту). К недостаткам С++ можно отнести его значительно большую, по сравнению с Object Pascal, сложность. Учитывая цели, которые ставились при разработке библиотеки AGraph, в первую очередь - соображения простоты использования, выбор был сделан в пользу Object Pascal.

Язык Object Pascal в той его версии, которая реализована в Delphi, также является развитым объектно-ориентированным языком программирования. По сравнению с ранними версиями языка (Turbo Pascal и Borland Pascal), в Object Pascal некоторые изменения претерпела объектная модель, был реализован механизм свойств объектов (object property), добавлены средства обработки исключительных ситуаций (конструкции try...except и try...finally), появилась возможность передавать в процедуры и функции переменное количество параметров (open array параметры). В Delphi 4.0 появились динамические массивы, перегрузка (overloading) процедур и функций, а также возможность указывать для параметров процедур и функций значения, принимаемые по умолчанию. Среди важных языковых средств C++, которые не реализованы в Object Pascal, следует отметить множественное наследование и механизм шаблонов (templates). Последний недостаток удалось частично преодолеть с помощью "псевдошаблонов".

2. Библиотека Vectors

Создание серьезных программных систем в настоящее время практически невозможно без использования вспомогательных программных компонент, реализующих базовые структуры данных и алгоритмы. Примером такой компоненты для C++ является стандартная библиотека шаблонов (С++ STL). Рассмотренные ранее С++-библиотеки для работы с графами демонстрируют различные подходы относительно создания или использования подобных базовых средств: в LEDA все необходимые структуры данных были реализованы "с нуля", библиотека GTL (University of Passau) базируется на C++ STL, библиотека GTL (Н-Новгород) основана на MFC 4.x.

При разработке библиотеки AGraph также возникла необходимость в базовых программных средствах. Поскольку готовых средств такого рода для Object Pascal не существовало (библиотека визуальных компонент Delphi VCL ориентирована на решение других задач), пришлось создавать их самостоятельно. Реализованные в ходе этой работы базовые структуры данных и алгоритмы вошли в состав библиотеки Vectors. В библиотеке реализованы векторы (динамические массивы) на базе основных типов Object Pascal, в том числе на базе всех целых и вещественных типов, логических переменных и строк. Векторы поддерживают большое количество операций; некоторые из которых являются общими для всех векторов, другие зависят от типа элементов данного вектора. В состав библиотеки входит также ряд производных и вспомогательных классов: разреженные векторы, матрицы, сбалансированные деревья поиска, приоритетные очереди, словари, потоки в памяти, файловые потоки и др.

При написании библиотеки Vectors учитывались соображения эффективности, надежности и переносимости. Многие векторные операции реализованы в нескольких вариантах: на Object Pascal и на встроенном ассемблере Object Pascal. Выбор между вариантами на Object Pascal и встроенном ассемблере осуществляется с помощью директив условной компиляции. Если программа компилируется в режиме, разрешающем использование ассемблерных вариантов, то при запуске программы средства времени исполнения автоматически определяют расширенные возможности процессора (в настоящее время проверяется поддержка MMX-инструкций) и выбирают наиболее эффективный вариант реализации той или иной операции с учетом возможностей процессора.

Для более эффективного поиска ошибок в прикладных программах библиотека поддерживает отладочный режим (также включаемый соответствующей директивой компиляции), в котором методы классов библиотеки осуществляют максимально полную проверку выполнения предусловий и корректности передаваемых им параметров. Кроме того, в отладочном режиме осуществляется контроль над операциями создания и уничтожения объектов, относящихся к классам библиотеки. Если при завершении программы какие-либо из этих объектов не уничтожаются, то пользователю выдается запрос на запись списка неуничтоженных объектов в файл.

Серьезным препятствием при написании библиотеки Vectors стало отсутствие в языке Object Pascal средств, аналогичных шаблонам C++. Очевидно, что независимая реализация векторов, отличающихся лишь типом элементов, привела бы к дублированию программного кода, многочисленным ошибкам и, в конечном счете, грозила бы потерей управляемости проектом. Решением данной проблемы могло бы стать использование внешнего макропроцессора, однако это значительно усложнило бы как разработку, так и использование библиотеки. Вместо этого в библиотеке был применен механизм "псевдошаблонов", основанный исключительно на средствах Object Pascal: директиве INCLUDE и переопределении типов.

3. Внутреннее представление графов

Существуют различные способы внутреннего представления графов в оперативной памяти ЭВМ, в том числе в виде списков (массивов) вершин и ребер, списков (массивов) смежности, матриц смежности, а также в виде комбинаций этих структур хранения. Выбор внутреннего представления оказывает решающее влияние на эффективность выполнения различных операций над графами и во многом определяет "технологию" использования той или иной библиотеки в прикладных программах.

Ниже перечисленные структуры хранения графов будут рассмотрены более подробно, но перед этим необходимо сделать следующее замечание. В теории графов вершины и ребра графов, как правило, лишены индивидуальности: при таком подходе граф можно задать, например, булевской матрицей смежности, где логическая единица на пересечении i-ой строки и j-го столбца означает существование ребра (или дуги) между i-ой и j-ой вершинами графа. В то же время, во всех рассматриваемых библиотеках вершины и ребра графа считаются уникальными объектами (здесь термин "объект" употребляется в контексте объектно-ориентированного программирования), которые различаются, по крайней мере, тем, что каждый из них занимает отдельный участок в оперативной памяти ЭВМ. Объект-вершина может содержать или не содержать какие-либо данные; объект-ребро содержит, как минимум, указатели на инцидентные ему вершины. Если подходить с технологической точки зрения, то наличие уникальных объектов-вершин и объектов-ребер повышает удобство реализации и эффективность многих алгоритмов (хотя и неэкономично в смысле расхода оперативной памяти). Существует и более фундаментальная причина: при решении прикладных задач вершины графа, а иногда и его ребра, соответствуют реальным объектам предметной области. Таким образом, структуры хранения графов в объектно-ориентированной библиотеке для работы с графами обеспечивают хранение не только "математического" графа, но и объектов, представляющих вершины и ребра этого графа. Еще одно замечание необходимо сделать относительно использования списков и/или массивов: эти структуры данных будут считаться взаимозаменяемыми, пока изложение не коснется конкретных библиотек.

