БИБЛИОТЕКА ПРОГРАММИСТА

Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн

C++ Boost Graph Library

- Обобщенное программирование на C++
- Концепции и алгоритмы BGL
- Алгоритмы в интернет-маршрутизации
- Задачи планирования сети

Addison-Wesley

ПИТЕР®

ПИТЕР®

Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine

The Boost Graph Library

User Guide and Reference Manual

Addison-Wesley

Boston • San Francisco • New York • Toronto
Montreal • London • Munich • Paris • Madrid • Capetown
Sydney • Tokyo • Singapore • Mexico City

БИБЛИОТЕКА ПРОГРАММИСТА

Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн

C++ Boost Graph Library

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2006

Джереми Сик, Лай-Кван Ли, Эндрю Ламсдэйн

C++ Boost Graph Library. Библиотека программиста

Перевел с английского Р.Сузи

Главный редактор	*Е. Строганова*
Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Адаменко*
Научный редактор	*В. Лаптев*
Иллюстрации	*В. Демидова, Л. Родионова*
Литературный редактор	*Е. Яковлева*
Художник	*Л. Адуевская*
Корректоры	*Н. Викторова, И. Смирнова*
Верстка	*Л. Родионова*

ББК 32.973-018.1
УДК 004.43

Дж. Сик, Л. Ли, Э. Ламсдэйн

С35 C++ Boost Graph Library. Библиотека программиста / Пер. с английского Сузи Р. — СПб.: Питер, 2006. — 304 с.: ил.

ISBN 5-469-00352-3

Издание, являющееся переводом одной из книг серии «C++ in Depth», посвящено описанию Boost Graph Library (BGL) — библиотеки для построения структур данных и алгоритмов вычислений на графах, предназначенных для решения самых разнообразных задач: от оптимизации интернет-маршрутизации и планирования телефонных сетей до задач молекулярной биологии. Содержит развернутое описание BGL, демонстрирует примеры приложений к реальным задачам. Первая часть является полным руководством пользователя, начинается с введения понятий теории графов, терминологии и описания обобщенных алгоритмов на графах, знакомит пользователя со всеми основными возможностями библиотеки BGL. Вторая часть — полное справочное руководство, содержит документацию ко всем концепциям BGL, ее алгоритмам и классам.

ISBN 5-469-00352-3

ISBN 0-201-72914-8 (англ.)

ООО «Питер Принт», 194044, Санкт-Петербург, Б. Сампсониевский пр., д. 29а.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано к печати 28.09.05. Формат 70×100/16. Усл. п. л. 24,51. Тираж 2000. Заказ 375

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29

Краткое содержание

Содержание

Ричарду и Элизабет
Дж. Г. С.

Юн
L-Q. L.

Вэнди, Бену, Эмили и Бетани
А. Л.

Предисловие

Когда я впервые увидел эту книгу, я почувствовал зависть. В конце концов желание создать библиотеку, подобную Boost Graph Library (BGL), привело меня к открытию обобщенного программирования. В 1984 году я вошел в профессорско-преподавательский состав Политехнического университета в Бруклине, имея некоторые, еще довольно смутные, идеи построения библиотек программных компонентов. По правде говоря, они были на втором плане: в то время мои истинные интересы касались формальных обоснований естественного языка, кое в чем напоминающего Органона Аристотеля, но более полного и формального. Я был, наверное, единственным доцентом на кафедрах информатики и электротехники, кто собирался получить должность, внимательно изучая категории Аристотеля. Интересно заметить, что дизайн Standard Template Library (STL), в частности лежащий в основе онтологии объектов, базируется на моем понимании того, что отношение «целое-часть» является основополагающим отношением, которое описывает реальный мир, и что оно вообще не похоже на отношение «элемент-множество», известное нам из теории множеств. Реальные объекты не имеют общих частей: моя нога не является еще чьей-то ногой. То же самое можно сказать и о контейнерах STL. Например операции `std::list::splice` перемещают части из одного контейнера в другой подобно операции по пересадке органов: моя почка — это моя почка, до тех пор, пока ее не пересадят кому-нибудь другому.

В любом случае, я был твердо убежден, что программные компоненты должны быть функциональными и основываться на системе функционального программирования Джона Бэкуса (John Backus's FP system). Единственной новой идеей было то, что функции могут быть связаны с некоторыми аксиомами. Например, «алгоритм русского крестьянина», позволяющий вычислить n-ю степень за $O(\log n)$ шагов, подходит для любого объекта, для которого определена бинарная ассоциативная операция. Другими словами, я верил, что алгоритмы должны быть связаны с тем, что сейчас мы называем концепциями (см. раздел 2.3 этой книги), которые я обозначал как структурные типы (structure types), а теоретики называют *многосортными алгебрами* (multi-sorted algebras).

Удачей для меня было то, что в Политехническом университете был замечательный человек — Аарон Кершенбаум, обладавший глубокими знаниями в области

алгоритмов на графах и стремлением их реализовать. Аарон заинтересовался моими попытками расчленить программы на простые примитивы и провел много времени, обучая меня графовым алгоритмам и работая со мной над их реализацией. Он также показал мне, что есть фундаментальные вещи, которые нельзя сделать в функциональном стиле без высоких изменений в сложности. Хотя я часто мог реализовать алгоритмы, требующие линейного времени, в функциональном стиле без изменения асимптотической сложности, реализовать на практике алгоритмы с логарифмическим временем без изменения сложности на линейную я не мог. В частности, Аарон объяснял мне, почему очередь по приоритетам столь важна для алгоритмов на графах. Он был хорошо осведомлен в этом вопросе: Кнут в его книге по Stanford GraphBase [22] приписывает авторство Аарону в применении частично упорядоченных бинарных деревьев (binary heaps) к алгоритмам Прима и Дейкстры.

Мы очень обрадовались, когда смогли получить алгоритмы Прима и Дейкстры как два случая одного обобщенного алгоритма (высокоуровневого). Это просто замечательно, что код BGL схож с нашим кодом (см. например сноску в разделе 13.4.2). Следующий пример на языке Scheme показывает, как два алгоритма были реализованы в терминах одного высокоуровневого алгоритма. Единственная разница заключается в том, как комбинируются значения расстояний: сложением у Дейкстры и выбором второго операнда у Прима:

```
(define dijkstra
  (make-scan-based-algorithm-with-mark
    make-heap-with-membership-and-values + < ))

(define prim
  (make-scan-based-algorithm-with-mark
    make-heap-with-membership-and-values (lambda (x y) y) < ))
```

На поиск подходящего языка программирования для эффективной реализации подобного стиля у меня ушло почти десять лет. Наконец я нашел C++, который помог мне создавать программы, полезные для общества. Более того, C++ серьезно повлиял на мой проект, основанный на модели машины для C. Шаблоны и перегрузка функций — вот особенности C++, позволившие создать STL.

Я часто слышу от людей выражения недовольства по поводу перегрузки в C++, но, как это обычно бывает для большинства полезных механизмов, перегрузка может быть использована неправильно. Для разработки полезных абстракций перегрузка незаменима. Если мы обратимся к математике, то многие ее идеи получили развитие именно благодаря перегрузке. В качестве примера можно привести расширение понятия чисел от натуральных к целым, к рациональным, к гауссовым, к p-адическим числам и т. д. Можно легко догадаться о некоторых вещах без знания точных определений. Если я вижу выражение, в котором используются операции сложения и умножения, я предполагаю наличие дистрибутивности. Если я вижу знак «меньше» и сложение, то предполагаю, что если $a < b$, то $a + c < b + c$ (я редко складываю несчетные количества). Перегрузка позволяет нам переносить знание от одного типа к другому.

Важно понять, что можно писать обобщенные алгоритмы, просто используя перегрузку, без шаблонов, но это требует много нажатий на клавиши. То есть для каждого класса, например удовлетворяющего требованиям итератора произвольного доступа, нужно вручную определять все относящиеся к нему алгоритмы. Это утомительно, но осуществимо (нужно определять только сигнатуры: тела будут те же). Нужно заметить, что настраиваемые модули (generics) в языке Ада требу-

ют ручной реализации и, поэтому, не так полезны, поскольку каждый алгоритм нужно реализовывать вручную. Шаблоны C++ решают эту проблему, позволяя определять такие вещи единожды.

Еще есть вещи, нужные в обобщенном программировании, но непредставимые в C++. *Обобщенные алгоритмы* — это алгоритмы, которые работают с объектами, обладающими похожими интерфейсами. Не идентичными, как в объектно-ориентированном программировании, а именно похожими. Использование не только бинарных методов вызывает проблему (см. раздел 2.1.3), интерфейсы фактически описываются с помощью одного типа (односортная алгебра). Если мы внимательно посмотрим на объекты вроде итераторов, то увидим, что они могут быть описаны только в терминах нескольких типов: тип самого итератора, тип значения и тип расстояния. Другими словами, необходимо три типа, чтобы определить интерфейсы для одного типа. В C++ для этого нет нужного аппарата. В результате мы не можем определить итераторы и, следовательно, откомпилировать обобщенные алгоритмы. Например, если мы определим алгоритм reduce таким образом:

```
template <class InputIterator, class BinaryOperationWithIdentity>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first, InputIterator last,
       BinaryOperationWithIdentity op)
{
  typedef typename iterator_traits<InputIterator>::value_type T;
  if (first == last) return identity_element(op);
  T result = *first;
  while (++first != last) result = op(result, *first);
  return result;
}
```

но вместо ++first != last напишем ++first < last, никакой компилятор не сможет обнаружить ошибку в месте определения. Хотя стандарт ясно декларирует, что operator< не нужен для итераторов ввода, у компилятора нет возможности знать об этом. Требования к итераторам сформулированы только на словах. Мы пытаемся программировать с концепциями (многосортными алгебрами) на языке, в котором для них нет поддержки.

Насколько сложно расширить C++, чтобы действительно позволить этот стиль программирования? Во-первых, нам нужно ввести концепции как новое средство интерфейса. Например, мы можем определить следующие концепции:

```
concept SemiRegular : Assignable, DefaultConstructible {};
concept Regular : SemiRegular, EqualityComparable {};
concept InputIterator : Regular, Incrementable {
  SemiRegular value_type;
  Integral distance_type;
  const value_type& operator*();
};

value_type(InputIterator)
reduce(InputIterator first, InputIterator last,
       BinaryOperationWithIdentity op)
(value_type(InputIterator) == argument_type(BinaryOperationWithIdentity))
{
  if (first == last) return identity_element(op);
  value_type(InputIterator) result = *first;
  while (++first != last) result = op(result, *first);
  return result;
}
```

Обобщенные функции (generic functions) — это функции, которые принимают концепции как аргументы и в добавление к списку аргументов имеют список ограничений типов. Теперь полная проверка типов может быть выполнена в месте определения, без рассмотрения мест вызова или в местах вызова без рассмотрения тела алгоритма.

Иногда требуется несколько реализаций одной и той же концепции. Например, для слияния возможна следующая запись:

```
OutputIterator merge(InputIterator[1] first1, InputIterator[1] last1,
                     InputIterator[2] first2, InputIterator[2] last2,
                     OutputIterator result)
(bool operator<(value_type(InputIterator[1]), value_type(InputIterator[2])),
value_type(InputIterator[1]) == value_type(InputIterator[2]),
output_type(OutputIterator) == value_type(InputIterator[2]));
```

Заметим, что это слияние не такое эффективное, как в STL. Невозможно соединить список чисел с плавающей запятой и вектор чисел двойной точности в дек целых чисел. Но алгоритмы STL часто производят неожиданные и, по моему мнению, нежелательные преобразования типов. Если нужно слить числа с двойной точностью и числа с плавающей запятой в целые, лучше использовать явный функциональный объект для сравнения и специальный итератор вывода для преобразования.

В языке C++ поддерживается два различных механизма абстракции: объектно-ориентированный подход и шаблоны. Использование объектно-ориентированного подхода позволяет точно определить интерфейс и провести диспетчеризацию времени исполнения. Вместе с этим диспетчеризация бинарных или мультиметодов неосуществима, а связывание времени исполнения часто неэффективно. Шаблоны предназначены для более сложных интерфейсов и разрешены во время компиляции. Серьезным препятствием для их использования разработчиками программного обеспечения является тот факт, что в шаблонах отсутствует разделение между интерфейсами и реализацией. Например, недавно я пытался откомпилировать пример STL-программы из десяти строк, используя один из наиболее популярных компиляторов C++... Я был в шоке, увидев несколько страниц неразборчивых сообщений об ошибках. Можно предположить, что введение концепций объединит возможности обоих подходов и уберет ограничения, накладываемые ими. Кроме того, можно представить концепции в виде виртуальных таблиц, распространяющихся на указатели к описателям типов: виртуальная таблица для итератора ввода содержит не только указатели на `operator*` и `operator++`, но и указатели на актуальный тип итератора, тип его значения и тип расстояния. А затем можно ввести указатели и ссылки на концепции!

Обобщенное программирование — это сравнительно молодое направление в программировании. Я счастлив наблюдать, что небольшая попытка, начатая двадцать лет назад Дэйвом Массером, Дипаком Капуром, Аароном Кершенбаумом и мной, привела к появлению библиотек нового поколения, таких как BGL и MTL. Я должен поздравить Университет Индианы с лучшей командой по обобщенному программированию в мире. Я уверен, что они сотворят и другие чудеса!

Александр Степанов.

Пало-Альто, Калифорния. Сентябрь, 2001[1]

---

[1] Я бы хотел поблагодарить Джона Вилкинсона, Марка Манассэ, Марка Наджорка и Джереми Сика за многие ценные предложения.

Введение

Графовая абстракция — это мощный инструмент решения задач, используемый для описания отношений между дискретными объектами. Практические задачи могут быть смоделированы в виде графов для различных областей, например таких, как маршрутизация пакетов в Интернете, проектирование телефонной сети, системы сборки программного обеспечения, поисковые машины WWW, молекулярная биология, системы автоматизированного планирования дорожного маршрута, научные вычисления и т. п. Достоинством графовой абстракции является тот факт, что найденное решение проблемы теории графов может быть использовано для решения проблем в широком диапазоне областей. Например, задача нахождения выхода из лабиринта и задача нахождения групп взаимно достижимых веб-страниц могут быть решены с помощью поиска в глубину — важнейшего положения из теории графов. При сосредоточении на сути этих задач, а именно на графовой модели, описывающей дискретные объекты и отношения между ними, специалисты по теории графов нашли решения не просто для «горстки» отдельных проблем, а для целых семейств задач.

Сразу же возникает вопрос. Если теория графов всеобще и широко применяется для произвольных сфер задач, не должно ли программное обеспечение, реализующее графовые алгоритмы, быть таким же универсальным в применении? Может показаться, что теория графов — это идеальная область для повторного использования программного кода. Однако до сих пор потенциальное повторное использование было далеко от реальности. Графовые задачи редко встречаются в чистой теоретико-графовой форме, они чаще включены в более крупные проблемы, зависящие от области применения. В результате данные, которые можно смоделировать как граф, зачастую явно не представлены как граф, и тогда они закладываются в некоторую структуру данных, специфичную для приложения. Даже в случае, когда данные приложения явно представлены в виде графа, конкретное представление, выбираемое программистом, может не совпадать с представлением, ожидаемым библиотекой. Более того, различные приложения могут накладывать разные ограничения к временной и пространственной сложности графовых структур данных.

Эти особенности являются серьезной проблемой для разработчика библиотеки алгоритмов на графах, который хочет предоставить повторно используемое программное обеспечение. Невозможно предусмотреть все возможные структуры данных, которые могут потребоваться, и написать различные версии графового алгоритма специально для каждой из них. В настоящее время алгоритмы на графах пишутся в терминах той структуры данных, которая наиболее удобна для алгоритма, и пользователи должны преобразовывать их структуры данных к такому формату, чтобы применить алгоритм. Это неэффективное решение, поглощающее время программиста и вычислительные ресурсы. Часто затраты на преобразования оказываются слишком высокими, и программист переписывает алгоритм в терминах своей собственной структуры данных. Этот подход отнимает время и способствует появлению ошибок, а также имеет тенденцию приводить к недостаточно эффективным решениям, поскольку программист приложения может не быть экспертом в области графовых алгоритмов.

Обобщенное программирование

Стандартная библиотека шаблонов (Standard Template Library, STL) [40] появилась в 1994 году и была сразу принята в стандарт C++. *STL* — библиотека взаимозаменяемых компонентов для решения многих фундаментальных задач на последовательностях элементов. Отличие библиотеки STL от предлагаемых ранее библиотек состоит в том, что каждый STL-алгоритм может работать с широким набором последовательных структур данных: связные списки, массивы, множества и т. п. Абстракция итератора обеспечила интерфейс между контейнерами и алгоритмами, и шаблонный механизм C++ предоставил нужную гибкость в реализации без потери эффективности. Каждый алгоритм в STL является шаблоном функции, параметризованным по типам итераторов, с которыми он работает. Любой итератор, который удовлетворяет минимальному набору требований, может быть использован независимо от структуры данных, обходимой итератором. Системный подход, использованный в STL для построения абстракций и взаимозаменяемых компонентов, называется *обобщенным программированием*.

Обобщенное программирование хорошо зарекомендовало себя при решении проблемы повторного использования кода для библиотек алгоритмов на графах. В рамках обобщенного программирования алгоритмы на графах могут быть сделаны более гибкими и легко используемыми в большом наборе приложений. Каждый графовый алгоритм пишется не в терминах специфической структуры данных, а для графовой абстракции, которая может быть реализована многими различными структурами данных. Написание обобщенных графовых алгоритмов имеет дополнительное преимущество, являясь более естественным. Абстракция, свойственная псевдокоду описания алгоритма, сохраняется в обобщенной функции.

Библиотека Boost Graph Library (BGL) — это первая библиотека графов C++, применяющая понятия обобщенного программирования при создании алгоритмов на графах.