Списки вершин и ребер

Граф представляется в виде двух списков, один из которых содержит указатели на его вершины, второй - на ребра (как всегда, каждое ребро хранит в себе указатели на инцидентные ему вершины). Данная структура хранения обеспечивает эффективное добавление в граф вершин и ребер, а также итерацию по вершинам и ребрам. В то же время, это представления неудобно, когда необходимо определить ребра, инцидентные заданной вершине графа.

Списки смежности

Граф представляется списком входящих в него вершин. В свою очередь, каждая вершина содержит список инцидентных ей ребер (или списки входящих и исходящих дуг в случае орграфов). Данное представление обеспечивает эффективное добавление в граф вершин и ребер, итерацию по вершинам графа и доступ к ребрам, инцидентным заданной вершине, но не поддерживает итерацию по ребрам графа.

Матрицы смежности

Граф задается квадратной матрицей размерности NxN, где N - количество вершин в графе; на пересечении i-го столбца и j-ой строки матрицы находится либо указатель на ребро, соединяющее вершины i и j, если эти вершины инцидентны, либо nil, если они не инцидентны. Вершины могут либо храниться в отдельном списке, либо только в составе инцидентных им ребер (в случае связных графов). Это представление удобно для реализации некоторых алгоритмов, но не обеспечивает эффективное добавление и удаление вершин. Кроме того, оно является самым неэкономичным по расходу памяти (за исключением графов, в которых количество ребер значительно превышает количество вершин).

Из приведенного анализа видно, что каждая из трех рассмотренных структур хранения графов обладает своими достоинствами и недостатками. Внутреннее представление графов в универсальной библиотеке должно обеспечивать эффективную реализацию большого числа разнообразных алгоритмов, поэтому такие библиотеки используют комбинированные представления, либо, как это сделано в GTL (Н-Новгород), позволяют явно указать внутреннее представление при создании объекта-графа.

Распространенным вариантом комбинированного внутреннего представления является объединение представлений в виде списков вершин/ребер и списков смежности. Такая структура хранения обеспечивает эффективное добавление и удаление вершин и ребер, итерацию по вершинам и ребрам и, в то же время, позволяет легко определить ребра, инцидентные заданной вершине графа. Подобное представление используется в библиотеках LEDA и GTL (University of Passau).

Библиотека AGraph также использует комбинированное представление, но вместо списков применяются динамические массивы указателей, реализованные в библиотеке Vectors (класс TPointerVector, он же TClassList). Сравнение списков и динамических массивов, реализованных в Vectors, с точки зрения эффктивности выполнения различных операций приведено в следующей таблице (n обозначает количество вершин в графе, m - количество ребер):

|  |  |  |
| --- | --- | --- |
| **Операция** | **Эффективность** | |
|  | **Списки** | **Массивы** |
| Добавление вершины | ребра | O(1) | O(n) | O(m) в худшем случае, O(1) в среднем |
| Удаление вершины | ребра | O(1) | O(n) | O(m) |
| Доступ к вершине | ребру по индексу | O(n) | O(m) | O(1) |

Списки позволяют эффективно добавлять и удалять вершины (ребра) графа, но не обеспечивают прямой доступ к ним по индексу (т.е. порядковому номеру) вершины (ребра) в соответствующем списке. Возможность получить ссылку на элемент графа по его индексу является весьма полезной при реализации многих алгоритмов, поэтому для решения данной проблемы приходится использовать дополнительные структуры данных.

Достоинством динамических массивов является быстрый доступ к элементам по индексу. Наиболее "дорогой" операцией при использовании динамических массивов является добавление элемента, поскольку в худшем случае для этого требуется выделить новый блок памяти увеличенного размера, скопировать содержимое старого блока памяти в новый и освободить старый блок, причем, по крайней мере, этап копирования имеет сложность O(n). В то же время, в среднем операция добавления вершин (ребер) в AGraph выполняется более эффективно. Это достигается за счет того, что при увеличении размера динамического массива (вектора) в библиотеке Vectors память выделяется сразу для нескольких элементов, поэтому в большинстве случаев операция добавления элемента не требует фактического изменения размера вектора.

Особенностью библиотеки AGraph является то, что каждая вершина (ребро) графа хранит собственный индекс в массиве вершин (ребер) графа. Наличие такой "обратной" ссылки во многих случаях значительно упрощает работу с графом. Поддержание этой ссылки не ухудшает асимптотическую сложность операций добавления и удаления вершин (ребер) графа.

4. Базовые средства

К базовым средствам библиотеки для работы с графами относятся средства, обеспечивающие выполнение наиболее общих операций над графом и его элементами, в том числе создание и уничтожение объектов-графов, добавление в граф вершин и ребер, удаление их из графа, итерацию по вершинам и ребрам и т.д. Базовые средства библиотеки AGraph в основном соответствуют аналогичным средствам других библиотек (см. пример 1). При создании программного интерфейса приложений (API) библиотеки AGraph первоочередное внимание уделялось обеспечению максимальных функциональных возможностей библиотеки при сохранении простоты ее использования. Имена классов библиотеки, их полей, методов и свойств (property) соответствуют распространенной англоязычной терминологии теории графов и общепринятым соглашениям языка Object Pascal.

// создание графа

G:=TGraph.Create;

// добавление вершин и ребер

V:=G.AddVertex;

G.AddVertices(5);

E:=G.AddEdge(G[0], G[1]); // ребро (v0, v1)

G.AddEdgeI(0, 3); // ребро (v0, v3)

G.AddEdges([1, 2, 2, 4]); // ребра (v1, v2) и (v2, v4)

// итерация по вершинам графа

for I:=0 to G.VertexCount - 1 do begin

V:=G.Vertices[I];

// итерация по ребрам, инцидентным заданной вершине графа

for J:=0 to V.Degree - 1 do

With V.IncidentEdge[J] do {...} ;

end;

// итерация по ребрам графа

for I:=0 to G.EdgeCount - 1 do

With G.Edges[I] do {...} ;

// удаление ребра (v0, v1)

E.Free;

// удаление ребра (v1, v2)

G.GetEdgeI(1, 2).Free;

// удаление вершины (с инцидентными ребрами)

G.Vertices[2].Free;

// уничтожение графа

G.Free;

Пример 1. Базовые операции над графами в библиотеке AGraph.

Если сравнивать библиотеки AGraph и LEDA, то можно отметить два существенных отличия. Первое из них связано с использованием динамических массивов для внутреннего представления графов в библиотеке AGraph. Массивы позволяют применять простой for-цикл для итерации по вершинам и ребрам графа. В библиотеке LEDA, использующей списки для хранения вершин и ребер, для той же цели необходимо использовать специальные макросы, а в библиотеке GTL (Passau), основанной на STL, - итераторы STL [библиотека LEDA также поддерживает "STL-style" итераторы (пока в качестве экспериментальной возможности)]. Второе отличие заключается в том, что в AGraph для удаления вершин и ребер из графа используется стандартный способ уничтожения объектов Object Pascal - вызов метода Free, в то время как в библиотеке LEDA для удаления вершин и ребер из графа приходится использовать специальные методы классов-графов.

Наиболее важным различием между библиотеками AGraph и GTL (Н-Новгород) является то, что в библиотеке GTL вершины и ребра существуют отдельно от графов и могут принадлежать сразу нескольким графам либо ни одному из графов. Для удаления неиспользуемых вершин (ребер) из памяти в GTL используется техника счетчиков ссылок (reference count): когда объект (вершина или ребро) присоединяется к графу или другой структуре (метод AddRef), счетчик увеличивается, когда удаляется из структуры (метод Release) - уменьшается. При удалении ссылки на объект из последней структуры, он должен удалить себя из памяти. Такое решение позволяет сэкономить память в тех случаях, когда графы конструируются из уже существующих объектов. Одновременно снимается проблема отождествления объектов "родственных" графов: например, в GTL порожденный подграф графа содержит *те же* вершины, что и сам граф (а не копии этих вершин!). В то же время, такая интерпретация вершин и ребер может затруднить использование библиотеки. Во-первых, проблемы могут возникнуть, когда с вершинами и ребрами графа связаны какие-либо атрибуты (см. ниже) - изменение атрибута вершины (ребра) одного графа может вызвать неожиданное для пользователя изменение атрибута вершины (ребра) *другого* графа. Во-вторых, возможность существования "автономных" вершин и ребер, на мой взгляд, противоречит интуитивному пониманию графа.

5. Использование атрибутов

Во многих случаях необходимо связать с вершинами и ребрами графа дополнительные данные - атрибуты (метки) вершин и ребер графа. Так, во взвешенном графе каждое ребро имеет целый или вещественный атрибут - вес ребра; в молекулярном графе с вершинами и ребрами может быть связан целый ряд атрибутов (для вершин - номер атома, представляемого данной вершиной графа, в периодической таблице элементов Д.И.Менделеева, заряд атома и др., для ребер - тип валентной связи, которой соответствует данное ребро).

Существует несколько способов привязки атрибутов к вершинам и ребрам графа. Наиболее простым из них является использование для хранения каждого атрибута вспомогательной структуры данных, между элементами которой, с одной стороны, и вершинами (ребрами) графа, с другой стороны, существует взаимно-однозначное соответствие (см. пример 2). В данном примере используется особенность библиотеки AGraph, обеспечивающая эффективное получение для объекта-вершины (объекта-ребра) его индекса в массиве вершин (ребер).

// создание графа

G:=TGraph.Create;

V:=G.AddVertex;

E:=G.AddEdge(V, G.AddVertex);

// создание динамического массива для хранения целого

атрибута вершин графа (первый параметр - количество

элементов, второй - значение по умолчанию)

AttrVector1:=TIntegerVector.Create(G.VertexCount, 0);

// создание динамического массива для хранения строкового

атрибута ребер графа

AttrVector2:=TStrLst.Create;

AttrVector2.Count:=G.EdgeCount;

// присваивание значений атрибутам вершины V и ребра E графа

AttrVector1[V.Index]:=1;

AttrVector2[E.Index]:='AGraph';

Пример 2. Использование динамического массива для хранения атрибутов вершин и ребер графа в библиотеке AGraph.

В библиотеке LEDA для реализации аналогичного способа привязки атрибутов к вершинам и ребрам графа необходимо использовать специальные структуры данных - классы node\_array и edge\_array (либо отличные от них по реализации, но эквивалентные в данном контексте классы node\_map и edge\_map). Это связано с тем, что в LEDA объекты-вершины и объекты-ребра хранятся в списках, а не в динамических массивах.