Немного из истории BGL

Boost Graph Library появилась как библиотека обобщенных графовых компонентов (Generic Graph Component Library, GCCL) в лаборатории научных вычислений (Lab for Scientific Computing, LSC). Эта лаборатория под руководством профессора Эндрю Ламсдэйна охватывала различные сферы деятельности, занималась исследованиями алгоритмов, программного обеспечения, инструментов и систем времени выполнения для вычислительной науки и техники[1]. Особое внимание было уделено разработке промышленного, высокопроизводительного программного обеспечения с использованием современных языков программирования и методов, в том числе обобщенного программирования.

Вскоре после того как была выпущена STL, в LSC началась работа по применению обобщенного программирования к научным расчетам. Библиотека матричных шаблонов (Matrix Template Library, MTL) была одним из первых проектов. Многие уроки, усвоенные во время создания MTL, были учтены при проектировании и реализации GGCL.

Одним из важных классов вычислений линейной алгебры в научных расчетах является класс вычислений с разреженными матрицами, в котором графовые алгоритмы играют большую роль. Когда лаборатория LSC разрабатывала методы обработки разреженных матриц для MTL, необходимость в высокопроизводительных повторно используемых (и обобщенных) графовых алгоритмах стала очевидна. Однако ни в одной из графовых библиотек, доступных в то время (LEDA, GTL, Stanford GraphBase), не использовался обобщенный стиль программирования в отличие от STL или MTL. Таким образом, данные библиотеки не удовлетворяли требованиям LSC в гибкости и производительности. Другие исследователи также были заинтересованы в создании обобщенной библиотеки алгоритмов на графах для C++. Во время встречи с Бьерном Страуструпом мы познакомились с некоторыми людьми из «AT&T», тоже нуждавшимися в такой библиотеке. Другая ранняя работа в области графовых алгоритмов включала отдельные примеры кодов, написанных Александром Степановым, а также Дитмаром Кюлем в его магистерской диссертации.

Джереми Сик, вдохновленный домашними упражнениями по алгоритмам для своего курса, начал создавать прототипы интерфейса и некоторых графовых классов весной 1998 года с учетом более ранних разработок. Затем Лай-Кван Ли разработал первую версию GGCL, которая стала его магистерским диссертационным проектом.

В следующем году авторы начали сотрудничать с Александром Степановым и Мэтью Остерном. В это время реализация Степанова для компонент связности на основе непересекающихся множеств была добавлена к GGCL, и началась работа по документированию концепций для GGCL подобно документации Остерна для STL.

В том же году авторам стало известно о Boost, и они были обрадованы тем, что нашли организацию, заинтересованную в создании высококачественных библиотек

[1] С тех пор LSC была преобразована в лабораторию открытых систем (Open Systems Laboratory, OSL). Хотя название и местоположение изменились, программа работы остается прежней. Дополнительная информация находится на веб-сайте OSL http://www.osl.iu.edu.

C++ с открытыми исходными кодами. В Boost было несколько человек, уделявших внимание обобщенным графовым алгоритмам, и одним из этих людей был Дитмар Кюль. Некоторые обсуждения обобщенных интерфейсов для графовых структур привели к пересмотру GGCL и появлению в ней новых интерфейсов, очень похожих на те, что есть в Boost Graph Library сейчас.

4 сентября 2000 года GGCL была формально рецензирована под руководством Дэвида Абрахамса и стала Boost Graph Library. Первый выпуск BGL состоялся 27 сентября 2000 года. BGL не является «замороженной» библиотекой. Она продолжает расти и развиваться для наибольшего удовлетворения потребностей своих пользователей. Мы приглашаем читателей присоединиться к группе Boost для работы над расширением BGL.

Что такое Boost?

Boost — сетевое сообщество, которое поддерживает разработку и проводит коллегиальную оценку свободных библиотек для C++. Особое внимание уделяется переносимым и высококачественным библиотекам, которые хорошо работают совместно (и «в том же духе») со стандартной библиотекой C++. Члены сообщества предоставляют предложения (проекты и реализации библиотек) для объективной оценки. Сообщество Boost (под управлением менеджера по рецензированию) рассматривает библиотеку, обеспечивает обратную связь с участниками и принимает решение о включении библиотеки в набор Boost-библиотек. Библиотеки доступны с веб-сайта http://www.boost.org. Кроме того, список рассылки Boost является важным местом для обсуждения планов и организации сотрудничества.

Получение и установка программного обеспечения BGL

Библиотека алгоритмов на графах Boost Graph Library доступна как часть коллекции библиотек Boost. Свежий выпуск библиотек Boost можно загрузить с помощью браузера по следующим адресам: http://www.boost.org/boost_all.zip (zip-архив для Windows), http://www.boost.org/boost_all.tar.gz (для Unix), а также по FTP из каталога ftp://boost.sourceforge.net/pub/boost/release/.

Zip-архив коллекции библиотек Boost можно разархивировать программой WinZip или ее аналогом. Tar-архив для Unix можно разархивировать с помощью такой команды:

```
gunzip -cd boost_all.tar.gz | tar xvf -
```

В результате создается каталог, имя которого состоит из слова boost и номера версии: например, разархивирование версии 1.31.0 создает каталог boost_1_31_0. В этом каталоге находятся два важных подкаталога: boost и libs. Подкаталог boost содержит заголовочные файлы для всех библиотек коллекции. Подкаталог libs имеет отдельные подкаталоги для каждой библиотеки в коллекции. Эти подкаталоги содержат файлы исходных кодов и документации к данным библиотекам.

Сориентироваться в архиве можно, открыв в браузере веб-страницу boost_1_31_0/index.htm.

Заголовочные файлы BGL находятся в каталоге boost/graph/. Однако для BGL нужны и другие заголовочные файлы, так как в библиотеке используются различные компоненты Boost. Гипертекстовая документация находится в каталоге libs/graph/doc/, а исходные коды примеров — в libs/graph/example/. В каталоге libs/graph/test/ содержатся тестовые наборы для BGL. Исходные файлы для реализации анализаторов Graphviz-файлов и программ печати расположены в libs/graph/src/.

Кроме того, что описано далее, для использования BGL не требуется компиляция и сборка. Все, что необходимо, — это добавить каталог заголовочных файлов Boost к пути поиска заголовочных файлов. Например, если в Windows 2000 библиотека версии 1.31.0 разархивирована в корень диска C, для компиляторов Borland, GCC и Metrowerks добавьте `-Ic:/boost_1_31_0` к командной строке компилятора, а для Microsoft Visual C++ — `/I "c:/boost_1_31_0"`. Для интегрированных сред разработки (IDE) укажите `c:/boost_1_31_0` (или то, во что вы переименовали этот каталог) к пути поиска заголовочных файлов, используя соответствующий диалог. Перед применением BGL-интерфейсов к LEDA или Stanford GraphBase последние должны быть установлены согласно инструкциям к ним. Для использования функции `read_graphviz()` (для чтения Graphviz-файлов «AT&T») необходимо собрать и скомпоновать дополнительную библиотеку из каталога boost_1_31_0/libs/graph/src.

Библиотека Boost Graph Library написана на ISO/IEC Standard C++ и компилируется большинством компиляторов C++. Чтобы получить последнюю информацию о поддержке различных компиляторов, на веб-сайте Boost загрузите страницу «Compiler Status» по адресу http://www.boost.org/status/compiler_status.html.

Как пользоваться книгой

Эта книга является одновременно руководством пользователя и справочным пособием по BGL. Она предназначена для того, чтобы читатель смог использовать BGL для решения задач на графах, встречающихся в реальной жизни. Книга должна представлять интерес для программистов, желающих более углубленно изучить обобщенное программирование. Хотя сейчас уже достаточно много книг о том, как использовать обобщенные библиотеки (что почти всегда означает применение стандартной библиотеки STL), в очень немногих действительно описывается, как создать обобщенное программное обеспечение. Тем не менее обобщенное программирование — это жизненно важная, новая парадигма разработки программного обеспечения. Мы надеемся, что с помощью примеров из этой книги читатель научится программировать обобщенно (а не только использовать обобщенные библиотеки), применять и расширять принципы обобщенного программирования за пределы контейнерных типов и алгоритмов STL.

В качестве третьего помощника к руководству пользователя и справочному пособию выступит сам исходный код библиотеки BGL. Код BGL предназначен не только для обучения и образования, но и для реального использования.

Для студентов, изучающих графовые алгоритмы и структуры данных, BGL предоставляет обширный набор алгоритмов. Студент может сконцентрироваться

на изучении важной теории, на которой основаны графовые алгоритмы, без риска «увязнуть» и отвлечься на слишком большое количество деталей реализации.

Для профессиональных программистов BGL предлагает высококачественные реализации структур данных и алгоритмов. Программисты смогут значительно сократить время разработки за счет надежности библиотеки. Время, которое было бы затрачено на разработку (и отладку) сложных графовых структур данных и алгоритмов, может быть использовано для других дел. Более того, гибкий интерфейс BGL позволит программистам применять графовые алгоритмы в ситуациях, когда граф может существовать лишь неявно.

Для теоретиков эта книга дает стимул к использованию обобщенного программирования для реализации теоретико-графовых алгоритмов. Алгоритмы, написанные с применением интерфейса BGL, смогут широко применяться и иметь возможность повторного использования в различных областях.

Мы предполагаем, что читатель хорошо понимает C++. Мы не пытаемся научить читателя C++ в этой книге, так как для этого есть много хороших источников (мы особенно рекомендуем [42] и [25]). Мы также подразумеваем некоторое знакомство с STL (см. [34] и [3]). Однако мы представляем наиболее передовые возможности C++, использованные для реализации обобщенных библиотек в целом и BGL в частности.

В книге также вводятся необходимые понятия теории графов, но без особого рассмотрения. Для детального знакомства с элементарной теорией графов см. [10].

Грамотное программирование

Примеры программ в этой книге представлены с использованием *стиля грамотного программирования* (literate programming style), разработанного Дональдом Кнутом. Стиль грамотного программирования состоит в написании исходного кода и документации в одном и том же файле. Затем специальная программа преобразует файл в «чистый» файл исходного кода и в файл документации с красиво напечатанным исходным кодом. При грамотном программировании легче гарантировать, что примеры кодов в книге действительно компилируются и запускаются и что они соответствуют тексту.

Исходный код для каждого примера разбит на части. Части могут содержать ссылки на другие части. Например, ниже приведена часть, названная «Определение функции сортировки слияниями», которая ссылается на другие части, названные «Разделить массив пополам и отсортировать каждую половину» и «Слить обе половины». Примеры часто могут начинаться с части, которая является схемой всего вычисления, после чего следуют части, заполняющие детали. Например, следующий шаблон функции является обобщенной реализацией алгоритма сортировки слияниями [10]. В алгоритме два шага: сортировка каждой части и слияние частей:

```
⟨ Определение функции сортировки слияниями ⟩ ≡
template <typename RandomAccessIterator, typename Compare>
void merge_sort(RandomAccessIterator first,
                RandomAccessIterator last, Compare cmp)
```

```
{
  if (first + 1 < last) {
    ⟨ Разделить массив пополам и отсортировать каждую половину ⟩
    ⟨ Слить обе половины ⟩
  }
}
```

Обычно размер каждой части ограничен несколькими строками кода, которые выполняют определенную задачу. Имена для частей выбраны так, чтобы передавать суть этой задачи.

```
⟨ Разделить массив пополам и отсортировать каждую половину ⟩ ≡
RandomAccessIterator mid = first + (last - first)/2;
merge_sort(first, mid, cmp);
merge_sort(mid, last, cmp);
```

Функция `std::inplace_merge()` осуществляет основную работу этого алгоритма, создавая единый отсортированный массив из двух массивов:

```
⟨ Слить обе половины ⟩ ≡
std::inplace_merge(first, mid, last, cmp);
```

Иногда для названия используется имя файла. Это означает, что часть записана в отдельный файл. Многие примеры в этой книге записаны в файлах и могут быть найдены в каталоге libs/graph/example/ в поставке Boost. В следующем примере название функции `merge_sort()` записано в заголовочном файле:

```
⟨ merge-sort.hpp ⟩ ≡
#ifndef MERGE_SORT_HPP
#define MERGE_SORT_HPP

  ⟨ Определение функции сортировки слияниями ⟩

#endif // MERGE_SORT_HPP
```

Благодарности

Мы обязаны поблагодарить множество людей, которые вдохновляли и поддерживали нас в разработке BGL и в написании этой книги.

Особую признательность хотелось бы выразить Александру Степанову и Дэвиду Массеру за их первые шаги в деле обобщенного программирования и постоянную поддержку нашей работы, а также за вклад в библиотеку BGL. Мы особенно благодарим Дэвида Массера за чтение корректуры к этой книге. Работа Мэттью Остерна по документированию концепций STL обеспечила фундамент для создания концепций BGL. Мы благодарим Дитмара Кюля за его работу над обобщенными графовыми алгоритмами и паттернами проектирования, особенно за абстракцию ассоциативного свойства. Нужно заметить, что работа была бы невозможна без выразительной мощи языка C++ Бьерна Страуструпа.

Дэвид Абрахамс, Дженс Морер, Дитмар Кюль, Беман Дэйвс, Гари Пауэл, Грег Колвин и другие участники Boost сделали ценный вклад в интерфейс BGL, внесли многочисленные предложения по улучшению, подробно вычитывали эту книгу. Мы также благодарим следующих пользователей BGL, чьи вопросы помогли усовершенствовать BGL (и эту книгу): Гордон Вудхал, Дэйв Лонгхорн, Джоэль Филипс, Эдвард Люк и Стефен Норт.

Мы также благодарим следующих людей за просмотр этой книги во время ее написания: Жана Кристиана ван Винкеля, Дэвид Массера, Бемана Дэйвса и Джеффри Сквайреса.

Большое спасибо нашему редактору Деборе Лафферти и координаторам выпуска Киму Арни Малкахи, Черли Фергюссону и Мерси Барнс, а также остальной части команды издательства «Addison-Wesley» — работать с вами было одно удовольствие.

В начале работа над BGL была поддержана грантом NSF ACI-9982205. Части BGL были завершены, когда третий автор был в годичном отпуске в Американской Национальной лаборатории Лоуренса Беркли (где первые два автора были случайными гостями). Все рисунки графов в этой книге были сделаны с использованием программы dot из пакета Graphviz.

Лицензия

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http://www.piter.com.

Часть I

Руководство пользователя

Введение

1

В этой главе будут приведены общие положения о некоторых интерфейсах и компонентах библиотеки для работы с графами Boost Graph Library (BGL). Мы начнем с краткого обзора терминологии теории графов. В качестве примера хорошо моделируемой графом системы будет выступать сеть интернет-маршрутизаторов. Обобщенные интерфейсы, определяемые в BGL, описываются в разделе 1.2. Конкретные графовые классы, реализующие эти интерфейсы, рассматриваются в разделе 1.3. Некоторые сведения об обобщенных алгоритмах на графах приведены в разделе 1.4 данной главы.

1.1. Немного терминологии из теории графов

Графовая модель сети интернет-маршрутизаторов показана на рис. 1.1. Маршрутизаторы помечены кружками с буквами, а соединения между ними подписаны средними величинами задержек передачи сигнала (в миллисекундах).

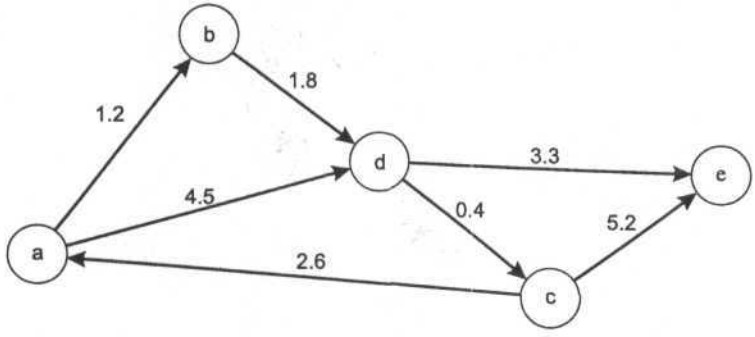


Рис. 1.1. Сеть интернет-маршрутизаторов

Если придерживаться терминологии теории графов, каждый маршрутизатор в примере сети является *вершиной* (vertex) (ее еще называют *узлом*), а каждое со-

единение — *ребром* (edge) (или *дугой*). Граф G состоит из *множества вершин* V и *множества ребер* E, что можно записать как $G = (V, E)$. Количество элементов множества вершин (число вершин графа) обозначается $|V|$, а число элементов множества ребер — $|E|$. Ребро записывается как пара, состоящая из вершин, которые соединяет ребро. Пара (u, v) обозначает ребро, соединяющее вершину u с вершиной v.

Сеть маршрутизаторов (рис. 1.1) может быть выражена в нотации теории множеств следующим образом:

$V = \{a, b, c, d, e\}$
$E = \{(a, b), (a, d), (b, d), (c, a), (c, e), (d, c), (d, e)\}$
$G = (V, E)$

Граф может быть *ориентированным* или *неориентированным*, в зависимости от того, ориентированы или нет ребра из множества ребер. Ребра ориентированного графа обычно называются дугами (хотя далее мы будем пользоваться термином ребро). Ребро ориентированного графа (орграфа) обозначается как упорядоченная пара (u, v), где u — *начальная вершина*, а v — *конечная вершина* ребра. Ребра (u, v) и (v, u) являются различными. В неориентированном графе ребро соединяет вершины в обоих направлениях, поэтому порядок вершин в ребре не имеет значения: (u, v) и (v, u) обозначают одно и то же ребро. Ребро, начало и конец которого совпадают, называется *петлей*. Такие ребра не допускаются в неориентированном графе. Два или более ребра, соединяющие одни и те же вершины, называются *параллельными*, а граф, имеющий такие ребра, — *мультиграфом*.