// создание графа

graph G;

node v = G.new\_node();

edge e = G.new\_edge(v, G.new\_node());

// создание структуры node\_arrray для хранения атрибута типа int

для вершин графа G

node\_array attr1(G);

// создание структуры edge\_arrray для хранения атрибута типа

string для ребер графа G

edge\_array attr2(G);

// присваивание значений атрибутам вершины v и ребра e графа G

attr1[v]:=5;

attr2[e]:="Saarbruecken";

Пример 3. Использование классов node\_array и edge\_array для хранения атрибутов вершин и ребер графа в библиотеке LEDA.

Описанный способ хранения атрибутов подходит только для статических графов, т.к. при модификации графа соответствие между вершинами (ребрами) графа и элементами вспомогательной структуры данных теряется. Кроме того, определенные таким образом атрибуты не могут автоматически сохраняться при записи графа в поток или копироваться при копировании графов. Наиболее естественным способом "надежной" привязки атрибутов к вершинам (ребрам) графа является хранение атрибутов (или ссылок на атрибуты) непосредственно в объектах-вершинах (объектах-ребрах) графа. Библиотеки LEDA и GTL (University of Passay) предлагают для этого параметризуемый класс графов, библиотека GTL (Н-Новгород) - использование классов-"привкусов" (Flavor) и множественного наследования. Оба этих метода хорошо работают, когда набор атрибутов вершин (ребер) графа известен во время компиляции программы. В библиотеке AGraph реализованы еще более гибкие средства, позволяющие создавать и уничтожать атрибуты *динамически*, в процессе исполнения.

Параметризуемый класс графов в LEDA позволяет создавать графы, с каждой вершиной и каждым ребром которого связан атрибут заданного типа (см. пример 4). Доступ к таким атрибутам является в LEDA более эффективным, чем доступ к атрибутам, определенным с помощью node\_array и edge\_array.

// создание графа

GRAPH H;

node v = H.new\_node();

edge e = H.new\_edge(v, H.new\_node());

// присваивание значений атрибутам вершины v и ребра e графа G

H[v]:=5;

H[e]:="Saarbruecken";

Пример 4. Использование параметризуемого класса графов LEDA.

В библиотеке GTL (Н-Новгород) для создания вершин и ребер с заданными свойствами используется механизм классов-"привкусов" (Flavor). Данный механизм может использоваться для привязки атрибутов к вершинам и ребрам графа, хотя его возможности этим не ограничиваются. Flavor - это абстрактный (чисто виртуальный в терминологии С++) класс, который применяется в качестве "добавки" при порождении новых классов с использованием множественного наследования. Flavor требует, чтобы объект обладал некоторыми свойствами, но не привносит их в объект сам. Например, класс CWeightedEdge ("взвешенное ребро") порождается от классов CEdge ("ребро") и специального класса CWeightFlavor. В классе CWeightFlavor определены два абстрактных виртуальных метода - SetWeight и GetWeight, которые должны быть перекрыты в классе CWeightedEdge. GTL предоставляет ряд "привкусов" для построения распространенных типов объектов (при необходимости пользователи могут расширить набор Flavor). Порожденные с помощью Flavor классы, в свою очередь, используются в качестве параметров шаблонных классов графов для создания классов графов, вершины и ребра которых обладают заданными свойствами (см. пример 5).

// определение класса CWEdge (взвешенное ребро)

template class CWEdge:public CEdge,CWeightFlavor

{

protected:

double m\_dWeight;

public

...

virtual void SetWeight(double dWeight) {m\_dWeight=dWeight;}

virtual double GetWeight() const {return m\_dWeight;}

...

};

// определение синонимов для вершин и ребер

#define V CVertex

#define E CWEdge

...

// создание ориентированного графа с использованием представления

в виде списков смежности

CGraphAdjList xGraphAL(TRUE);

// создание графа с использованием макросов

VPOS xPos1,xPos2,xPos3;

AL\_ADDVERTEX(&xGraphAL,new V,xPos1);

AL\_ADDVERTEX(&xGraphAL,new V,xPos2);

AL\_ADDVERTEX(&xGraphAL,new V,xPos3);

AL\_ADDEDGE(&xGraphAL,xPos1,xPos2,new E(1.0));

AL\_ADDEDGE(&xGraphAL,xPos1,xPos3,new E(3.0));

// доступ к весу ребра (методы GetWeight и SetWeight определены в

классе CWeightFlavor)

E\* e = xGraphAL.GetIncidEdgeAt(xPos1, 0);

double w = e->GetWeight;

e->SetWeight(1.0);

Пример 5. Использование классов-"привкусов" в GTL (Н-Новгород).

Смысл в использовании Flavor проявляется, когда объект должен обладать несколькими свойствами: например, требуется "взвешенное ребро с пропускной способностью". Если использовать обычное наследование, то можно построить два *различных* класса, которые фактически представляют *один и тот же* вид ребра. Множественное наследование от классов "взвешенное ребро" и "ребро с пропускной способностью" также не помогает, т.к. при этом класс "ребро" будет наследоваться дважды. Проблема легко решается с помощью Flavors: достаточно определить Flavor "пропускная способность" и породить требуемый класс от класса TEdge и двух Flavor.

Методы привязки атрибутов к элементам графа с помощью параметризуемых классов графов, реализованные в библиотеке LEDA, или с помощью классов-"привкусов", реализованные в GTL (Н-Новгород), используют средства языка C++, которые отсутствуют в Object Pascal - шаблоны и множественное наследование. Данное обстоятельство привело к реализации в библиотеке AGraph собственного механизма поддержки атрибутов. С одной стороны, это решение является единственно возможным; с другой стороны, данный механизм является более высокоуровневым и обладает большей гибкостью, чем средства других библиотек.

Атрибуты в AGraph фактически являются переменными, определенными на элементах графа. Каждый атрибут имеет уникальное имя и тип, относящийся к одному из нескольких предопределенных типов. Типы атрибутов соответствуют основным типам языка программирования Object Pascal (Integer, Boolean, Char, Double, String и др.). Библиотека позволяет *динамически создавать и уничтожать* атрибуты вершин и ребер графа, причем можно создавать как общие для всех вершин (ребер) графа атрибуты, так и локальные атрибуты, определенные только для отдельных вершин (ребер) графа (см. пример 6). Доступ к атрибутам осуществляется с помощью реализованного в Object Pascal механизма свойств (property). Для каждого из поддерживаемых типов атрибутов определены свои методы доступа (AsBool, AsChar, AsInt8, AsInt16, AsInt32, AsFloat, AsString и т.д.), благодаря чему на атрибуты распространяется контроль типов. Поскольку граф "владеет" всеми своими атрибутами, их сохранение, восстановление и копирование при выполнении соответствующих операций над графом осуществляется автоматически, полностью "прозрачно" для программиста - пользователя библиотеки.

// создание графа

G:=TGraph.Create;

// создание общего атрибута вершин графа типа String с именем 'Name'

G.CreateVertexAttr('Name', AttrString);

// присваивание значений атрибуту 'Name' вершин 0 и 1 графа

G[0].AsString['Name']:='Moscow';

G[1].AsString['Name']:='Minsk';

// уничтожение общего атрибута вершин графа с именем 'Name'

G.DropVertexAttr('Name');

// создание локального атрибута типа Integer с именем 'Color'

для вершины 0 графа и присваивание ему значения

G[0].Local.Map.CreateAttr('Color', AttrInteger);

G[0].Local.AsInteger['Color']:=1;

Пример 6. Работа с атрибутами в библиотеке AGraph.

Для поддержки атрибутов в библиотеке используется собственный механизм распределения памяти, который обеспечивает высокую эффективность операций создания и уничтожения атрибутов и малый расход памяти для хранения атрибутов. Единственным недостатком данного подхода является относительно медленный доступ к атрибутам: основным способом идентификации атрибута является его имя, поэтому при каждом обращении к атрибуту по имени осуществляется поиск в таблице имен атрибутов. Библиотека AGraph предоставляет низкоуровневые средства, позволяющие значительно понизить "накладные расходы" на доступ к атрибутам (ценой некоторого усложнения программирования и потенциального снижения надежности). Так, можно один раз вычислить смещение некоторого атрибута в блоке памяти, отведенном для хранения атрибутов, для того, чтобы впоследствии обращаться к данному атрибуту по смещению, а не по имени. Благодаря этому исключается относительно медленный этап поиска в таблице имен атрибутов, но снижается надежность. Существует и другой способ повышения производительности, наиболее эффективный при интенсивном использовании атрибутов: перед началом работы некоторой процедуры следует скопировать атрибуты во временную структуру данных, которая поддерживает прямой доступ (например, динамический массив), и в дальнейшем работать с этой структурой, т.е. использовать на "локальном уровне" способ привязки данных к графу, который уже был рассмотрен. Разумеется, в этом случае необходимо помнить о синхронизации графа и временной структуры данных.

Атрибуты в библиотеке AGraph предназначены не только для привязки пользовательских данных, но и активно используются внутри самой библиотеки. Например, для ребер графа (класс TEdge) определен метод RingEdge, который проверяет, является ли ребро кольцевым (т.е. при удалении данного ребра количество связных компонент графа не увеличивается). Поскольку эта проверка является относительно дорогой операцией (время выполнения может достигать O(n+m)), нежелательно осуществлять ее при каждом обращении к методу RingEdge. В библиотеке используется следующий прием: при первом обращении к методу RingEdge библиотека выполняет соответствующий алгоритм, создает глобальный атрибут ребер графа и запоминает в нем результат работы алгоритма. До тех пор, пока граф не подвергнется изменениям, которые могут повлечь нарушение правильности запомненных значений, при последующих обращениях к методу RingEdge возвращается запомненное значение. Если граф подвергнется таким изменениям, то атрибут будет автоматически уничтожен. То же самое можно было бы сделать, добавив в класс TEdge дополнительное поле для запоминания результатов выполнения метода RingEdge, однако в таком случае при отсутствии обращений к методу RingEdge память, необходимая для хранения данного поля, расходовалась бы напрасно.

6. Поддержка различных видов графов

Одна из проблем, которые возникают при разработке универсальной библиотеки для работы с графами - как реализовать поддержку различных видов графов: ориентированных и неориентированных графов, взвешенных графов, деревьев и т.д. Вид графа, во-первых, задает набор атрибутов, которые определены для вершин и ребер данного графа (например, вес ребра для взвешенного графа или указатель на родительскую вершину для дерева), во-вторых, определяет, какие методы можно применять к этому графу, и, в-третьих, может влиять на их выполнение (например, метод поиска кратчайшего пути между заданными вершинами графа выполняется по-разному для ориентированных и неориентированных графов).

Библиотеки LEDA явно поддерживает два вида графов - ориентированные и неориентированные графы: в библиотеке определены параметризуемые классы (GRAPH и UGRAPH) для каждого из этих видов. Какие-либо средства для поддержки других видов графов не предусмотрены. Если процедура или функция являются специфичной для графа определенного вида (например, функция нахождения максимального потока в транспортной сети), то все необходимые параметры (в последнем примере - пропускные способности дуг сети) непосредственно передаются в эту процедуру или функцию (например, с помощью динамических массивов).