Если граф содержит ребро (u, v), то говорят, что вершина v *смежна* с вершиной u. Для ориентированного графа ребро (u, v) является *исходящим* для вершины u и *входящим* для вершины v. В неориентированном графе ребро (u, v) *инцидентно* вершине u (и вершине v). Множества смежности вершин графа на рис. 1.1 описываются следующим образом:

$Adjacent[a] = \{b, d\}$
$Adjacent[b] = \{d\}$
$Adjacent[c] = \{a, e\}$
$Adjacent[d] = \{c, e\}$
$Adjacent[e] = \{\}$

Далее записаны исходящие ребра для каждой вершины:

$OutEdges[a] = \{(a, b), (a, d)\}$
$OutEdges[b] = \{(b, d)\}$
$OutEdges[c] = \{(c, a), (c, e)\}$
$OutEdges[d] = \{(d, c), (d, e)\}$
$OutEdges[e] = \{\}$

Входящие ребра представлены аналогично:

$InEdges[a] = \{(c, a)\}$
$InEdges[b] = \{(a, b)\}$
$InEdges[c] = \{(d, c)\}$
$InEdges[d] = \{(a, d), (b, d)\}$
$InEdges[e] = \{(c, e), (d, e)\}$

1.2. Графовые концепции

Одна из основных задач обобщенной библиотеки — определить интерфейсы, которые позволят писать алгоритмы, не зависящие от конкретной структуры данных. Заметим, что под *интерфейсом* подразумевается не только набор прототипов функций, но и наборы синтаксических условий, таких как имена функций и их аргументов, семантических условий (вызываемые функции должны иметь определенные эффекты), гарантии той или иной сложности по времени и памяти.

Пользуясь терминологией из книги «Обобщенное программирование и STL» (см. главу «Дополнение к библиографии), мы используем слово концепция (concept) для обозначения этого более богатого определения интерфейса. *Стандартная библиотека шаблонов* (Standard Template Library, STL) определяет набор концепций итераторов, которые обеспечивают обобщенный механизм обхода и доступа к последовательностям объектов. Аналогично, рассматриваемая в этой книге BGL определяет набор концепций для изучения графов и манипулирования ими. В данном разделе мы рассмотрим эти концепции. Примеры раздела не относятся к конкретным типам графов. Они написаны как шаблоны функций, где граф является параметром шаблона. Обобщенная функция, написанная с помощью BGL, может применяться к любому типу графа из BGL или даже к типам графов, определенным пользователем. В разделе 1.3 мы обсудим конкретные графовые классы, поставляемые вместе с BGL.

1.2.1. Описатели вершин и ребер

В BGL вершинами и ребрами можно управлять при помощи «непрозрачных» манипуляторов, называемых *описателями вершин* (vertex descriptors) и *описателями ребер* (edge descriptors,). Графовые типы могут использовать различные типы для своих описателей. Например, некоторые графовые типы применяют целые числа, а другие — указатели. Типы дескрипторов[1] для графового типа всегда доступны через класс `graph_traits`. Более подробная информация по использованию классов свойств приведена в разделе 2.4, а класс `graph_traits` обсуждается, в частности, в разделе 14.2.1.

Описатели вершин выполняют очень примитивные функции. Их можно создать только при помощи конструктора, со значениями по умолчанию, затем скопировать при необходимости или сравнить на равенство. Описатели ребер подобны описателям вершин, но предоставляют доступ к связанным с ними вершинам. Следующий шаблон функции демонстрирует[2] реализацию обобщенной функции `is_self_loop()`, определяющей, является ли ребро петлей:

```
template <typename Graph>
bool is_self_loop(typename graph_traits<Graph>::edge_descriptor e,
                  const Graph& g) {
```

[1] Термины «описатель» и «дескриптор» означают одно и то же. — *Примеч. ред.*

[2] Из эстетических соображений для определения параметров шаблона мы используем typename вместо его аналога class.

```
    typename graph_traits<Graph>::vertex_descriptor u, v;
    u = source(e, g);
    v = target(e, g);
    return u == v;
}
```

1.2.2. Отображение свойств

Графы становятся полезными в виде моделей для задач из специфических областей при помощи ассоциирования объектов и величин с вершинами и ребрами. Например, для графа на рис. 1.1 каждая вершина имеет имя, состоящее из одного символа, а каждое ребро — величину задержки передачи сигнала. В BGL прикрепленные объекты или величины называются *свойствами*. Существует много возможностей для реализации, которые могут быть использованы для ассоциирования свойства с вершиной или ребром: члены структуры (struct); отдельные массивы, индексированные номером вершины или ребра; хэш-таблицы и т. п. Однако для написания обобщенных алгоритмов на графах нам нужен универсальный синтаксис для доступа к свойствам, независимый от того, как они хранятся. Этот универсальный синтаксис определяется концепциями *отображения свойства* (property map).

Отображение свойства — это объект, обеспечивающий отображение из множества объектов-ключей во множество объектов-значений. Концепции отображений свойств определяют только три функции:

- `get(p_map, key)` — возвращает объект-значение для ключа `key`;
- `put(p_map, key, value)` — присваивает значение `value` объекту-значению, соответствующему ключу `key`;
- `p_map[key]` — возвращает ссылку на объект-значение.

В следующем примере приведена обобщенная функция `print_vertex_name()`, которая выводит имя вершины v для данного отображения свойства `name_map`:

```
template <typename VertexDescriptor, typename VertexNameMap>
void print_vertex_name(VertexDescriptor v, VertexNameMap name_map)
{
  std::cout << get(name_map, v);
}
```

Аналогично задержка передачи сигнала для ребра может быть напечатана с помощью функции `print_trans_delay()`:

```
template <typename Graph, typename TransDelayMap, typename VertexNameMap>
void print_trans_delay(typename graph_traits<Graph>::edge_descriptor e,
const Graph& g, TransDelayMap delay_map, VertexNameMap name_map)
{
  std::cout << "trans-delay(" << get(name_map, source(e, g)) << ","
            << get(name_map, target(e, g)) << ") = " << get(delay_map, e);
}
```

Функции `print_vertex_name()` и `print_trans_delay()` будут использованы в следующем разделе.

Отображения свойств обсуждаются более подробно в главе 15, включая приемы создания отображений свойств, определяемых пользователем. Способы добавления свойств в граф и получения соответствующих отображений описаны в разделе 3.6.

1.2.3. Обход графа

Абстракция «граф» состоит из нескольких видов наборов (collections): вершины и ребра графа, исходящие и входящие ребра, смежные вершины для каждой вершины. По аналогии с STL, BGL использует итераторы для обеспечения доступа к каждому из этих наборов. Имеется пять видов итераторов для графов, один для каждого набора:

1. *Итератор вершин* используется для обхода всех вершин графа. Тип значения итератора вершин — описатель вершины.
2. *Итератор ребер* используется для обхода всех ребер графа. Тип значения итератора ребер — описатель ребра.
3. *Итератор исходящих ребер* применяется для доступа ко всем исходящим ребрам данной вершины *u*. Тип значения этого итератора — описатель ребра. Каждый описатель ребра, выдаваемый этим итератором, имеет *u* в качестве начальной вершины и смежную с *u* вершину в качестве конечной.
4. *Итератор входящих ребер* применяется для доступа ко всем входящим ребрам вершины *v*. Тип значения — описатель ребра. Каждый описатель ребра, выдаваемый этим итератором, имеет *v* в качестве конечной вершины и вершину, к которой *v* является смежной, в качестве начальной вершины.
5. *Итератор смежности* делает доступными вершины, смежные данной. Тип значения этого итератора — описатель вершины.

Как и описатели, каждый граф имеет свои собственные типы итераторов, доступные из класса `graph_traits`. Для каждого из только что описанных итераторов BGL интерфейс определяет функцию, возвращающую `std::pair` объектов-итераторов: первый итератор указывает на первый объект последовательности, а второй итератор указывает за ее конец. Функция `print_vertex_name()`, выводящая на печать имена всех вершин в графе, приведена в листинге 1.1.

Листинг 1.1. Функция для печати имен вершин графа

```
template <typename Graph, typename VertexNameMap>
void print_vertex_names(const Graph& g, VertexNameMap name_map)
{
 std::cout << "vertices(g) = { ";
      typedef typename graph_traits<Graph>::vertex_iterator iter_t;
      for (std::pair<iter_t, iter_t> p = vertices(g);
           p.first != p.second; ++p.first) {
        print_vertex_name(*p.first, name_map); std::cout << ' ';
      }
  std::cout << "}" << std::endl;
}
```

Применение этой функции к объекту-графу, моделирующему сеть маршрутизаторов (см. рис. 1.1), приводит к следующему результату:

```
vertices(g) = { a b c d e }
```

Функция `print_trans_delay()`, которая печатает задержки передачи сигнала, прикрепленные к каждому ребру графа, приведена в листинге 1.2. В функции используется функция `tie()` (из `boost/tuple/tuple.hpp`) для осуществления прямого присваивания из `std::pair` в скалярные переменные `first` и `last`.

Листинг 1.2. Печать задержек

```
template <typename Graph, typename TransDelayMap, typename VertexNameMap>
void print_trans_delays(const Graph& g, TransDelayMap trans_delay_map,
                        VertexNameMap name_map)
{
  typename graph_traits<Graph>::edge_iterator first, last;
  for (tie(first, last) = edges(g); first != last; ++first) {
    print_trans_delay(*first, g, trans_delay_map, name_map);
    std::cout << std::endl;
  }
}
```

Вывод этой функции для графа с рис. 1.1 следующий:

```
trans-delay(a,b) = 1.2
trans-delay(a,d) = 4.5
trans-delay(b,d) = 1.8
trans-delay(c,a) = 2.6
trans-delay(c,e) = 5.2
trans-delay(d,c) = 0.4
trans-delay(d,e) = 3.3
```

В дополнение к функциям `vertices()` и `edges()` имеются `out_edges()`, `in_edges()` и `adjacent_vertices()`, которые используют в качестве аргументов описатель вершины и объект-граф и возвращают пару итераторов.

Многие алгоритмы не нуждаются во всем разнообразии доступных итераторов, иногда графовые типы не могут предоставить эффективные реализации для всех типов итераторов. Осторожнее применяйте в алгоритмах конкретные типы графов, нельзя требовать от них исполнения неподдерживаемых операций. Если вы попытаетесь использовать графовый тип, который не предоставляет требуемую алгоритмом операцию, возникнет ошибка компиляции. Сопровождающая такую ошибку информация поможет понять, какая операция не реализована. В разделе 2.5 этот материал описан более детально.

Операции, доступные для данного графового типа, приведены в документации к этому типу. Раздел «Модель для» из справочных глав 12–14 содержит информацию о предоставляемых операциях, в нем перечислены концепции, поддерживаемые данным графовым типом. Операции, требуемые некоторым алгоритмом, даны в документации к алгоритму, где перечислены концепции, требуемые каждым параметром алгоритма.

1.2.4. Создание и модификация графа

Библиотека BGL позволяет определить интерфейсы для добавления или удаления вершин и ребер графа. В этом разделе мы рассмотрим небольшой пример создания графа, моделирующего сеть маршрутизаторов, изображенных на рис. 1.1. Сначала используется функция `add_vertex()` для добавления к графу пяти узлов, представляющих маршрутизаторы. Функция `add_edge()` применяется для того, чтобы добавить соединения между маршрутизаторами.

```
template <typename Graph, typename VertexNameMap, typename TransDelayMap>
void build_router_network(Graph& g, VertexNameMap name_map,
                          TransDelayMap delay_map)
```

```
{
  ⟨ Добавить маршрутизаторы к сети ⟩
  ⟨ Добавить соединения сети ⟩
}
```

Функция add_vertex() возвращает описатель вершины для новой вершины. Мы используем этот описатель для присваивания вершине имени в отображении свойства имен:

```
⟨ Добавить маршрутизаторы к сети ⟩ ≡
typename graph_traits<Graph>::vertex_descriptor a, b, c, d, e;
a = add_vertex(g); name_map[a] = 'a';
b = add_vertex(g); name_map[b] = 'b';
c = add_vertex(g); name_map[c] = 'c';
d = add_vertex(g); name_map[d] = 'd';
e = add_vertex(g); name_map[e] = 'e';
```

Функция add_edge() возвращает std::pair, где первый член пары — описатель ребра для нового ребра, а второй — логический флаг, показывающий, добавлено ли ребро (в некоторых графовых типах невозможно добавить ребро, если ребро с тем же началом и концом уже есть в графе). Код добавления соединений сети приведен в листинге 1.3.

Листинг 1.3. Добавление соединений сети

```
⟨ Добавить соединения сети ⟩ ≡
typename graph_traits<Graph>::edge_descriptor ed;
bool inserted;

tie(ed, inserted) = add_edge(a, b, g); delay_map[ed] = 1.2;
tie(ed, inserted) = add_edge(a, d, g); delay_map[ed] = 4.5;
tie(ed, inserted) = add_edge(b, d, g); delay_map[ed] = 1.8;
tie(ed, inserted) = add_edge(c, a, g); delay_map[ed] = 2.6;
tie(ed, inserted) = add_edge(c, e, g); delay_map[ed] = 5.2;
tie(ed, inserted) = add_edge(d, c, g); delay_map[ed] = 0.4;
tie(ed, inserted) = add_edge(d, e, g); delay_map[ed] = 3.3;
```

В некоторых случаях более эффективно добавлять или удалять несколько вершин или ребер одновременно, а не по одной. Интерфейс BGL имеет функцию и для этого.

1.2.5. Посетители алгоритмов

Многие из алгоритмов STL имеют параметр объект-функцию, предоставляющий механизм для настройки поведения алгоритма в конкретном приложении. Хорошим примером является функция std::sort(), имеющая параметр compare для задания сравнения:

```
template <typename RandomAccessIterator, typename BinaryPredicate>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          BinaryPredicate compare)
```

Параметр compare является объектом-функцией (иногда используют название «функтор»). Его применение проиллюстрировано в следующем примере.

Рассмотрим код программы для адресной книги. Сортировка массива адресов по фамилии контактного лица может быть осуществлена вызовом функции std::sort() с соответствующим объектом-функцией. Пример такой функции приведен далее:

```
struct compare_last_name {
  bool operator()(const address_info& x, const address_info& y) const {
    return x.last_name < y.last_name;
  }
};
```

Сортировка массива адресов производится вызовом функции `std::sort()` с передачей специализированной функции сравнения.

```
std::vector<address_info> addresses;
// ...
compare_last_name compare;
std::sort(addresses.begin(), addresses.end(), compare);
```

Библиотека BGL предоставляет механизм, подобный функциональным объектам, для специализации алгоритмов на графах. Эти объекты называются *посетителями алгоритмов* (algorithm visitors). Посетитель BGL — объект из нескольких функций. Вместо одного `operator()` для функционального объекта посетитель BGL определяет несколько функций, вызываемых в определенных событийных точках алгоритма (событийные точки изменяются для каждого алгоритма). Несмотря на название, посетители в BGL несколько отличны от паттерна «посетитель», описанного в книге «Паттерны проектирования» [14] авторов Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. («банда четырех»). Посетитель «банды четырех», предоставляет механизм, выполняющий новые операции над объектной структурой без модификации классов. Подобно посетителю «банды четырех» назначение посетителя BGL — обеспечить механизм для расширения. Однако разница заключается в том, что посетитель в BGL расширяет алгоритмы, а не структуры объекта.

В листинге 1.4 приведен код вывода на экран интернет-маршрутизаторов (см. рис. 1.1) путем расширения функции поиска в ширину `breadth_first_search()` с помощью посетителя. Посетитель печатает имя вершины при наступлении события посещения вершины (см. раздел 4.1.1, где описана процедура поиска в ширину). Класс посетителя определяется в соответствии с интерфейсом, описанным в концепции `BFSVisitor`.

Листинг 1.4. Пример класса посетителя

```
template <typename VertexNameMap>
class bfs_name_printer : public default_bfs_visitor {
// Наследовать действия в событийных точках по умолчанию (пустые)
public:
  bfs_name_printer(VertexNameMap n_map) : m_name_map(n_map) { }
  template <typename Vertex, typename Graph>
  void discover_vertex(Vertex u, const Graph& ) const {
    std::cout << get(m_name_map, u) << ' ';
  }
private:
  VertexNameMap m_name_map;
};
```

Далее был создан объект-посетитель типа `bfs_name_printer` и передан функции `breadth_first_search()`. Функция `visitor()`, использованная здесь, является частью процедуры именованного параметра, которая описана в разделе 2.7.

```
bfs_name_printer<VertexNameMap> vis(name_map);
std::cout << "Порядок посещения вершин: ";
breadth_first_search(g, a, visitor(vis));
std::cout << std::endl;
```

Это дает на выходе следующую запись:

```
Порядок посещения вершин: a b d c e
```

Ребра дерева поиска в ширину изображены на рис. 1.2. черными стрелками.

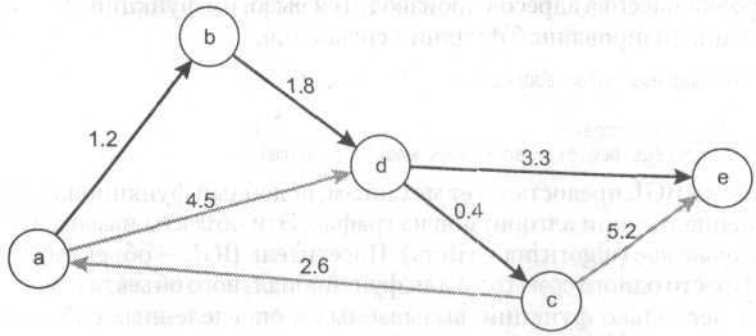


Рис. 1.2. Путь обхода во время поиска в ширину

1.3. Классы и адаптеры графов

Графовые типы, предоставляемые BGL, можно отнести к одной из двух категорий. К первой относятся графовые классы, которые служат для хранения графов в памяти, ко второй — графовые адаптеры (adapters), которые создают измененные представления графов (views) или интерфейс к BGL графу, основанному на другом типе.

1.3.1. Классы графов

Библиотека BGL содержит два первичных графовых класса: список смежности `adjacency_list` и матрица смежности `adjacency_matrix`.

Главный компонент для представления графов — `adjacency_list`. Этот класс обобщает традиционное представление графа в виде списка смежности. Граф представляется набором вершин, каждая из которых хранится со своим набором исходящих ребер. Фактическая реализация набора вершин и ребер может подстраиваться под определенные нужды приложения. Класс `adjacency_list` имеет несколько параметров шаблона: `EdgeList`, `VertexList`, `Directed`, `VertexProperties`, `EdgeProperties` и `GraphProperties`.

- `EdgeList` и `VertexList` предназначены для классов, используемых для хранения списка вершин и списка ребер графа. Эти параметры позволяют найти компромисс между скоростью обхода и скоростью вставки/удаления, а также выбрать уровень потребления памяти. Кроме того, параметр `EdgeList` определяет, могут ли добавляться к графу параллельные вершины.
- `Directed` определяет, является ли граф ориентированным, неориентированным или двунаправленным (bidirectional graph). Двунаправленный граф предоставляет доступ не только к исходящим ребрам, но и к входящим.
- `VertexProperties`, `EdgeProperties` и `GraphProperties` определяют типы свойств, закрепленных за вершинами, ребрами и самим графом соответственно.

Полная документация по классу adjacency_list находится в разделе 14.1.1.

Для представления плотных графов (таких графов, у которых $|E| \approx |V|^2$) больше подходит класс adjacency_matrix. В adjacency_matrix доступ к произвольному ребру (u, v) очень эффективен (это доступ за постоянное время). Этот класс может служить для представления как ориентированных, так и неориентированных графов и обеспечивать механизм для закрепления свойств за вершинами и ребрами. Полная документация по классу adjacency_list находится в разделе 14.1.2.

Стоит заметить, что хотя все приводимые в этой книге примеры относительно невелики (чтобы графическое изображение графа могло поместиться на одной странице), графовые классы BGL являются эффективными по алгоритмам и требуемой памяти. Они могут использоваться для представления графов с миллионами вершин.

1.3.2. Адаптеры графов

Библиотека BGL включает в себя большое количество *адаптеров графов*. Первая группа классов адаптирует любой BGL-граф для реализации нового поведения. Для этого применяются следующие адаптеры графов:

- reverse_graph — адаптер, обращающий направления ребер ориентированного графа таким образом, что входящие вершины ведут себя как исходящие и наоборот;
- filtered_graph — адаптер, создающий представление графа, где два объекта-функции выступают предикатами, контролирующими видимость вершин и ребер в новом графе.

Также BGL обеспечивает поддержку объектов и структур данных, не являющихся графовыми классами BGL. Эта поддержка осуществляется через классы-адаптеры и *перегруженные функции* (overloaded functions). Ниже приведено описание этих интерфейсов.

- edge_list — адаптер, который создает BGL-граф из выхода итератора ребер.
- Пакет программ Stanford GraphBase поддерживается перегруженными функциями из заголовочного файла boost/graph/stanford_graph.hpp. В результате GraphBase-тип Graph* адаптируется к графовому интерфейсу BGL.
- LEDA (популярный объектно-ориентированный пакет) включает структуры данных и алгоритмы для графов. Тип GRAPH<vtype, etype> из LEDA адаптируется к графовому интерфейсу BGL с помощью перегруженных функций из файла boost/graph/leda_graph.hpp.
- Тип std::vector<std::list<int>> из STL преобразуется в граф благодаря перегруженным функциям из файла boost/graph/vector_as_graph.hpp.

Интерфейс BGL более детально описан в справочнике по концепциям в главе 12. Каждый графовый класс реализует некоторые (или все) из этих концепций. Класс adjacency_list можно считать канонической реализацией (моделью) BGL-графа, поскольку он иллюстрирует все базовые идеи и интерфейсы BGL-графов.

1.4. Обобщенные алгоритмы на графах

Алгоритмы на графах из BGL являются обобщенными. Они чрезвычайно гибкие по типам структур данных графов, к которым они могут применяться, и по возможностям специализации для решения широкого диапазона задач. Сначала мы рассмотрим функцию `topological_sort()` для двух различных типов графов, а затем продемонстрируем работу обобщенного алгоритма функции `depth_first_search()` применительно к реализации `topological_sort()`.

1.4.1. Обобщенный алгоритм топологической сортировки

Топологическое упорядочение ориентированного графа — это такое упорядочение его вершин, при котором если в графе присутствует ребро (u, v), то вершина u появляется до вершины v в упорядочении. Шаблон функции `topological_sort()` имеет два аргумента: граф для упорядочения и итератор вывода. Алгоритм записывает вершины в итератор вывода в обратном топологическом порядке.

Топологические упорядочения, например, применяются в задачах планирования. На рис. 1.3 изображен граф, вершинами которого являются подлежащие выполнению задания, а ребра показывают зависимости между заданиями (например, получить деньги в банкомате нужно перед покупкой продуктов).

Рис. 1.3. Граф зависимостей между заданиями

В следующих двух разделах будет показано, как применить алгоритм топологической сортировки к этой задаче. В каждом разделе для иллюстрации обобщенной природы алгоритмов BGL использован свой тип графа.

Использование топологической сортировки с вектором списков

Вначале мы применяем топологическую сортировку к графу, построенному с помощью `std::vector< std::list<int> >`. Код программы топологической сортировки графа приведен в листинге 1.5.

Листинг 1.5. Топологическая сортировка графа. Файл topo-sort1.cpp

```
⟨ topo-sort1.cpp ⟩ ≡
#include <deque>  // для хранения упорядоченных вершин
#include <vector>
#include <list>
#include <iostream>
#include <boost/graph/vector_as_graph.hpp>
#include <boost/graph/topological_sort.hpp>
int main()
{
  using namespace boost;
  ⟨ Создать метки для каждого задания ⟩
  ⟨ Создать граф ⟩
  ⟨ Выполнить топологическую сортировку и вывести результат ⟩
  return EXIT_SUCCESS;
}
```

Вершины графа представлены целыми числами от нуля до шести, поэтому удобно хранить метки вершин в массиве. В листинге 1.6 приведен код для создания соответствующих меток.

Листинг 1.6. Создание меток для заданий

```
⟨ Создать метки для каждого задания ⟩ ≡
const char* tasks[ ] = {
  "получить деньги в банкомате",
  "забрать детей из школы",
  "купить продукты",
  "привести детей на тренировку",
  "забрать детей с тренировки",
  "приготовить ужин",
  "съесть ужин",
};
const int n_tasks = sizeof (tasks) / sizeof (char*);
```

Граф реализован в виде вектора списков. Каждая вершина графа связана индексом вектора. Таким образом, размер вектора определяется количеством вершин в графе. Список по этому же индексу используется для представления ребер, исходящих от данной вершины к другим вершинам графа. Каждое ребро (u, v) добавляется к графу помещением числа для v в список u. Используемая структура данных проиллюстрирована на рис. 1.4.

Благодаря функциям из boost/graph/vector_as_graph.hpp вектор списков соответствует BGL-концепции `VertexListGraph`, а значит, может быть использован в функции `topological_sort()`. Код создания графа приведен в листинге 1.7

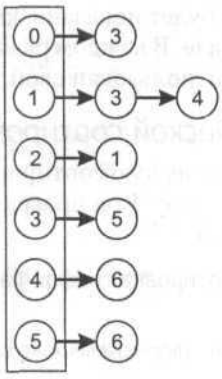


Рис. 1.4. Представление графа зависимостей заданий в виде вектора списков

Листинг 1.7. Создание графа

```
⟨ Создать граф ⟩ ≡
std::vector< std::list<int> > g(n_tasks);
g[0].push_back(3);
g[1].push_back(3);
g[1].push_back(4);
g[2].push_back(1);
g[3].push_back(5);
g[4].push_back(6);
g[5].push_back(6);
```

Чтобы применить `topological_sort()`, необходимо подготовить место для хранения результатов. В BGL-алгоритме топологической сортировки вывод записывается в обратном топологическом порядке (так как это можно реализовать эффективнее). Восстановление топологического порядка требует обращения упорядочения, вычисленного алгоритмом. В следующем примере (листинг 1.8) используется `std::dequeue` в качестве структуры данных для вывода результатов, так как `std::dequeue` может делать вставку в начало за постоянное время, что нужно для получения обратного порядка. Кроме того, вызов `topological_sort()` требует одного из двух: 1) задания отображения свойства для цветовой окраски вершин; 2) задания отображения из вершин в целые числа, чтобы алгоритм мог создать свое собственное отображение свойства для отметки этапов прохождения по графу. Поскольку в нашем примере вершины уже являются целыми числами, мы просто задаем тождественное отображение свойства `identity_property_map` как отображение индексов вершин. Функция `vertex_index_map()` используется для задания именованных параметров (см. раздел 2.7). Код топологической сортировки и вывода результата приведен в листинге 1.8.

Листинг 1.8. Топологическая сортировка и вывод результата

```
⟨ Выполнить топологическую сортировку и вывести результат ⟩ ≡
std::deque<int> topo_order;

topological_sort(g,
  std::front_inserter(topo_order),
  vertex_index_map(identity_property_map()));

int n = 1;
```

```
for (std::deque<int>::iterator i = topo_order.begin();
     i != topo_order.end(); ++i, ++n) {
  std::cout << tasks[*i] << std::endl;
}
```

В результате показан порядок, при котором задания могли бы быть выполнены с учетом заданных зависимостей:

```
получить деньги в банкомате
купить продукты
приготовить ужин
забрать детей из школы
привести детей на тренировку
забрать детей с тренировки
съесть ужин
```

Использование топологической сортировки с классом adjacency_list

Для демонстрации гибкости обобщенного алгоритма `topological_sort()` будем использовать совершенно другой тип графа: шаблон класса `adjacency_list`. Так как функция `topological_sort()` является шаблоном, в нем могут быть использованы графовые структуры произвольного типа. Все, что необходимо, — это чтобы тип удовлетворял концепции, требуемой алгоритмом. Первые два параметра шаблона класса `adjacency_list` определяют конкретную используемую внутреннюю структуру. Первый аргумент `listS` указывает, что `std::list` используется для каждого списка исходящих ребер. Второй аргумент `vecS` свидетельствует о том, что `std::vector` используется как основа списка смежности. Эта версия класса `adjacency_list` аналогична по характеру вектору списков, которым мы пользовались в предыдущем разделе.

```
⟨ Создать объект для списка смежности ⟩ ≡
adjacency_list<listS, vecS, directedS> g(n_tasks);
```

Функция `add_edge()` предоставляет интерфейс для вставки ребер в граф класса `adjacency_list` (и в любые другие графы, поддерживающие концепцию EdgeMutableGraph). Если `std::vector` используется в качестве основы списка смежности, тип описателя вершин для `adjacency_list` будет целым числом. Но использовать целые числа для задания вершин возможно не во всех графовых типах.

```
⟨ Добавить ребра к списку смежности ⟩ ≡
add_edge(0, 3, g);
add_edge(1, 3, g);
add_edge(1, 4, g);
add_edge(2, 1, g);
add_edge(3, 5, g);
add_edge(4, 6, g);
add_edge(5, 6, g);
```

В остальном программа аналогична предыдущему примеру (см. листинг 1.5), за исключением того, что вместо заголовочного файла vector_as_graph.hpp используется файл adjacency_list.hpp. В листинге 1.9 приведен код программы топологиче-ской сортировки графа (две части из предыдущего раздела использованы повторно).

Листинг 1.9. Топологическая сортировка графа. Файл topo-sort2.cpp

```
⟨ topo-sort2.cpp 18 ⟩ ≡
#include <vector>
```

продолжение ⇩

Листинг 1.9 (*продолжение*)

```
#include <deque>
#include <boost/graph/topological_sort.hpp>
#include <boost/graph/adjacency_list.hpp>
int main()
{
  using namespace boost;
  < Создать метки для каждого задания >
  < Создать объект для списка смежности >
  < Добавить ребра к списку смежности >
  < Выполнить топологическую сортировку и вывести результат >
  return EXIT_SUCCESS;
}
```

1.4.2. Обобщенный алгоритм поиска в глубину

Реализация `topological_sort()` в BGL состоит всего из нескольких строк, так как этот алгоритм может использовать функцию `depth_first_search()`, как это обычно и делается в учебниках. Реализация состоит из комбинации `depth_first_search()` и посетителя, устанавливающего порядок, в котором вершины проходят через событийную точку «закончить вершину» поиска в глубину. Объяснение того, почему таким образом действительно вычисляется топологическое упорядочение, дано в разделе 3.3.

В листинге 1.10 описывается алгоритмический посетитель, который записывает вершины при их прохождении через соответствующую событийную точку поиска в глубину. Чтобы сделать этот класс более общим, упорядочение вершин записывается в итераторе вывода, так что пользователь может выбрать метод вывода сам.

Листинг 1.10. Посетитель для записи посещаемых вершин

```
template <typename OutputIterator>
class topo_sort_visitor : public default_dfs_visitor {
// наследовать действия по умолчанию (пустые)
public:
  topo_sort_visitor(OutputIterator iter) : m_iter(iter) { }
  template <typename Vertex, typename Graph>
  void finish_vertex(Vertex u, const Graph&) { *m_iter++ = u; }
private:
  OutputIterator m_iter;
};
```

Итак, `topological_sort()` реализуется запуском алгоритма поиска в глубину `depth_first_search()` с `topo_sort_visitor()` в качестве параметра.

```
template <typename Graph, typename OutputIterator>
void topological_sort(Graph& g, OutputIterator result_iter) {
  topo_sort_visitor<OutputIterator> vis(result_iter);
  depth_first_search(g, visitor(vis));
}
```

Обобщенное программирование в C++

2.1. Введение

Обобщенное программирование (ОП, generic programming, GP) — методология проектирования и реализации программ, которая разделяет структуры данных и алгоритмы через использование абстрактных спецификаций требований. В C++ для обобщенного программирования характерно использование *параметрического полиморфизма шаблонов* (templates) с акцентом на производительность. Методология обобщенного программирования была использована нами при создании библиотеки BGL. Для понимания устройства и структуры BGL читателю необходимо иметь хорошие знания в области обобщенного программирования. В силу своей относительной новизны (по крайней мере, в сообществе пользователей C++) мы решили дать в этой главе введение в обобщенное программирование. Также будут обсуждаться основные приемы ОП в C++, базирующиеся на шаблонах. Эти приемы — не просто набор «фокусов»: совместно они образуют новый подъязык в рамках C++.

Абстрактные спецификации требований в обобщенном программировании подобны прежнему понятию абстрактных типов данных. *Абстрактный тип данных* — это спецификация типа. Она состоит из описания подходящих операций и задает семантику этих операций, зачастую включая пред- и постусловия и аксиомы (или инварианты) [30]. Классический пример абстрактного типа данных — стек с методами записи («push») и извлечения («pop») данных. Есть много способов реализации стека: массив переменного размера, связный список и другие, но детали реализации не важны для пользователя стека до тех пор, пока реализация удовлетворяет спецификации абстрактного типа данных.

В обобщенном программировании понятие абстрактного типа данных расширяется. Вместо определения спецификации для отдельного типа мы описываем семейство типов, которые имеют общий интерфейс и семантическое поведение (semantic behavior). Набор требований, описывающий интерфейс и семантическое поведение, называется *концепцией* (concept). Алгоритмы, сконструированные

в обобщенном стиле, можно применить к любым типам, удовлетворяющим требованиям этого алгоритма. Эта способность использовать различные типы с одной и той же переменной (или параметром функции) называется *полиморфизмом*.

2.1.1. Полиморфизм в объектно-ориентированном программировании

В объектно-ориентированном программировании (ООП) полиморфизм реализуется посредством виртуальных функций и наследования, что называется *полиморфизмом подтипов*. Требования концепции к интерфейсу могут быть записаны как виртуальные функции в абстрактном базовом классе. Предусловия и инварианты соответствуют утверждениям (assertions), когда это возможно. Конкретные классы наследуют из абстрактного базового класса и определяют реализации этих функций. Говорят, что конкретные классы являются подтипами (или производными классами) базового класса. Обобщенные функции пишутся в терминах абстрактного класса, а во время выполнения вызовы функций осуществляются для конкретного типа объекта (в C++ через таблицы виртуальных функций). Любой подтип абстрактного базового класса взаимозаменяем и может быть использован в обобщенной функции.

Классическим примером концепции из математики является *аддитивная абелева группа* — это множество элементов с ассоциативной операцией сложения, имеющее обратный элемент и нуль [45]. Мы можем представить эту концепцию в C++, определив абстрактный базовый класс следующим образом:

```
// Концепция AdditiveAbelianGroup как абстрактный базовый класс
class AdditiveAbelianGroup {
public:
  virtual void add(AdditiveAbelianGroup* y) = 0;
  virtual AdditiveAbelianGroup* inverse() = 0;
  virtual AdditiveAbelianGroup* zero() = 0;
};
```

Используя этот абстрактный базовый класс, мы можем написать универсальную функцию `sum()`:

```
AdditiveAbelianGroup* sum(array<AdditiveAbelianGroup*> v)
{
  AdditiveAbelianGroup* total = v[0]->zero();
  for (int i = 0; i < v.size(); ++i)
    total->add(v[i]);
  return total;
}
```

Функция `sum()` будет работать с любым массивом, тип элемента которого производен от `AdditiveAbelianGroup`. Примерами таких типов могут служить числа и векторы.

```
class Real : public AdditiveAbelianGroup {
  // ...
};
class Vector : public AdditiveAbelianGroup {
  // ...
};
```

2.1.2. Полиморфизм в обобщенном программировании

В обобщенном программировании полиморфизм реализуется с помощью шаблонов классов или функций. Шаблоны обеспечивают *параметрический полиморфизм*. Ниже функция `sum()` записана как шаблон. Наличие базового класса `AdditiveAbelianGroup` уже не нужно, хотя мы оставили его в качестве имени параметра шаблона для удобства и документирования.

```
template <typename AdditiveAbelianGroup>
AdditiveAbelianGroup sum(array<AdditiveAbelianGroup> v)
{
  AdditiveAbelianGroup total = v[0].zero();
  for (int i = 0; i < v.size(); ++i)
    total.add(v[i]);
  return total;
}
```

В C++ концепция выражается в наборе требований к аргументу шаблона, чтобы шаблон класса или функции мог быть успешно скомпилирован и исполнен.