Библиотека GTL (Н-Новгород) также непосредственно поддерживает ориентированные и неориентированные графы - конструкторы классов-графов по умолчанию строят неориентированный граф, однако существуют варианты конструкторов с явным заданием "ориентированности" графа. В то же время, библиотека поддерживает и другие виды графов с помощью классов-"привкусов". Алгоритмы, специфичные для конкретных видов графов, реализованы как обычные функции-члены (методы) параметризуемых классов графов, но в этих функциях неявно предполагается, что классы-вершины и классы-ребра, используемые при конкретизации данного параметризуемого класса графов, предоставляют необходимые интерфейсные функции. Это, в свою очередь достигается наследованием классов-вершин и классов-ребер от соответствующих классов-"привкусов".

Библиотека AGraph предлагает высокоуровневый (хотя и не вполне универсальный) подход к решению проблемы поддержки различных видов графов, использующий возможности библиотеки по динамическому созданию и уничтожению атрибутов. В библиотеке определен набор "свойств" графа, которые соответствуют конкретным видам графов. Текущая версия библиотеки поддерживает ориентированные графы, деревья, транспортные сети, взвешенные графы, геометрические графы. Не все "свойства" являются независимыми: так, транспортная сеть всегда является ориентированным графом.

Поддержка всех "свойств" реализована в одном и том же классе TGraph, который имеет свойство (в смысле property языка Object Pascal) Features типа "множество". В процессе исполнения графу можно присвоить любую комбинацию предопределенных "свойств" графов (см. пример 7). При этом библиотека автоматически создает необходимые атрибуты и разрешает использование специфичных для данного "свойства" методов (в противном случае попытка их применения приводит к возбуждению исключительной ситуации). Благодаря тому, что библиотеке известно, к каким видам относится данный граф, операции записи графа в поток и чтения из потока, а также копирования графов отрабатываются корректно (с сохранением всей "видовой" информации), причем это не требует дополнительной работы со стороны программиста - пользователя библиотеки. Поддерживаемые нынешней версией библиотеки AGraph виды не исчерпывают всех возможных разновидностей графов, которые могут понадобиться при решении прикладных задач, однако даже этот набор способен значительно облегчить работу прикладного программиста.

// создание взвешенного ориентированного дерева

G:=TGraph.Create;

G.Features:=[Directed, Tree, Weighted];

// добавление корня и двух листьев

V:=G.AddVertex;

// свойства (property) и методы, специфичные для деревьев

G.Root:=V;

V.AddChild;

U:=V.AddChild;

P:=U.Parent; // P = V

// свойства (property), специфичные для взвешенных графов

G.Edges[0].Weight:=5.1;

G.Edges[1].Weight:=2.2;

// метод FindMinWeightPath интерпретирует граф как ориентированный

// или неориентированный в зависимости от Features

T:=G.FindMinWeightPath(V[0], V[1], nil); // T = 2.2

Пример 7. Использование "свойств" (Features) графа.

Ниже все поддерживаемые библиотекой AGraph виды графов будут рассмотрены более подробно.

Ориентированные графы

Граф интерпретируется как ориентированный, если во множество Features графа входит флаг Directed (Directed in Features = True). Поддержка орграфов не требует хранения каких-либо дополнительных данных: один из концов ребра TEdge.V1 считается началом дуги, а другой конец TEdge.V2 - концом дуги. Многие методы класса TGraph учитывают свойство ориентированности графа; в то же время, доступны методы, которые рассматривают граф как ориентированный или неориентированный независимо от значения Features.

Деревья

Граф является деревом, если в его множество Features входит флаг Tree (Tree in Features = True). Указание на то, что граф является деревом (Directed in Features = True), позволяет упростить работу с древовидными структурами. Одна из вершин дерева помечается как корень. Для того, чтобы сделать вершину корнем, надо присвоить свойству IsRoot вершины значение True или, что то же самое, присвоить свойству Root графа указатель на эту вершину. Каждая вершина дерева, кроме корня, содержит ссылку на родительскую вершину (Parent). Для построения дерева следует использовать метод TVertex.AddChild.

Транспортные сети

Транспортная сеть (Network in Features = True) - это ориентированный граф с двумя выделенными вершинами: истоком (TGraph.NetworkSource) и стоком (TGraph.NetworkSink). Исток обладает тем свойством, что в него не входит ни одна дуга; из стока, напротив, не исходит ни одна дуга. Каждой дуге сети приписано неотрицательное вещественное число - максимальный поток, который может быть пропущен через эту дугу. Одной из наиболее известных задач на сетях является задача нахождения максимального потока в сети. В библиотеке реализовано решение этой задачи; для этого необходимо построить граф - транспортную сеть, указать с помощью свойств графа NetworkSource и NetworkSink исток и сток сети (то же самое можно сделать, присвоив значение True свойствам IsNetworkSource и IsNetworkSink соответствующих вершин сети), задать максимальные потоки на дугах сети с помощью свойства TEdge.MaxFlow и вызвать метод TGraph.FindMaxFlowThroughNetwork. Если построенная сеть корректна (IsNetworkCorrect = True), то этот метод возвращает значение максимального потока в сети и устанавливает свойства TEdge.Flow в значения потоков на дугах, при которых достигается максимальный поток.

Взвешенные графы

Взвешенный граф (Weighted in Features = True) - неориентированный или ориентированный граф, ребрам (дугам) которого поставлено в соответствие вещественное число, называемое весом. Вес ребра доступен с помощью свойства TEdge.Weight. Классической задачей на взвешенном графе является задача нахождения кратчайшего пути (пути с минимальной суммой весов входящих в этот путь ребер или дуг) между заданными вершинами, либо между всеми парами вершин. Задача нахождения кратчайшего пути между заданными вершинами во взвешенном графе с неотрицательными весами решается в библиотеке методами TGraph.FindMinWeightPathCond / TGraph.FindMinWeightPath, в которых используется алгоритм Дейкстры. В библиотеке реализован также алгоритм Флойда (TGraph.CreateMinWeightPathsMatrix), позволяющий найти кратчайшие пути между всеми парами вершин. Алгоритм Флойда применим к ографам с дугами отрицательного веса; при наличии контуров отрицательной длины алгоритм их обнаруживает.

Геометрические графы

В геометрических графах для каждой вершины графа определены вещественные координаты: двумерные (X, Y) или трехмерные (X, Y, Z) (соответственно, Geom2D или Geom3D). В настоящее время в библиотеке определен только ряд вспомогательных методов для работы с такими графами, однако в будущем планируется реализовать алгоритмы визуализации графов.

7. Реализованные алгоритмы

В библиотеке AGraph реализованы алгоритмы решения многих теоретико-графовых задач, в том числе:

* поиск путей с минимальным количеством промежуточных вершин между заданными вершинами графа;
* поиск путей минимальной длины во взвешенном графе;
* определение связности графа;
* определение отношения ребра к кольцевым системам;
* нахождение компонент связности графа;
* нахождение сильных компонент ориентированного графа;
* построение матрицы связности и обобщенной матрицы связности графа;
* построение матрицы достижимости графа;
* построение матрицы расстояний графа;
* поиск максимальных независимых множеств вершин графа;
* поиск системы независимых минимальных колец графа (исключая петли), покрывающих все кольцевые ребра графа;
* построение графа, дополнительного к заданному графу;
* построение реберного графа для заданного графа;
* построение кратчайшего остовного дерева для заданного взвешенного графа;
* поиск максимального потока в транспортной сети;
* поиск хроматического числа и оптимальной вершинной раскраски графа;
* поиск эйлерова цикла в графе;
* поиск гамильтоновых циклов в орграфе;
* решение задачи почтальона для взвешенного графа с неотрицательными весами ребер;
* определение планарности графа;
* распознавание изоморфизма графов с учетом атрибутов.

В качестве источников алгоритмов использовались монографии и статьи по теории графов. При решении задач, для которых известны алгоритмы полиномиальной сложности, использовались, в основном, алгоритмы, которые являются асимптотически оптимальными среди всех известных алгоритмов решения данной задачи, или близки к оптимальным. Для некоторых из задач, решаемых библиотекой, алгоритмы полиномиальной сложности не существуют или неизвестны. В таком случае приходится использовать переборные алгоритмы, приближенные алгоритмы или алгоритмы, обеспечивающие эффективное решение для частных случаев задачи. Библиотека AGraph предоставляет ряд алгоритмов, которые относятся ко второй и третьей из этих категорий (например, приближенный алгоритм вершинной раскраски графов). В то же время, основное внимание в библиотеке уделялось реализации универсальных алгоритмов. Для некоторых "трудных" задач переборные алгоритмы были реализованы самостоятельно; для решения других в библиотеку были перенесены наиболее эффективные программные реализации, которые удалось найти (разумеется, в том случае, когда авторы программ допускают такое использование). Так, функция распознавания изоморфизма графов основана на программе, разработанной М.Венто [5]; функция нахождения хроматического числа и оптимальной вершинной раскраски графа основана на программе, разработанной М.Триком [6].

Важнейшей проблемой при разработке программного обеспечения является обеспечение его корректности. Необходимой составляющей для достижения корректности является использование корректных алгоритмов, однако правильная и эффективная реализация алгоритмов также является нетривиальной задачей. При создании библиотеки AGraph принимались различные меры для обнаружения и исправления ошибок. Во-первых, в отладочном режиме методы классов библиотек AGraph и Vectors выполняют проверки предусловий и правильности передаваемых параметров; в случае обнаружения ошибки возбуждается исключительная ситуация (exception). Во-вторых, все изменения активно тестируются как в ручном, так и в автоматическом режиме, для чего были созданы соответствующие программные тесты. Разумеется, все эти методы не гарантируют отсутствие ошибок, однако их использование позволило значительно повысить надежность библиотеки.

С технологической точки зрения алгоритмы можно разделить на две категории: первые из них реализованы в виде методов класса TGraph (модуль graphs.pas), вторые - в отдельных модулях. Реализация алгоритмов в модуле graphs.pas позволяет достичь максимальной эффективности за счет доступа к деталям внутреннего представления графа (т.е. к защищенным полям и методам классов TVertex, TEdge и TGraph), поэтому таким способом реализованы, в основном, "базовые" алгоритмы: поиск путей, определение компонент связности графа и т.д.

8. Ввод/вывод графов

Одной из проблем при создании средств работы с помеченными графами является выбор внешнего файлового формата для хранения графов. До недавнего времени каждая программная система использовала свой собственный, уникальный формат, что приводило к сложностям при организации обмена данными. Относительно недавно разработчики системы Graphlet предложили универсальный переносимый формат файла для представления помеченных графов - GML (Graph Modelling Language) [7]. В настоящее время GML-формат поддерживается многими прикладными программами и библиотеками для работы с графами.

GML-файл состоит из пар ключ-значение. Примерами ключей являются стандартные ключи **graph**, **node** и **edge**. Значениями могут быть целые и вещественные числа, строки и списки; в свою очередь, списки также содержат пары ключ-значение, в том числе вложенные списки. Важным достоинством формата GML является его открытость и расширяемость: любой разработчик имеет право определять свои ключи для хранения необходимой информации. В настоящее время этот формат поддерживается многими прикладными программами и библиотеками для работы с графами. Библиотека AGraph также поддерживает запись и чтение графов в GML-формате, но с некоторыми ограничениями (для хранения строк не используется кодировка ISO 8859).