Хотя концепции присутствуют в обобщенном программировании неявно, они очень важны и должны быть тщательно документированы. В настоящее время такая документация обычно появляется в комментариях к коду или в книгах, таких как «Обобщенное программирование и STL» (см. главу «Дополнение к библиографии»). Вернемся к примеру с `AdditiveAbelianGroup`, но уже в качестве концепции AdditiveAbelianGroup.

```
//  концепция AdditiveAbelianGroup (аддитивная абелева группа)
//    правильные выражения:
//       x.add(y)          // добавить
//       y = x.inverse()   // обратить
//       y = x.zero()      // ноль
//    семантика:
//      ...
```

Конкретные типы, удовлетворяющие требованиям AdditiveAbelianGroup, не обязательно должны наследовать от некоторого базового класса. Типы аргументов шаблона подставляются в шаблон функции при инстанцировании (во время компиляции). Для описания отношения между конкретными типами и концепциями, которым они удовлетворяют, используется термин *модель*. Например, `Real` и `Vector` являются моделями концепции AdditiveAbelianGroup.

```
struct Real { //  нет наследования
        // ...
};
struct Vector { //  нет наследования
        // ...
};
```

2.1.3. Сравнение ОП и ООП

До сих пор мы не совсем точно описывали обобщенное программирование как «программирование с шаблонами», а объектно-ориентированное — как «программирование с наследованием». Это может быть несколько неправильно истолковано, так как суть этих методологий только косвенно связана с шаблонами и наследованием.

Если говорить более формально, обобщенное программирование основано на параметрическом полиморфизме, тогда как объектно-ориентированное — на полиморфизме подтипов. В C++ эти полиморфизмы реализованы в виде шаблонов и наследования, но другие языки могут предлагать для этого иные механизмы. Например, расширение сигнатур (signatures extension) в GNU C++ [4] обеспечивает альтернативную форму полиморфизма подтипов. *Мультиметоды* (в таких языках, как CLOS [21]) предлагают семантику, более близкую к параметрическому полиморфизму, но с диспетчеризацией вызовов времени выполнения (в отличие от диспетчеризации во время компиляции шаблонов).

Тем не менее, поскольку мы выбрали стандартный C++, было бы полезно сопоставить ОП и ООП, сравнив наследование (и виртуальные функции) с шаблонами в контексте C++.

Виртуальные функции медленнее шаблонных функций

Вызов виртуальной функции выполняется медленнее, чем вызов шаблонной функции (последний столь же быстр, как и вызов обычной функции), так как вызов виртуальной функции включает в себя лишнее разыменование указателя для нахождения метода в таблице виртуальных функций. Сами по себе эти «накладные расходы» могут быть незначительными, но они способны косвенно повлиять на скомпилированный код: например, не позволить оптимизирующему компилятору встроить функцию в код (как *встраиваемую функцию*, inline-функцию) и применить дальнейшие оптимизации полученного кода.

Конечно, размер издержек на виртуальные функции всецело зависит от объема производимых в функции вычислений. Для компонентов уровня итераторов и контейнеров STL или итераторов графов BGL потери при вызове функции могут оказаться значительными. Производительность на этом уровне сильно зависит от того, являются ли функции вроде `operator++()` встраиваемыми. По этой причине шаблоны являются единственным выбором для реализации эффективных, низкоуровневых, повторно используемых компонентов, подобных тем, что можно найти в STL или BGL.

Диспетчеризация времени выполнения в сравнении с диспетчеризацией во время компиляции

Диспетчеризация времени выполнения виртуальных функций и наследование являются, вне всякого сомнения, одними из лучших свойств объектно-ориентированного программирования. Для некоторых видов компонентов диспетчеризация времени выполнения является безусловным требованием, так как решения могут приниматься на основании информации, доступной только во время выполнения. В этом случае виртуальные функции и наследование необходимы.

Шаблоны не могут предложить диспетчеризацию времени выполнения, но они обеспечивают значительную гибкость во время компиляции. Фактически, если диспетчеризация может быть выполнена во время компиляции, шаблоны обеспечивают большую гибкость, чем наследование, так как не ограничивают типы аргументов шаблона родством с определенным базовым классом (но об этом мы поговорим позже).

Размер кода: виртуальные функции малы, шаблоны велики

Распространенной проблемой в шаблонном программировании является «разбухание» кода из-за неправильного использования шаблонов. Тщательно спроектированные компоненты на основе шаблонов не увеличивают объем кода по сравнению с их аналогами, построенными на наследовании. Основной способ управлять размером кода — отделить функциональность, зависящую от типов шаблона, от функциональности, от них не зависящей. Примером этого может служить реализация `std::list` в библиотеке STL от фирмы SGI.

Проблема бинарного метода

При использовании подтипов (наследования и виртуальных функций) для реализации операций, работающих с двумя или более объектами, возникает серьезная проблема. Эта проблема известна как «проблема бинарного метода» [8]. Классическим примером (приведен далее) этой проблемы является интерфейс класса `Point` (точка, заданная координатами на плоскости), имеющий функцию-метод класса `equal()`. Эта проблема особенно актуальна для BGL, поскольку многие из определяемых в BGL типов (дескрипторы вершин, ребер и итераторы) требуют наличия операции `operator==()` подобно `equal()` для класса `Point`.

Следующий абстрактный базовый класс описывает интерфейс класса `Point`.

```
class Point {
public:
  virtual bool equal(const Point* p) const = 0;
};
```

Используя этот интерфейс, разработчик библиотеки может написать «обобщенную» функцию, которая получает аргументы любого производного от `Point` типа и выводит, являются ли они равными.

```
void print_equal(const Point* a, const Point* b) {
  std::cout << std::boolalpha << a->equal(b) << std::endl;
}
```

Рассмотрим теперь реализацию некоторого «точечного» типа, скажем, класса цветных точек `ColorPoint`. Предположим, что в программе это будет единственный «точечный» класс, который мы будем использовать. В этом случае достаточно определить равенство между двумя объектами `ColorPoint`, не определяя равенства между цветными точками и точками других видов.

```
class ColorPoint : public Point {
public:
  ColorPoint(float x, float y, std::string c) : x(x), y(y), color(c) { }
  virtual bool equal(const ColorPoint* p) const
    { return color == p->color && x == p->x && y == p->y; }
protected:
  float x, y;
  std::string color;
};
```

При попытке использовать этот класс выясняется, что функция `ColorPoint::equal()` не заместила `Point::equal()`. При создании экземпляра `ColorPoint` возникает следующая ошибка:

```
error: object of abstract class type "ColorPoint" is not allowed:
  pure virtual function "Point::equal" has no overrider
ошибка: объект абстрактного типа класса "ColorPoint" не разрешается
создавать:
  чистая виртуальная функция "Point::equal" не замещена
```

Получается, что по правилу типизации тип параметра функции-метода класса в производных классах должен быть либо тот же, либо базовым классом такого же типа, как у параметра в базовом классе. В случае с классом `ColorPoint` параметр для `equal()` должен быть `Point`, а не `ColorPoint`. Однако это изменение приводит к другой проблеме. Внутри функции `equal()` аргумент класса `Point` должен быть приведен к типу `ColorPoint` для выполнения сравнения. Добавление этого приведения означает, что на этапе компиляции остается неизвестным, является ли программа, использующая `ColorPoint`, типобезопасной. Методу `equal()` по ошибке может быть передан объект другого «точечного» класса, вызвав исключение времени выполнения. Ниже описан класс `ColorPoint2`, в котором параметр `equal()` изменен на `Point`, а также добавлено приведение к нужному типу:

```
class ColorPoint2 : public Point {
public:
  ColorPoint2(float x, float y, std::string s) : x(x), y(y), color(s) { }
  virtual bool equal(const Point* p) const {
    const ColorPoint2* cp = dynamic_cast<const ColorPoint2*>(p);
    return color == cp->color && x == cp->x && y == cp->y;
  }
protected:
  float x, y;
  std::string color;
};
```

Теперь предположим, что вместо виртуальных функций мы бы использовали шаблоны функций для выражения полиморфизма. Тогда функция `print_equal()` могла бы быть написана так:

```
template <typename PointType>
void print_equal2(const PointType* a, const PointType* b) {
  std::cout << std::boolalpha << a->equal(b) << std::endl;
}
```

Для использования этой функции класс цветных точек не нуждается в наследовании из класса точек, и проблемы с приведением типов не возникает. При вызове `print_equal2()` с двумя объектами типа `ColorPoint` параметр `PointType` принимает значение `ColorPoint`, что приводит только к вызову `ColorPoint::equal()`. Тем самым поддерживается безопасность типов.

```
ColorPoint* a = new ColorPoint(0.0, 0.0, "blue");
ColorPoint* b = new ColorPoint(0.0, 0.0, "green");
print_equal2(a, b);
```

Так как BGL реализована на шаблонах функций, проблема бинарного метода нам не угрожает. И наоборот, эта проблема возникала бы на каждом шаге, если BGL была основана на виртуальных функциях.

2.2. Обобщенное программирование и STL

Предметная область STL затрагивает основные алгоритмы из информатики (например, структуры массивов и списков, алгоритмы поиска и сортировки). Было

много «фундаментальных» библиотек, которые пытались предоставить всеобъемлющий набор структур данных и алгоритмов. От них STL отличается применением обобщенного программирования (процессом и практикой).

Как объясняют в своей книге [35] Массер и Степанов, процесс ОП в применении к некоторой задаче состоит из следующих основных шагов:

1. Обнаружение полезных и эффективных алгоритмов.
2. Определение их обобщенного представления (то есть параметризация каждого алгоритма таким образом, чтобы требований к обрабатываемым данным было как можно меньше).
3. Описание набора (минимальных) требований, при удовлетворении которых эти алгоритмы могут быть выполнены эффективно.
4. Создание среды разработки (framework), основанной на классификации требований.

Этот процесс отражен в структуре и устройстве компонентов STL.

На практике процесс минимизации и проектирования среды разработки подразумевает такую структуру, где алгоритмы не зависят от конкретных типов данных, с которыми работают. Вернее, алгоритмы пишутся для обобщенных спецификаций, которые были установлены в ходе анализа потребностей этих алгоритмов.

В алгоритмах обычно требуется выполнять обход структуры данных и получать доступ к ее элементам. Если структуры данных предоставляют стандартный интерфейс для обхода и доступа, обобщенные алгоритмы могут быть смело смешаны и согласованы со структурами данных (в терминологии STL — контейнерами).

Главным помощником в задаче разделения алгоритмов и контейнеров в STL является *итератор* (иногда его называют *обобщенным указателем*). Итераторы предоставляют механизм для обхода контейнеров и доступа к элементам. Интерфейс между алгоритмом и контейнером строится на основе требований к итератору, которым должен соответствовать тип итераторов данного контейнера. Обобщенные алгоритмы наиболее гибки, когда они написаны в терминах итераторов и не полагаются на конкретный вид контейнера.

Итераторы подразделяются на несколько категорий (видов), например: InputIterator, ForwardIterator, RandomAccessIterator. Взаимосвязь между контейнерами, алгоритмами и итераторами представлена на рис. 2.1.

Рис. 2.1. Отделение контейнеров и алгоритмов с помощью итераторов

Библиотека STL определяет набор требований для каждого вида итераторов. В требованиях говорится, какие операции (допустимые выражения) определены для данного итератора и каков смысл каждой операции. Для примера рассмотрим некоторые требования к итератору произвольного доступа из STL (он включает в себя требования к ForwardIterator), приведенные в табл. 2.1. Тип `X` означает

итераторный тип, T — указываемый тип, U — тип члена T. Объекты a, b, и r — итераторы, m — член T, n — целое число.

Таблица 2.1. Некоторые требования к итератору произвольного доступа из STL

Выражение	Тип результата	Примечание
a == b	bool	*a == *b
a != b	bool	!(a == b)
a < b	bool	b - a > 0
*a	T&	Разыменование a
a->m	U&	(*a).m
++r	X&	Из r == s следует ++r == ++s
--r	X&	Из r == s следует --r == --s
r += n	X&	Такой же, как n от ++r
a + n	X	{tmp = a; return tmp += n;}
b - a distance(b, a)	Расстояние	(a < b) ? distance(a, b) : -
a[n]	Приводимый к T	*(a + n)

Пример вычислений с накоплением

В качестве конкретного примера рассмотрим алгоритм accumulate(), который последовательно применяет бинарную операцию к начальному значению и каждому элементу контейнера. Произведем суммирование элементов контейнера, используя операцию сложения. Следующий код показывает, как можно реализовать accumulate() на C++. Аргументы first и last — это итераторы, отмечающие начальный и запредельный элементы последовательности. Все аргументы функции параметризованы по типу, таким образом, алгоритм может быть использован с любым контейнером, который моделирует концепцию InputIterator. При прохождении последовательности итератор применяет тот же синтаксис, что и указатели; в частности, operator++() переводит итератор на следующую позицию в последовательности. В табл. 2.1 перечислено несколько других способов продвижения итераторов (главным образом для итератора произвольного доступа). Для доступа к элементу контейнера, на который указывает итератор, можно использовать операцию разыменования operator*() или операцию индексирования operator[]() — для доступа по смещению от текущей позиции итератора.

```
template <typename InputIterator, typename T, typename BinaryOperator>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperator binary_op)
{
  for (; first != last; ++first)
    init = binary_op(init, *first);
  return init;
}
```

Для демонстрации гибкости интерфейса, предоставляемого итератором, мы используем шаблон функции accumulate() с вектором и со связным списком (и то и другое из STL) (листинг 2.1).

Листинг 2.1. Демонстрация интерфейса итератора

```
// использование accumulate() с вектором
std::vector<double> x(10, 1.0);
double sum1;
sum1 = std::accumulate(x.begin(), x.end(), 0.0, std::plus<double>());

// использование accumulate() со связным списком
std::list<double> y;
double sum2;
// копирование значения вектора в список
std::copy(x.begin(), x.end(), std::back_inserter(y));
sum2 = std::accumulate(y.begin(), y.end(), 0.0, std::plus<double>());
assert(sum1 == sum2); // они должны быть равны
```

2.3. Концепции и модели

В предыдущем разделе был приведен пример требований к RandomAccessIterator. Мы также наблюдали, как InputIterator был применен в качестве требования для функции `accumulate()` и как были использованы этой функцией `std::list::iterator` и `std::vector::iterator`. В этом разделе будут определены термины для описания отношений между наборами требований, функций и типов.

В контексте обобщенного программирования термин «концепция» употребляется для описания набора требований, которым должен удовлетворять аргумент шаблона, чтобы функция или класс работали правильно. В тексте использован специальный шрифт для того, чтобы отличать имена концепций.

Примеры определений концепций можно найти в стандарте C++. Многие из них относятся к итераторам. Кроме того, полную документацию по концепциям, использованным в STL, можно найти в книге Мэттью Остерна «Обобщенное программирование и STL» (см. главу «Дополнение к библиографии») и на веб-сайте, посвященном SGI STL http://www.sgi.com/tech/stl/. Эти концепции часто применяются в определениях концепций BGL.

2.3.1. Наборы требований

Требования к концепциям состоят из набора допустимых выражений, ассоциированных типов, инвариантов и гарантии сложности. Тип моделирует концепцию, если он удовлетворяет набору ее требований. Концепция может дополнять требования другой концепции. Это называется *уточнением* (refinement) *концепции*.

- *Допустимые выражения* — это выражения C++, которые должны успешно компилироваться для типов, заданных в выражении, чтобы они считались моделирующими концепцию.
- *Ассоциированные типы* — вспомогательные типы, имеющие некоторое отношение к типу `T`, моделирующему концепцию. Требования концепции обычно содержат утверждения об ассоциированных типах. Например, требования к итератору обычно включают ассоциированный тип `value_type` и необходимость того, чтобы объекты, возвращаемые операцией разыменования итератора, были именно данного типа. В C++ принято использовать *класс свойств* (traits class) для отображения типа `T` в ассоциированные типы концепции.

- *Инварианты* — такие характеристики типов, которые должны быть постоянно верны на этапе выполнения программы. Инварианты часто принимают форму пред- и постусловий. Когда предусловие не выполняется, поведение операции не определено и может приводить к ошибкам. Такая ситуация возникает в библиотеке BGL. Некоторые библиотеки предоставляют отладочные версии, которые используют утверждения или генерируют исключения при нарушении предусловия. Будущие версии BGL, возможно, будут делать так же.
- *Гарантии сложности* — это максимальные пределы того, как долго может происходить выполнение одного из допустимых выражений или сколько различных ресурсов может потребовать это вычисление.

2.3.2. Пример: InputIterator

В этом разделе мы рассмотрим InputIterator как пример концепции. Во-первых, концепция InputIterator является уточнением TrivialIterator, которая, в свою очередь уточняет Assignable и EqualityComparable. Таким образом, InputIterator удовлетворяет всем требованиям TrivialIterator (который удовлетворяет требованиям Assignable и EqualityComparable).

Таким образом тип, моделирующий InputIterator, будет иметь оператор разыменования, он может быть скопирован, присвоен, значения этого типа можно сравнивать с другими объектами-итераторами с помощью операций == и !=.

Концепция InputIterator требует наличия операций *преинкремента* и *постинкремента*. Эти требования обозначены следующими допустимыми выражениями. Объекты `i` и `j` являются экземплярами типа `T`, моделирующего InputIterator.

```
i = j       // присваивание (из Assignable)
T i(j);     // копирование (из Assignable)
i == j      // проверка на равенство (из EqualityComparable)
i != j      // проверка на неравенство (из EqualityComparable)
*i          // разыменование (из TrivialIterator)
++i         // преинкремент
i++         // постинкремент
```

Класс `std::iterator_traits` предоставляет доступ к ассоциированным типам итераторного типа. Тип объекта, на который указывает итератор (назовем его `X`), может быть определен через `value_type` класса свойств. Другими ассоциированными типами являются `reference`, `pointer`, `difference_type` и `iterator_category`. Ассоциированные типы и классы свойств обсуждаются более детально в разделе 2.4. В следующем шаблоне функции мы используем класс `iterator_traits` для получения `value_type` итератора и разыменовываем итератор:

```
template <typename Iterator> void dereference_example(Iterator i)
{
  typename iterator_traits<Iterator>::value_type t;
  t = *i;
}
```

Что касается гарантий сложности, все действия InputIterator должны выполняться за постоянное время. Примерами типов, моделирующих InputIterator, являются `std::list<int>::iterator`, `double*` и `std::istream_iterator<char>`.

Цель определения концепций становится яснее при рассмотрении реализаций обобщенных алгоритмов. Ниже следует реализация функции `std::for_each()`. Внутри функции к объектам-итераторам `first` и `last` применяются ровно четыре операции: сравнение с помощью `operator!=()`, инкремент `operator++()`, разыменовывание `operator*()` и создание копии. Чтобы эта функция успешно откомпилировалась и правильно работала, аргументы-итераторы должны поддерживать как минимум эти четыре операции. Концепция `InputIterator` включает данные операции (и еще некоторые), так что это разумный выбор для лаконичного описания требований к `for_each()`.

```
template <typename InputIterator, typename Function>
Function for_each(InputIterator first, InputIterator last, Function f )
{
  for ( ; first != last; ++first)
    f (*first);
  return f ;
}
```

2.4. Ассоциированные типы и классы свойств

Одним из наиболее важных механизмов, используемых в обобщенном программировании, является *класс свойств* (traits class), который был предложен Натаном Мэйерсом [36]. Механизм класса свойств может показаться немного неестественным при первом знакомстве (из-за синтаксиса), но суть этой идеи проста. Очень важно научиться использовать классы свойств, так как они постоянно применяются в обобщенных библиотеках вроде STL и BGL.

2.4.1. Ассоциированные типы в шаблонах функций

Класс свойств — это просто способ определения информации о типе, без которого мы бы о нем вообще ничего не узнали. Например, рассмотрим обобщенную функцию `sum()`:

```
template <typename Array>
X sum(const Array& v, int n)
{
  X total = 0;
  for (int i = 0; i < n; ++i)
    total += v[i];
  return total;
}
```

С точки зрения этой шаблонной функции о шаблонном типе `Array` известно немногое. Например, тип элементов этого массива не задан. Однако эта информация необходима для объявления локальной переменной `total`, которая должна быть того же типа, что и элементы `Array`. Поэтому имя `X` здесь — только макропеременная, которую нужно заменить на что-то другое, чтобы получить корректную функцию `sum()`.

2.4.2. Определители типов, вложенные в классах

Для того чтобы получить информацию о типе, можно применить операцию разрешения области видимости `::` для доступа к операторам `typedef`, вложенным внутрь класса. Например, класс массива может выглядеть так:

```
class my_array {
public:
  typedef double value_type;  //  тип элемента массива
  double& operator[ ](int i) {
    return m_data[i];
  };
private:
  double* m_data;
};
```

Тип элемента массива можно получить через `array::value_type`. Обобщенная функция `sum()` реализуется с использованием этого приема следующим образом (заметьте, что `X` были заменены на `typename Array::value_type`[1]):

```
template <typename Array>
typename Array::value_type sum(const Array& v, int n)
{
  typename Array::value_type total = 0;
  for (int i = 0; i < n; ++i)
    total += v[i];
  return total;
}
```

В этой функции `sum()` применение вложенного `typedef` целесообразно, пока `Array` является классом, имеющим внутри себя `typedef`. Однако существуют случаи, при которых иметь вложенный `typedef` непрактично или просто невозможно. Допустим, нам захочется использовать обобщенную функцию `sum()` с классом сторонних производителей, который не предоставляет требуемый `typedef`. Или понадобится функция `sum()` со встроенным типом вроде `double*`:

```
int n = 100;
double* x = new double[n];
sum(x, n);
```

В обоих случаях это вполне возможно, так как функциональные требования для использования выполнены: оператор `operator[]()` работает и с `double*`, и с воображаемым классом сторонних производителей. Ограничением для применения является необходимость передачи информации от классов, которые мы хотим использовать, в функцию `sum()`.

2.4.3. Определение класса свойств

Решением этой задачи является класс свойств — шаблон класса, единственое назначение которого — обеспечение отображения из одного типа в другие типы, функции, константы. Механизм языка C++, позволяющий шаблону класса создавать такое отображение, называется *специализацией шаблона* (template specialization). Отображение достигается созданием различных версий класса свойств для определенных параметров-типов. Мы покажем, как это работает, создав класс `array_traits` для использования функции `sum()`.

Шаблонный класс `array_traits` строится на основе типа `Array` и позволяет определить `value_type` (тип элемента) массива. По умолчанию (полностью ша-

[1] Когда тип слева от операции разрешения области видимости :: каким-либо образом зависит от аргумента шаблона, используйте ключевое слово typename перед типом.

блонный) предполагается, что массив — тип со вложенным typedef, такой как my_array:

```
template <typename Array>
struct array_traits {
  typedef typename Array::value_type value_type;
};
```

Теперь возможно создать специализацию шаблона array_traits для обработки ситуации, когда аргумент шаблона является встроенным типом, например double*:

```
template <> struct array_traits<double*> {
  typedef double value_type;
};
```

Классы сторонних производителей, скажем, johns_int_array, могут быть приспособлены к нашим нуждам аналогично:

```
template <> struct array_traits<johns_int_array> {
  typedef int value_type;
};
```

Функция sum(), написанная с использованием класса array_traits, показана ниже. Чтобы осуществить доступ к типу для переменной total, мы извлекаем value_type из array_traits.

```
template <typename Array>
typename array_traits<Array>::value_type sum(const Array& v, int n)
{
  typename array_traits<Array>::value_type total = 0;
  for (int i = 0; i < n; ++i)
    total += v[i];
  return total;
}
```

2.4.4. Частичная специализация

Писать отдельный класс свойств для каждого указательного типа непрактично и нежелательно. Ниже показано, как использовать *частичную специализацию* для организации array_traits для всех типов указателей. Компилятор C++ сопоставит аргумент шаблона, заданный при инстанцировании traits_class, и все определенные специализации для T* и выберет наиболее подходящую специализацию. Частичная специализация для T* подойдет для любого типа-указателя. Для double* компилятором будет выбрана полная специализация (приведенная в разделе 2.4.3.) как более подходящая для конкретного типа-указателя.

```
template <typename T>
struct array_traits<T*> {
  typedef T value_type;
};
```

Частичная специализация может также быть использована для создания версии array_traits для шаблона класса стороннего производителя.

```
template <typename T>
struct array_traits<johns_array<T> > {
  typedef T value_type;
};
```

Наиболее известным применяемым классом из класса свойств является iterator_traits для STL. Библиотека BGL также использует классы свойств, такие как graph_traits и property_traits. Обычно класс свойств применяется с конкретной концепцией или семейством концепций. Класс iterator_traits работает с семейством итераторных концепций, класс graph_traits работает с семейством графовых концепций BGL.

2.4.5. Диспетчеризация тегов

Диспетчеризация тегов (tag dispatching) — это прием, часто работающий «рука об руку» с классами свойств. В нем используется перегрузка функций для диспетчеризации, основанной на свойствах типа. Хорошим примером является реализация функции std::advance() в STL, которая по умолчанию осуществляет инкремент итератора n раз. В зависимости от вида итератора могут применяться различные оптимизации в реализации данной функции. Если итератор обеспечивает произвольный доступ, функция advance() может быть легко и очень эффективно реализована с помощью i += n. Если итератор двунаправленный, то возможен вариант с отрицательным n, поэтому необходимо осуществить декремент итератора n раз. Отношение между внешним полиморфизмом и классами свойств состоит в том, что свойство, которое будет использовано для диспетчеризации (в нашем случае iterator_category), доступно через traits_class.

В следующем примере (листинг 2.2) функция advance() использует класс iterator_traits для определения iterator_category. Затем она выполняет вызов перегруженной функции advance_dispatch(). Подходящая функция advance_dispatch() выбирается компилятором на основе того, к какому типу (теговому классу в листинге ниже) будет отнесен iterator_category. *Тег* — это простой класс, основным назначением которого является передача определенного свойства для использования в теговой диспетчеризации. Принято давать теговым классам имена, оканчивающиеся на _tag. Мы не определяем перегруженную функцию для forward_iterator_tag, так как этот случай обрабатывается перегруженной функцией из input_iterator_tag.

Листинг 2.2. Пример диспетчеризации тегов

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public bidirectional_iterator_tag {};

template <typename InputIterator, typename Distance>
void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag)
    { while (n--) ++i; }

template <typename BidirectionalIterator, typename Distance>
void advance_dispatch(BidirectionalIterator& i, Distance n,
                      bidirectional_iterator_tag)
{
  if (n >= 0)
    while (n--) ++i;
  else
    while (n++) --i;
}
```

```
template <typename RandomAccessIterator, typename Distance>
void advance_dispatch(RandomAccessIterator& i, Distance n,
                      random_access_iterator_tag)
    { i += n; }

template <typename InputIterator, typename Distance>
void advance(InputIterator& i, Distance n)
{
  typedef typename iterator_traits<InputIterator>::iterator_category Cat;
  advance_dispatch(i, n, Cat());
}
```

Класс graph_traits из BGL включает три категории: directed_category, edge_parallel_category и traversal_category. Теги для этих категорий могут быть использованы в диспетчеризации аналогично iterator_category.

2.5. Проверка концепции

Важным аспектом использования обобщенной библиотеки является применение подходящих классов в качестве аргументов шаблона к алгоритмам. То есть необходимо, чтобы применяемые классы моделировали концепции, указанные в требованиях алгоритма. Если используется неподходящий класс, компилятор выдаст сообщения об ошибке, но расшифровать эти сообщения пользователю библиотеки шаблонов может быть достаточно сложно [2, 41]. Компилятор может выдать буквально страницы трудночитаемых сообщений, даже если допущена совсем небольшая ошибка.

Ниже, в листинге 2.3, допущена типичная ошибка, где функция std::sort() применяется к массиву объектов. В этом случае operator<() не реализован для типа foo — это означает, что foo нарушает требования LessThanComparable (которые даны в документации к std::sort()).

Листинг 2.3. Ошибка: невыполнение требований концепции

```
#include <algorithm>
class foo { };
int main(int, char*[ ])
{
  foo array_of_foo[10];
  std::sort(array_of_foo, array_of_foo + 10);
  return 0;
}
```

Сообщение об ошибке, возникающее в результате, нелегко понять, и все, за исключением наиболее опытных программистов C++, попадут в очень непростое положение, пытаясь найти источник возникновения ошибки из данного сообщения. В сообщении не упоминается концепция LessThanComparable, требования которой были нарушены, но показаны многие внутренние функции, вызываемые из std::sort(). Помимо этого, нет указания на строку, в которой произошла ошибка. В нашем случае это вызов std::sort(). Вот как выглядит сообщение об ошибке:

```
stl_heap.h: In function void       __adjust_heap<foo*,int,foo>(foo*,int,int,foo):
stl_heap.h:214:          instantiated from   __make_heap<foo*,foo,ptrdiff_t>(foo*,
        foo*,foo*,ptrdiff_t*)
stl_heap.h:225:          instantiated from make_heap<foo*>(foo*,foo*)
stl_algo.h:1562:          instantiated from   __partial_sort<foo*,foo>(foo*,foo*,
        foo*,foo*)
```

```
stl_algo.h:1574:          instantiated from partial_sort<foo*>(foo*,foo*,foo*)
stl_algo.h:1279:          instantiated from __introsort_loop<foo*,foo,int>(foo*,
        foo*,foo*,int)
stl_algo.h:1320:          instantiated from here
stl_heap.h:115: no match for foo. & < foo &
```

2.5.1. Классы для проверки концепций

Для преодоления этой проблемы мы придумали идиому C++ для принудительной проверки на соответствие концепции, которую мы назвали *проверкой концепции* [39]. Соответствующий код находится в библиотеке проверки концепций BCCL (Boost Concept Checking Library) [6]. Для каждой концепции BCCL предоставляет класс проверки концепции, такой как, например, следующий класс для LessThanComparable. Требуемые допустимые выражения для концепции проверяются в функции-методе класса `constraints()`.

```
template <typename T>
struct LessThanComparableConcept {
  void constraints() {
    (bool)(a < b);
  }
  T a, b;
};
```

По заданным пользователем аргументам шаблона в начале обобщенного алгоритма с использованием `function_requires()` из BCCL инстанцируется класс проверки концепции.

```
#include <boost/concept_check.hpp>
template <typename Iterator>
void safe_sort(Iterator first, Iterator last)
{
  typedef typename std::iterator_traits<Iterator>::value_type T;
  function_requires< LessThanComparableConcept<T> >();
  // другие требования ...
  std::sort(first, last);
}
```

Теперь, если `safe_sort()` будет использован неправильно, сообщение об ошибке (приведенное далее) будет намного понятнее: оно короче, обозначено место ошибки, приведено имя нарушенной концепции, внутренние функции алгоритма не фигурируют в сообщении.

```
boost/concept_check.hpp: In method
  void boost::LessThanComparableConcept<foo>::constraints():
boost/concept_check.hpp:31:   instantiated from
  boost::function_requires<boost::LessThanComparableConcept<foo> >()
sort_eg.cpp:11:  instantiated from safe_sort<foo*>(foo*, foo*)
sort_eg.cpp:21:  instantiated from here
  boost/concept_check.hpp:260: no match for foo & < foo &
```

В библиотеке BGL применяются проверки концепций для обеспечения пользователей более правильными сообщениями об ошибках. Для каждой графовой концепции создан соответствующий класс проверки концепции, определенный в заголовочном файле boost/graph/graph_concepts.hpp. В начале каждого алгоритма в BGL проверяются концепции каждого аргумента. Сообщения об ошибках из graph_concepts.hpp, скорее всего, свидетельствуют о том, что какой-нибудь из

типов одного из аргументов некоторого алгоритма нарушает требования этого алгоритма к концепции.

2.5.2. Прототипы концепций

Дополнительной к проверке концепций задачей является верификация того, что за документированные требования обобщенного алгоритма действительно реализуются алгоритмом. Эту задачу мы называем *покрытием концепции* (concept covering). Обычно авторы библиотек проверяют покрытие вручную, что приводит к ошибкам. Мы же разработали идиому для C++, которая использует проверку типов компилятора C++ [39] для автоматизации этой задачи. Код для покрытия концепций также находится в BCCL. Библиотека BCCL имеет *прототип-класс* для каждой концепции, применяемой в стандартной библиотеке. Прототип-класс представляет собой минимальную реализацию концепции. Для проверки того, покрывает ли концепция алгоритм, создается объект прототип-класса и передается алгоритму.

В следующем примере программа пытается проверить, покрываются ли требования `std::sort()` итератором, моделирующим RandomAccessIterator, который имеет тип значения LessThanComparable.

```
#include <algorithm>
#include <boost/concept_archetype.hpp>
int main()
{
  using namespace boost;
  typedef less_than_comparable_archetype<> T;
  random_access_iterator_archetype<T> ri;
  std::sort(ri, ri);
}
```

На самом деле эта программа не будет успешно скомпилирована, так как те концепции не покрывают требования `std::sort()` к аргументам своего шаблона. Результирующее сообщение об ошибке показывает, что алгоритм также требует, чтобы тип значения был CopyConstructible.

```
null_archetype(const null_archetype<int> &) is private
```

Алгоритму требуется не только конструктор копирования, но и операция присваивания. Эти требования объединяются в концепции Assignable. Следующий код показывает реализацию прототип-класса для Assignable. Прототип-класс снабжен параметром шаблона `Base` для того, чтобы можно было комбинировать прототипы. Для проверки `std::sort()` нам нужно соединить прототип-классы для концепций Assignable и LessThanComparable.

```
template <typename Base = null_archetype<> >
class assignable_archetype : public Base {
  typedef assignable_archetype self ;
public:
  assignable_archetype(const self &) { }
  self & operator=(const self &) { return *this; }
};
```

Библиотека BGL содержит прототип-классы для каждой графовой концепции в заголовочном файле boost/graph/graph_archetypes.hpp. Тестовые программы верификации спецификаций для всех алгоритмов BGL с использованием графовых прототипов находятся в каталоге libs/graph/test/.

2.6. Пространство имен

Как и в других Boost-библиотеках, каждый компонент Boost Graph Library определен в пространстве имен boost для того, чтобы избежать конфликта имен с другими библиотеками и приложениями. В этом разделе мы покажем, как получить доступ к классам и функциям BGL в пространстве имен boost.

2.6.1. Классы

Получить доступ к классам BGL можно несколькими способами. В следующем коде показаны три способа доступа к классу adjacency_list из пространства имен boost.

```
{ // Применить префикс пространства имен
  boost::adjacency_list<> g;
}
{ // Внести класс в текущую область видимости оператором using
  using boost::adjacency_list;
  adjacency_list<> g;
}
{ // Внести все компоненты Boost в текущую область видимости
  using namespace boost;
  adjacency_list<> g;
}
```

Для краткости и ясности представления в примерах кода в этой книге опущен префикс boost::, как если бы оператор using namespace boost; был задан в объемлющей области видимости. Для кода, в котором применяются Boost-библиотеки, мы рекомендуем использовать префикс boost:: в заголовочных файлах и либо указывать его же в исходных файлах, либо применять там операторы using с явным указанием классов. Старайтесь не использовать просто using namespace boost:, так как это может привести к конфликтам имен. Однако внутри функций (то есть в ограниченной области видимости) опасность появления конфликтов мала и using namespace можно использовать.

2.6.2. Поиск Кенига

Операции на графах

Интерфейс BGL состоит из перегруженных функций, определенных для каждого графового типа. Например, функция num_vertices() имеет один аргумент, объект-граф, и возвращает количество вершин. Эта функция перегружается для каждого графового класса BGL. Интересно (и, как мы увидим, к счастью), перегруженные функции могут вызываться без уточнения имени функции пространством имен. Используя процесс, называемый *поиском Кенига*, компилятор C++ рассматривает тип аргумента для перегруженных функций и ищет перегруженные функции *в пространстве имен аргумента*[1].

---

[1] Поиск Кенига назван так в честь его первооткрывателя Эндрю Кенига (Andrew Koenig). Иногда его называют поиском, зависимым от аргумента.

В следующем примере приведена демонстрация поиска Кенига. Рассмотрим случай, когда используются графовые классы из двух различных графовых библиотек. Каждая библиотека имеет свое собственное пространство имен, внутри которого определен графовый класс и функция num_vertices().