Наряду с форматом GML, библиотека поддерживает запись графов в поток и чтение их из потока с использованием двоичного формата (методы TGraph.WriteToStream и TGraph.ReadFromStream). Данный способ обеспечивает более высокую скорость записи/чтения графов и приводит к созданию файлов меньшего размера, однако не является переносимым.

9. Создание специализированных классов графов

Библиотека AGraph предоставляет гибкие средства (механизм поддержки динамических атрибутов и различных видов графов), позволяющие использовать ее для решения самых разных прикладных задач. Во многих случаях пользователю хватит возможностей, предоставляемых основным классом библиотеки TGraph. В то же время, создание специализированных классов графов оправдано, если это позволяет облегчить работу с библиотекой и/или повысить эффективность прикладных программ.

Примером специализированного класса графов является класс TChemGraph, предназначенный для работы с молекулярными графами. Данный класс является непосредственным потомком класса TGraph и поддерживает работу с молекулярными графами на уровне атомов и связей (см. пример 8). Для хранения необходимых данных используются атрибуты, но в целях ускорения доступа к ним вместо методов используется доступ по смещениям. TChemGraph обеспечивает также запись и чтение молекулярных графов с использованием распространенных MOL- и SDF-форматов.

// создание молекулярного графа

G:=TChemGraph.Create;

// добавление атомов и связей

A:=G.AddAtom(CarbonAtom); // добавить 'C'

G.AddAtom(AtomTbl.SearchName('N')); // добавить 'N'

G.AddAtom(AtomTbl.SearchName('Cl')); // добавить 'Cl'

G.AddBond(A, G[1], DoubleBond);

G.AddBond(A, G[2], SingleBond);

// свойства и методы, специфичные для молекулярных графов

G.Atom[1]:=CarbonAtom; // заменить 'N' на 'C'

S1:=G.AtomName[1]; // S1 = 'C'

S2:=G.BruttoFormula; // S2 = 'С2Сl1'

Пример 8. Использование класса TChemGraph.

10. Эффективность

При создании библиотеки AGraph в качестве основных целей были поставлены обеспечение универсальности и простоты использования библиотеки. Соображения эффективности учитывались в той мере, в какой это не противоречило достижению данных целей. В то же время, решение многих прикладных задач требует обработки графов большого размера, и возможность решения этих задач на доступных вычислительных средствах напрямую зависит от эффективности реализации тех или иных алгоритмов.

Для оценки эффективности средств библиотеки AGraph было осуществлено решение ряда тестовых задач; те же задачи решались с помощью библиотеки LEDA. Поскольку данные библиотеки используют разные внутренние представления графов, различные методы привязки атрибутов к вершинам и ребрам графа, а также способы передачи параметров и возвращения результатов, прямое сравнение результатов этих испытаний не совсем корректно. Тем не менее, результаты показывают, что скоростные характеристики библиотек AGraph и LEDA являются, по крайней мере, сопоставимыми (см. таблицу 1).

При тестировании использовались следующие программные и аппаратные средства.

* **ЭВМ**: персональный компьютер на процессоре AMD K6-2 400 (частота системной шины 100 MHz), кэш второго уровня 512 Kb, ОЗУ 64 Mb.
* **ОС**: Windows 95 OSR 2.1.
* **Версии библиотек**: AGraph v.990915, LEDA 3.8.
* **Компиляторы**: для AGraph - Delphi 3.0, для LEDA - MS Visual C++ 5.0 (в обоих случаях отладочные проверки были выключены, использовалась максимальная оптимизация).

|  |  |  |
| --- | --- | --- |
|  | **AGraph** | **LEDA** |
| ***количество вершин |V|=100000, количество ребер |E|=200000\**** | | |
| **нахождение пути минимальной длины** | 0.4 с | 0.6 с |
| **нахождение пути минимального суммарного веса (граф интерпретировался как неориентированный)** | 1.5 с (вещественные веса) | 1.9 с (целые веса);  3.2 с (вещественные веса) |
| **нахождение пути минимального суммарного веса (граф интерпретировался как ориентированный)** | 1.3 с (вещественные веса) | 1.1 с (целые веса);  1.9 с (вещественные веса) |
| **нахождение связных компонент** | 0.6 с | 0.4 с |
| **нахождение сильных компонент (граф интерпретировался как ориентированный)** | 0.7 с | ошибка времени исполнения (переполнение стека) |
| **построение минимального остовного дерева** | 5.8 с | 4.8 с |

\* В библиотеке AGraph хранение графа такого размера потребовало около 26 Мб оперативной памяти и 21 Мб на диске в формате GML.

Литература

1. Нечепуренко М.И., Попков В.К., Майнагашев С.М. и др. Алгоритмы и программы решения задач на графах и сетях. - Новосибирск, Наука (сибирское отделение), 1990.
2. Mehlhorn K., Naher St. The LEDA Platform of Combinatorial and Geometric Computing. - Cambridge University Press, 1999.
3. Цыпнятов Е. Библиотека классов для программирования задач теории графов, дипломная работа. - Нижний Новгород, 1998.
4. Object Pascal Language Guide. Borland Delphi 3 for Windows 95 and Windows NT - Borland International Inc., 1997.
5. Cordella L.P., Foggia P., Sansone C., Vento M. An Efficient Algorithm for the Inexact Matching of ARG Using a Contextual Transformational Model. / Proc. of the 13th ICPR, IEEE Computer Society Press, 1996, vol.III, pp.180-184.
6. Mehrotra A., Trick M.A. A Column Generation Approach for Exact Graph Coloring / INFORMS Journal on Computing, 8:4, 1996.
7. Himsolt M. GML: A Portable Graph File Format / Technical Report, Universitat Passau, 1997, cf.; см. также краткое описание GML.