```
namespace lib_jack {
  class graph { /* ... */ };
  int num_vertices(const graph&) { /* ... */ }
}

namespace lib_jill {
  class graph { /* ... */ };
  int num_vertices(const graph&) { /* ... */ }
}
```

Предположим, что пользователь хочет применить некоторый обобщенный графовый алгоритм, скажем, boost::pail(), к обоим графовым типам.

```
int main()
{
  lib_jack::graph g1;
  boost::pail(g1);
  lib_jill::graph g2;
  boost::pail(g2);
}
```

Внутри boost::pail() присутствует вызов num_vertices(). В этой ситуации было бы желательно, чтобы при использовании графа из lib_jack вызывалась функция lib_jack::num_vertices(), а если граф из lib_jill — то lib_jill::num_vertices(). Поиск Кенига, как свойство языка C++, позволяет это сделать. Если вызов функции не уточнен указанием пространства имен, компилятор C++ будет осуществлять поиск в пространстве имен аргументов с целью определения правильной функции для вызова.

```
namespace boost {
  template <typename Graph>
  void pail(Graph& g)
  {
    typename graph_traits<Graph>::vertices_size_type
      N = num_vertices(g); // Разрешится в поиске Кенига
    // ...
  }
} // namespace boost
```

Графовые алгоритмы

Графовые алгоритмы из BGL отличаются от операций над графами тем, что они являются шаблонами функций, а не перегруженными функциями. Тем самым поиск Кенига неприменим к графовым алгоритмам BGL и получить к ним доступ можно только посредством префикса пространства имен boost:: или с использованием других методов, описанных в разделе 2.6.1. Например, для вызова алгоритма breadth_first_search() префикс boost:: необходим:

```
boost::breadth_first_search(g, start, visitor(vis));
```

2.7. Именованные параметры функций

Многие алгоритмы BGL имеют длинный список параметров, чтобы обеспечить максимальную гибкость. Однако часто такая гибкость не требуется и было бы достаточным использовать значения по умолчанию для большинства параметров. Например, рассмотрим следующий шаблон функции, имеющий три аргумента.

```
template <typename X, typename Y, typename Z>
void f (X x, Y y, Z z);
```

Пользователь должен иметь возможность передачи любого числа аргументов (и даже ни одного), а не указанные параметры должны получить значения по умолчанию. Также необходима возможность передачи аргумента для параметра y, но не для x или z. Некоторые языки программирования имеют прямую поддержку этих возможностей, называемых *именованными параметрами* (named parameters или keyword parameters). При применении именованных параметров в качестве признака для привязки параметров аргументам используется не порядок параметров (как это принято), а имя, свое у каждого аргумента.

```
// Если бы C++ поддерживал именованные параметры, мы могли бы написать:
int a;
int b;
f (z=b, x=a); // связать b с параметром z, a с параметром x
              // y получает значение аргумента по умолчанию
```

Конечно, C++ не поддерживает именованные параметры, но это свойство может быть реализовано с помощью небольшой хитрости. Библиотека BGL включает класс, называемый `bgl_named_params`, который имитирует именованные параметры, позволяя строить списки параметров[1]. В следующем коде приведен пример вызова функции `bellman_ford_shortest_path()` с использованием техники именованных параметров. Каждый из аргументов передается функции, чье имя показывает, с каким параметром связать аргумент. Заметьте, что именованные параметры разделены не запятой, а точкой. На класс `bgl_named_params` не ссылаются явно — он создается неявно при вызове `weight_map()`, а затем список аргументов расширяется путем вызовов к `distance_map()` и `predecessor_map()`.

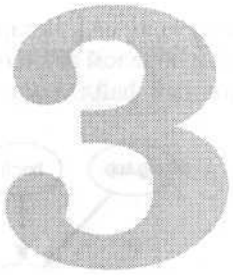
```
bool r = boost::bellman_ford_shortest_paths(g, int(N),
    boost::weight_map(weight).
    distance_map(&distance[0]).
    predecessor_map(&parent[0]));
```

Порядок, в котором следуют аргументы, неважен, пока каждому аргументу соответствует правильная функция. Приведенный ниже вызов функции `bellman_ford_shortest_paths()` эквивалентен приведенному выше.

```
bool r = boost::bellman_ford_shortest_paths(g, int(N),
    boost::predecessor_map(&parent[0]).
    distance_map(&distance[0]).
    weight_map(weight));
```

[1] Это обобщение идиомы, описанной в [41].

Изучаем BGL



Как уже было сказано, *концепции* играют центральную роль в обобщенном программировании. Концепции — это определения интерфейсов, позволяющие алгоритму использовать много различных компонентов. Библиотека алгоритмов на графах (BGL, Boost Graph Library) определяет большой набор концепций, затрагивающий различные аспекты работы с графом, такие как обход графа или изменение его структуры. В данной главе мы познакомим читателя с этими концепциями, а также дадим мотивацию к выбору той или иной концепции BGL.

Из описания процесса обобщенного программирования (см. главу 2 в начале) мы знаем, что концепции выявляются при анализе алгоритмов, используемых для решения проблем в определенных областях. В данной главе мы изучим проблему отслеживания зависимостей файлов в системе сборки. Для каждой подзадачи будут рассмотрены обобщенные, универсальные решения. В результате мы получим обобщенный алгоритм на графе и его приложение к задаче нахождения зависимостей между файлами.

«По пути» мы также рассмотрим более «земные», но необходимые темы, такие как создание графового объекта и наполнение его вершинами и ребрами.

3.1. Зависимости между файлами

Обычным применением для графов является представление зависимостей. Программисты сталкиваются с зависимостями между файлами каждый день при компиляции программ. Информация об этих зависимостях используется такими программами, как make, или средами разработки вроде Visual C++ для определения того, какие файлы должны быть перекомпилированы при генерации новой версии программы (или, в общем случае, для выполнения некоторой цели) после изменений в файлах с исходным кодом.

На рис. 3.1 изображен граф, у которого имеется отдельная вершина для каждого файла с исходным кодом, объектного файла и библиотеки, использованных

в программе killerapp. Ребро этого графа показывает, что целевой объект некоторым образом зависит от другого (например, заголовочный файл включается в исходный файл, а объектный файл компилируется из исходного).

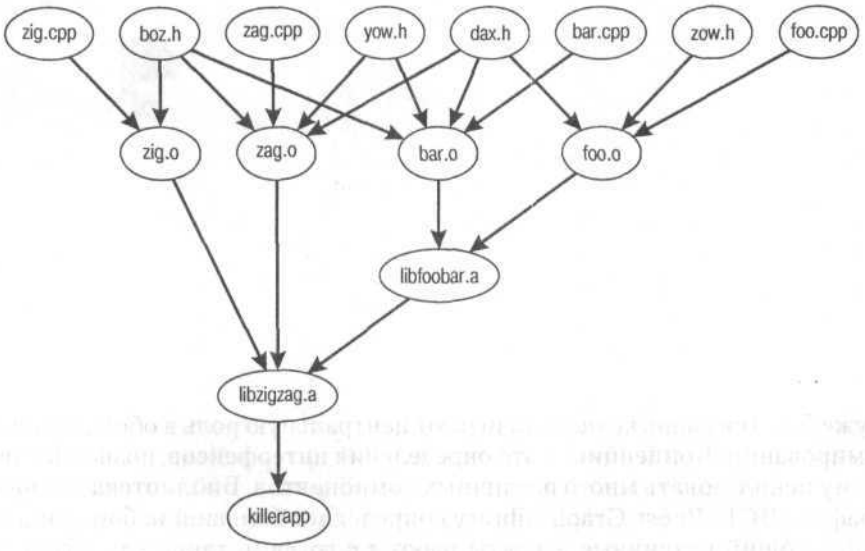


Рис. 3.1. Граф зависимостей между файлами

Ответы на многие вопросы, возникающие при разработке системы сборки, такой как make, могут быть сформулированы в терминах графа зависимостей. Нас может интересовать следующее:

1. Если необходимо собрать все цели, в каком порядке это должно быть сделано?
2. Присутствуют ли в зависимостях циклы? Цикл в зависимостях является ошибкой, и должно быть выдано соответствующее сообщение об ошибке.
3. Сколько шагов нужно сделать для сборки всех целей? Сколько шагов необходимо осуществить для сборки всех целей, если независимые цели собирать параллельно (например, на сети рабочих станций или на многопроцессорном компьютере)?

В следующих разделах эти вопросы ставятся в терминах графов и разрабатываются графовые алгоритмы для их решения. Граф, представленный на рис. 3.1, используется во всех примерах.

3.2. Подготовка графа

Перед тем как мы сможем решать поставленные выше вопросы, нам необходимо иметь представление графа зависимостей между файлами в памяти. То есть нам нужно, используя BGL, построить графовый объект.

3.2.1. Решаем, какой графовый класс использовать

В BGL имеется несколько графовых классов, из которых можно выбирать. Хотя в силу обобщенного характера алгоритмов BGL можно использовать любой определенный пользователем графовый класс, мы ограничимся только классами из BGL. Основными классами в BGL являются `adjacency_list` и `adjacency_matrix`. Первый целесообразно использовать во многих случаях, особенно для представления разреженных графов. Граф файловых зависимостей содержит очень малое число ребер для одной вершины, то есть он является разреженным. Класс `adjacency_matrix` подходит для представления плотных графов, но очень невыгоден для разреженных.

В этой главе используется исключительно `adjacency_list`, хотя многое из представленного можно без изменений использовать с классом `adjacency_matrix`, так как он имеет почти идентичный интерфейс с `adjacency_list`. Здесь мы используем тот же вариант `adjacency_list`, как и в разделе 1.4.1.

```
typedef adjacency_list<
  listS.       // Хранить исходящие ребра каждой вершины в std::list
  vecS.        // Хранить набор вершин в std::vector
  directedS    // Граф файловых зависимостей ориентированный
  > file_dep_graph;
```

3.2.2. Строим граф с помощью итераторов ребер

В разделе 1.2.4 мы показали, как можно использовать функции `add_vertex()` и `add_edge()` для создания графа. Эти функции добавляют вершины и ребра по одному за вызов, но во многих случаях необходимо добавить несколько ребер за один вызов. Для этого класс `adjacency_list` имеет конструктор, который принимает на входе два итератора, определяющих блок ребер. Итератором ребер может быть любой **InputIterator**, который разыменовывается в `std::pair` целых чисел (i, j), представляющих ребро графа. Целые числа i и j представляют вершины, где $0 \le i < |V|$ и $0 \le j < |V|$. Параметры `n` и `m` показывают соответственно количество вершин и ребер в графе. Эти параметры необязательны, но их применение увеличивает скорость создания графа. Параметр свойств графа `p` присоединяется к графовому объекту. Прототип конструктора, использующего итераторы ребер, записывается следующим образом:

```
template <typename EdgeIterator>
adjacency_list(EdgeIterator first, EdgeIterator last,
  vertices_size_type n = 0, edges_size_type m = 0,
  const GraphProperties& p = GraphProperties())
```

В следующем коде показано применение «реберного» конструктора для создания графа. Итератор `std::istream_iterator` служит основой итератора ввода, читающего данные о ребрах из файла. В файле задано число вершин графа, после чего приведены пары чисел, обозначающие ребра. Второй итератор ввода (создаваемый со значениями по умолчанию) является «заглушкой» для конца ввода. Итератор `std::istream_iterator` передается напрямую конструктору графа.

```
std::ifstream file_in("makefile-dependencies.dat");
typedef graph_traits<file_dep_graph>::vertices_size_type size_type;
size_type n_vertices;
```

```
file_in >> n_vertices; // прочесть число вершин
std::istream_iterator<std::pair<size_type, size_type> >
                    input_begin(file_in), input_end;
file_dep_graph g(input_begin, input_end, n_vertices);
```

Поскольку тип значения `std::istream_iterator` — это `std::pair`, итератор ввода нужно определить для `std::pair`.

```
namespace std {
  template <typename T>
  std::istream& operator>>(std::istream& in, std::pair<T,T>& p) {
    in >> p.first >> p.second;
    return in;
  }
}
```

3.3. Порядок компиляции

Первый вопрос касался определения порядка сборки всех целевых объектов. Основной заботой здесь является обеспечение того, чтобы до сборки данной цели были построены все цели, от которых она зависит. Это та же самая задача, которую мы рассматривали в разделе 1.4.1, когда планировали выполнение заданий.

3.3.1. Топологическая сортировка через поиск в глубину

Как уже упоминалось в разделе 1.4.2, топологическое упорядочение может быть получено с использованием алгоритма поиска в глубину (depth-first search). Напомним, что при поиске в глубину посещаются все вершины в графе, начиная с любой вершины, и затем выбирается новое ребро. В следующей вершине выбирается другое ребро для нового шага. Этот процесс продолжается, пока не заходит в тупик (вершина без исходящих ребер до еще не посещенных вершин). Затем алгоритм возвращается на последнюю посещенную вершину, у которой есть другая, еще не пройденная, смежная вершина. После того как будут посещены все вершины, которых можно достичь из выбранной в начале вершины, выбирается еще одна из непройденных вершин, и поиск продолжается. Ребра, которые алгоритм обошел в каждом из этих поисков, образуют *дерево поиска в глубину*, а все такие поиски дают *лес поиска в глубину*. Лес поиска в глубину не уникален для данного графа. Обычно можно найти несколько допустимых лесов для данного графа, так как порядок посещения вершин не фиксирован. Каждое уникальное упорядочение имеет свое дерево поиска в глубину.

Существуют две полезные метрики при поиске в глубину — *порядок посещения* и *порядок окончания обработки вершины*. Представим целочисленный счетчик, значение которого в начале равно нулю. Каждый раз при первом посещении вершины значение счетчика записывается как момент посещения для данной вершины и увеличивается на единицу. Аналогично, когда произведен обход всех вершин, достижимых из данной, значение счетчика записывается как момент окончания данной вершины. Значение счетчика посещения родительской вершины всегда меньше, чем у дочерней. И наоборот, значение счетчика окончания обработки меньше у дочерней вершины, чем у родительской. Поиск в глубину на графе файловых зависимостей показан на рис. 3.2. Ребра деревьев поиска в глубину

отмечены черными стрелками, а у вершин подписаны значения их счетчиков посещения и окончания обработки (через косую черту).

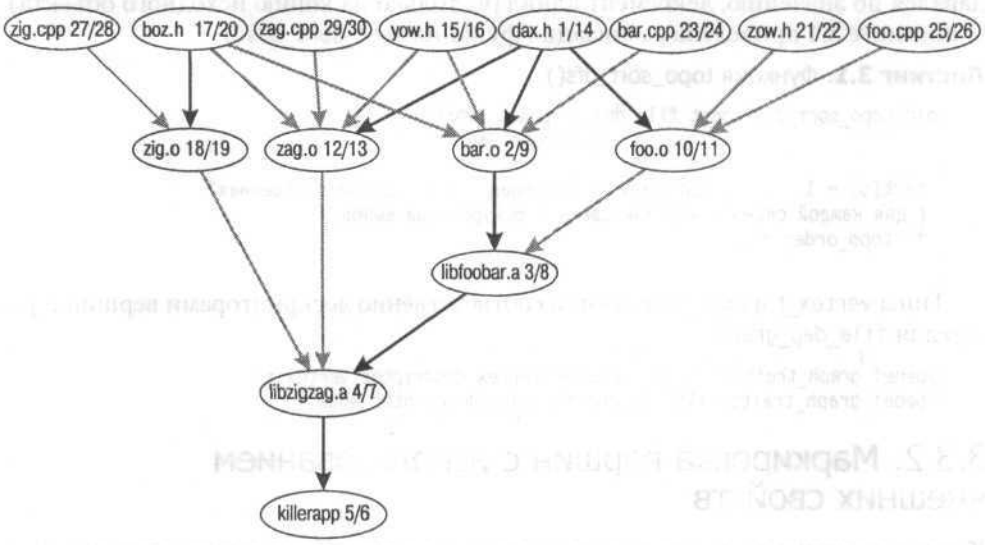


Рис. 3.2. Поиск в глубину на графе файловых зависимостей

Связь между топологическим упорядочением и поиском в глубину можно объяснить, рассмотрев три различных случая в некоторой точке алгоритма поиска в глубину, где проверяется ребро (u, v). В каждом случае значение счетчика в момент окончания обработки v всегда меньше того же значения для u. Таким образом, моменты окончания обработки задают топологическое упорядочение (в обратном порядке).

1. Вершина v еще не посещена. Это означает, что v будет потомком u и значение счетчика окончания обработки вершины будет меньше, чем у u, так как при поиске в глубину потомки u обрабатываются прежде, чем u.
2. Вершина v была посещена ранее в другом дереве поиска в глубину. Значит, значение счетчика окончания обработки v меньше, чем u.
3. Вершина v была посещена ранее в этом же дереве поиска в глубину. Если это случилось, граф содержит цикл и его топологическое упорядочение невозможно. Цикл — это путь из ребер, такой, что первая и последняя вершина пути — одна и та же.

Главной частью поиска в глубину является рекурсивный алгоритм, который вызывает самого себя для каждой смежной вершины. Мы создадим функцию `topo_sort_dfs()`, которая реализует поиск в глубину, модифицированный для вычисления топологического упорядочения. Первая версия этой функции будет необобщенной функцией, решающей задачу «в лоб». В следующих разделах мы внесем изменения, которые позволят создать обобщенный алгоритм.

Параметры `topo_sort_dfs()` включают в себя граф, стартовую вершину, указатель на массив для записи топологического порядка, а также массив для записи посещенных вершин. Указатель `topo_order` устанавливается на конец массива и затем уменьшается для получения прямого топологического порядка из обратного.

Заметьте, что topo_order передается по ссылке, поэтому его декремент модифицирует оригинал объекта при каждом рекурсивном вызове (если бы topo_order передавался по значению, декремент влиял бы только на копию исходного объекта). В листинге 3.1 приведена реализация функции topo_sort_dfs().

Листинг 3.1. Функция topo_sort_dfs()

```
void topo_sort_dfs(const file_dep_graph& g, vertex_t u,
                   vertex_t*& topo_order, int* mark)
{
  mark[u] = 1;  //  1 означает "посещенная", 0 – "еще не посещенная"
  ⟨ Для каждой смежной вершины сделать рекурсивный вызов ⟩
  *--topo_order = u;
}
```

Типы vertex_t и edge_t являются соответственно дескрипторами вершин и ребер для file_dep_graph.

```
typedef graph_traits<file_dep_graph>::vertex_descriptor vertex_t;
typedef graph_traits<file_dep_graph>::edge_descriptor edge_t;
```

3.3.2. Маркировка вершин с использованием внешних свойств

Каждая вершина во время поиска должна быть посещена только один раз. Для записи факта посещения вершины мы можем пометить ее в массиве, содержащем пометки для всех вершин. В общем случае будет использоваться термин *внешнее хранилище свойств* (external property storage) в качестве названия при сохранении свойств вершин или ребер в структуре данных вроде массива или хэш-таблицы, отдельной от графового объекта. Пометки — только одно из свойств, которые можно хранить в структурах данных, внешних по отношению к графу. Значения свойств ищутся по ключу, который может быть легко получен из дескриптора вершины или ребра. В нашем примере мы используем версию adjacency_list, где дескрипторы вершин — целые числа от нуля до num_vertices(g) - 1. Поэтому дескрипторы вершин сами могут быть использованы в качестве индексов к массиву пометок.

3.3.3. Доступ к смежным вершинам

В функции topo_sort_dfs() нам необходимо получить доступ к вершинам, смежным с вершиной *u*. Концепция AdjacencyGraph определяет интерфейс для доступа к смежным вершинам. Функция adjacent_vertices() получает вершину и граф в качестве аргументов и возвращает пару итераторов, чей тип значения — дескриптор вершин. Первый итератор указывает на первую смежную вершину, а второй — за конец последовательности смежных вершин. Смежные вершины выдаются итераторами в произвольном порядке. Оба этих итератора имеют тип adjacency_iterator из класса graph_traits. В справочном разделе по adjacency_list (раздел 14.1.1) написано, что тип adjacency_list моделирует концепцию AdjacencyGraph, а значит, мы можем корректно использовать функцию adjacent_vertices() с нашим графом файловых зависимостей. Код для обхода смежных вершин в topo_sort_dfs() показан ниже:

```
⟨ Для каждой смежной вершины сделать рекурсивный вызов ⟩ ≡
graph_traits<file_dep_graph>::adjacency_iterator vi, vi_end;
```

```
for (tie(vi, vi_end) = adjacent_vertices(u, g); vi != vi_end; ++vi)
  if (mark[*vi] == 0)
    topo_sort_dfs(g, *vi, topo_order, mark);
```

3.3.4. Обход всех вершин

Один из способов гарантировать, что упорядочение будет получено для каждой вершины графа (а не только для вершины, достигаемой из некоторой стартовой вершины), — поместить вызов `topo_sort_dfs()` в цикл по всем вершинам графа. Интерфейс для обхода всех вершин графа определен в концепции VertexListGraph. Функция `vertices()` получает графовый объект и возвращает пару итераторов вершин. Цикл по всем вершинам и создание массива пометок приведены в функции `topo_sort()` (листинг 3.2).

Листинг 3.2. Функция topo_sort()

```
void topo_sort(const file_dep_graph& g, vertex_t* topo_order)
{
  std::vector<int> mark(num_vertices(g), 0);
  graph_traits<file_dep_graph>::vertex_iterator vi, vi_end;
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    if (mark[*vi] == 0)
      topo_sort_dfs(g, *vi, topo_order, &mark[0]);
}
```

Для удобства нам нужно преобразовать вершины-числа в ассоциированные с ними имена целей. Список имен целей (в порядке, соответствующем номеру вершины) хранится в файле, так что возможно прочитать этот файл и сохранить имена в массиве, который затем будет использован для печати имен вершин.

```
std::vector<std::string> name(num_vertices(g));
std::ifstream name_in("makefile-target-names.dat");
graph_traits<file_dep_graph>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
  name_in >> name[*vi];
```

Теперь мы создадим массив `order` для хранения результатов и применим функцию топологической сортировки.

```
std::vector<vertex_t> order(num_vertices(g));
topo_sort(g, &order[0] + num_vertices(g));
for (int i = 0; i < num_vertices(g); ++i)
  std::cout << name[order[i]] << std::endl;
```

В результате получится следующий список:

```
zag.cpp
zig.cpp
foo.cpp
bar.cpp
zow.h
boz.h
zig.o
yow.h
dax.h
zag.o
foo.o
bar.o
```

```
libfoobar.a
libzigzag.a
killerapp
```

3.4. Циклические зависимости

Одним из важных предположений в предыдущем разделе было то, что граф файловых зависимостей не имеет циклов. Как установлено в разделе 3.3.1, граф с циклами не имеет топологического упорядочения. Формально верный Makefile не будет иметь циклов, но наша система сборки должна обнаруживать такие ошибки и сообщать о них.

Поиск в глубину может быть использован и для задачи выявления циклов. Если он будет применен к графу с циклом, одна из ветвей дерева поиска замкнется на себе, то есть, найдется ребро из вершины к одному из ее предков по дереву. Назовем такое ребро черным. Появление черного ребра можно обнаружить, если мы изменим схему отметки вершин. Вместо того чтобы отмечать каждую вершину как посещенную или не посещенную, мы будем использовать трехцветную схему: белый цвет будет обозначать не посещенную вершину, серый — посещенную, но не обработанную (в процессе поиска потомков), и черный — вершину, посещенную вместе со всеми потомками. Трехцветная схема полезна для некоторых графовых алгоритмов, поэтому заголовочный файл boost/graph/properties.hpp определяет следующий перечислимый тип:

```
enum default_color_type { white_color, gray_color, black_color };
```

Цикл в графе идентифицируется по серой смежной вершине — это означает, что ребро замыкается на предке. Код в листинге 3.3 представляет версию поиска в глубину, модифицированную для обнаружения циклов.

Листинг 3.3. Поиск в глубину с обнаружением циклов

```
bool has_cycle_dfs(const file_dep_graph& g, vertex_t u,
                   default_color_type* color)
{
  color[u] = gray_color;
  graph_traits<file_dep_graph>::adjacency_iterator vi, vi_end;
  for (tie(vi, vi_end) = adjacent_vertices(u, g); vi != vi_end; ++vi)
    if (color[*vi] == white_color)
      if (has_cycle_dfs(g, *vi, color))
        return true;  //  если обнаружен цикл, то немедленный возврат
      else if (color[*vi] == gray_color)  //  *vi является предком
        return true;
  color[u] = black_color;
  return false;
}
```

Как и в *топологической сортировке*, в функции has_cycle() (*листинг 3.4*) рекурсивный вызов помещен внутрь цикла по всем вершинам, так что возможно охватить при проверке все деревья поиска в графе.

Листинг 3.4. Функция has_cycle()

```
bool has_cycle(const file_dep_graph& g)
{
  std::vector<default_color_type> color(num_vertices(g), white_color);
```

```
  graph_traits<file_dep_graph>::vertex_iterator vi, vi_end;
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    if (color[*vi] == white_color)
      if (has_cycle_dfs(g, *vi, &color[0]))
  return false;
}
```

3.5. «На пути» к обобщенному поиску в глубину: посетители

На данный момент мы имеем две законченные функции: `topo_sort()` и `has_cycle()`, каждая из которых реализована в виде поиска в глубину, хотя и немного по-разному. Однако их существенная схожесть дает прекрасную возможность для повторного использования кода. Было бы намного лучше, если бы мы имели один алгоритм для поиска в глубину, который являлся обобщением `topo_sort()` и `has_cycle()` и использовал параметры для специализации поиска в глубину в каждой из задач.

Дизайн библиотеки STL подсказывает нам, как можно создать подходящим образом параметризованный алгоритм поиска в глубину. Многие из алгоритмов STL могут быть специализированы параметром в виде определенного пользователем объекта-функции. Нам хотелось бы параметризовать поиск таким же способом, реализовав в `topo_sort()` и `has_cycle()` передачу объекта-функции.

К сожалению, в нашем случае ситуация несколько более сложная, чем в типичном алгоритме STL. В частности, есть несколько различных мест, где должны происходить специализированные действия. Например, функция `topo_sort()` записывает упорядочение в самом конце рекурсивной функции `topo_sort_dfs()`, тогда как `has_cycle()` требует включения действия внутри цикла по смежным вершинам.

Решением этой проблемы является применение объекта-функции с более чем одним членом для обратного вызова. Вместо единственной функции `operator()` мы используем класс с несколькими функциями-методами класса, вызываемыми из различных мест (событийных точек). Такой объект-функция называется *посетителем алгоритма* (algorithm visitor). Посетитель для алгоритма поиска в глубину будет иметь пять функций: `discover_vertex()`, `tree_edge()`, `back_edge()`, `forward_or_cross_edge()` и `finish_vertex()`. Также вместо итерации по смежным вершинам мы будем обходить исходящие ребра. Это позволит передавать дескрипторы ребер функциям посетителя и тем самым предоставлять больше информации посетителю, определенному пользователем. Следующий код функции поиска в глубину, приведенный в листинге 3.5, имеет параметр шаблона для посетителя.

Листинг 3.5. Функция поиска в глубину с посетителем

```
template <typename Visitor>
void dfs_v1(const file_dep_graph& g, vertex_t u,
            default_color_type* color, Visitor vis)
{
  color[u] = gray_color;
  vis.discover_vertex(u, g);
  graph_traits<file_dep_graph>::out_edge_iterator ei, ei_end;
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei) {
    if (color[target(*ei, g)] == white_color) {
      vis.tree_edge(*ei, g);
```

продолжение ⇩

Листинг 3.5 (*продолжение*)

```
        dfs_v1(g, target(*ei, g), color, vis);
      } else if (color[target(*ei, g)] == gray_color)
        vis.back_edge(*ei, g);
      else
        vis.forward_or_cross_edge(*ei, g);
    }
    color[u] = black_color;
    vis.finish_vertex(u, g);
  }

  template <typename Visitor>
  void generic_dfs_v1(const file_dep_graph& g, Visitor vis)
  {
    std::vector<default_color_type> color(num_vertices(g), white_color);
    graph_traits<file_dep_graph>::vertex_iterator vi, vi_end;
    for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi) {
      if (color[*vi] == white_color)
        dfs_v1(g, *vi, &color[0], vis);
    }
  }
```

Пять функций-методов класса посетителя обеспечивают необходимую гибкость, но пользователь, которому нужно, скажем, только одно действие, не должен писать четыре пустые функции-метода класса. Эта проблема может быть легко решена созданием посетителя по умолчанию, из которого могут быть созданы посетители, определяемые пользователем (листинг 3.6).

Листинг 3.6. Структура посетителя со значениями по умолчанию

```
struct default_dfs_visitor {
  // посещение вершины
  template <typename V, typename G>
  void discover_vertex(V, const G&) { }
  // встретилось ребро дерева
  template <typename E, typename G>
  void tree_edge(E, const G&) { }
  // встретилось обратное ребро
  template <typename E, typename G>
  void back_edge(E, const G&) { }
  // встретилось прямое или поперечное ребро
  template <typename E, typename G>
  void forward_or_cross_edge(E, const G&) { }
  // окончание обработки
  template <typename V, typename G>
  void finish_vertex(V, const G&) { }
};
```

Для демонстрации того, что обобщенный алгоритм поиска в глубину может решить наши проблемы, мы переделаем функции `topo_sort()` и `has_cycle()`. Во-первых, нам нужно создать посетителя, записывающего топологическое упорядочение в событийной точке «окончание обработки вершины». Код этого посетителя выглядит так:

```
struct topo_visitor : public default_dfs_visitor {
  topo_visitor(vertex_t*& order) : topo_order(order) { }
  void finish_vertex(vertex_t u, const file_dep_graph&) {
```

```
    *--topo_order = u;
  }
  vertex_t*& topo_order;
};
```

Только две строки содержатся в коде функции `topo_sort()` при ее реализации с использованием обобщенного поиска в глубину. В первой строке создается объект-посетитель, а во второй происходит вызов обобщенного поиска.

```
void topo_sort(const file_dep_graph& g, vertex_t* topo_order)
{
  topo_visitor vis(topo_order);
  generic_dfs_v1(g, vis);
}
```

Для реализации функции `has_cycle()` мы используем посетителя, который записывает наличие цикла, когда встречается *обратное ребро* (back edge).

```
struct cycle_detector : public default_dfs_visitor {
  cycle_detector(bool& cycle) : has_cycle(cycle) { }
  void back_edge(edge t, const file_dep_graph&) {
    has_cycle = true;
  }
  bool& has_cycle;
};
```

Новая функция `has_cycle()` создает объект обнаружения цикла и передает его обобщенному алгоритму поиска в глубину.

```
bool has_cycle(const file_dep_graph& g)
{
  bool has_cycle = false;
  cycle_detector vis(has_cycle);
  generic_dfs_v1(g, vis);
  return has_cycle;
}
```

3.6. Подготовка графа: внутренние свойства

Прежде чем перейти к следующему вопросу по файловым зависимостям, уделим немного внимания другому типу графа. В предыдущих разделах мы использовали массивы для хранения такой информации, как имена вершин. Когда свойства вершин и ребер имеют то же время жизни, что и сам граф, может быть более удобным включить их непосредственно в графовый объект (назовем такие свойства внутренними). При написании собственного графового класса можно включить поля-методы класса для этих свойств в структуру вершины или ребра.

Класс `adjacency_list` имеет параметры шаблона `VertexProperties` и `EdgeProperties`, позволяющие присоединять (приписывать) произвольные свойства к вершинам и ребрам. Эти параметры шаблона предполагают класс `property<Tag, T>` в качестве типа аргумента. Здесь `Tag` — тип, задающий свойство, а `T` устанавливает тип объекта-свойства. Существует некоторое количество предопределенных свойств (см. раздел 15.2.3), таких как `vertex_name_t` и `edge_weight_t`. Например, для присоединения `std::string` к каждой вершине можно использовать следующий тип-свойство:

```
property<vertex_name_t, std::string>
```

Если предопределенных тегов свойств недостаточно, можно создать новый тег. Один из способов — определить тип-перечисление с именем vertex_xxx_t или edge_xxx_t, который содержит перечисление с тем же именем (без _t). Затем можно использовать BOOST_INSTALL_PROPERTY для специализации классов свойств property_kind и property_num[1]. Зададим свойство периода компиляции в виде стоимости (cost), которое мы будем использовать в следующем разделе для вычисления полного времени компиляции.

```
namespace boost {
  enum vertex_compile_cost_t { vertex_compile_cost = 111 }; // уникальный
                                                            номер
  BOOST_INSTALL_PROPERTY(vertex, compile_cost);
}
```

Класс property имеет необязательный третий параметр. Он нужен для вложения нескольких классов property с добавлением множества свойств каждой вершине или ребру. В листинге 3.7 приведен код создания нового typedef для графа, при этом одновременно добавляются свойства вершин и ребер.

Листинг 3.7. Определение типа для графа со свойствами

```
typedef adjacency_list<
  listS,          //  Хранить исходящие ребра в std::list
  listS,          //  Хранить набор вершин в std::list
  directedS,      //  Граф файловых зависимостей ориентированный
  //  свойства вершин
  property<vertex_name_t, std::string,
    property<vertex_compile_cost_t, float,
      property<vertex_distance_t, float,
        property<vertex_color_t, default_color_type> > > >,
  //  свойства ребер
  property<edge_weight_t, float>
  > file_dep_graph2;
```

Мы также изменили второй аргумент шаблона adjacency_list с vecS на listS. Это приводит к важным последствиям. Удаление вершины из графа теперь выполняется за постоянное время (с vecS удаление вершины линейно по числу вершин и ребер). С другой стороны, тип дескриптора вершины больше не является целым числом, поэтому хранить свойства в массивах и использовать вершину как смещение больше нельзя. Однако отдельное хранилище теперь не требуется, так как мы сохраняем свойства вершин в графе.

В разделе 1.2.2 было введено понятие отображения свойств. Напомним, что отображение свойств — это объект, который может быть использован для преобразования ключа (например, вершины) в значение (например, название вершины). Когда были заданы свойства для adjacency_list (как мы только что сделали), отображения для этих свойств могут быть получены с применением интерфейса PropertyGraph. Следующий код показывает пример получения двух отображений свойств: одного — для названий вершин и другого — для времени компиляции. Класс свойств property_map предоставляет тип отображения свойств.

```
typedef property_map<file_dep_graph2, vertex_name_t>::type name_map_t;
typedef property_map<file_dep_graph2, vertex_compile_cost_t>::type
  compile_cost_map_t;
typedef property_map<file_dep_graph2, vertex_distance_t>::type
```

[1] Определение новых тегов свойств было бы значительно проще, если бы больше компиляторов C++ удовлетворяли стандартам.

```
  distance_map_t;
typedef property_map<file_dep_graph2, vertex_color_t>::type color_map_t;
```

Функция `get()` возвращает объект отображения свойств:

```
name_map_t name_map = get(vertex_name, g);
compile_cost_map_t compile_cost_map = get(vertex_compile_cost, g);
distance_map_t distance_map = get(vertex_distance, g);
color_map_t color_map = get(vertex_color, g);
```

Данные по оценке времени компиляции для каждой цели в make-файле будут храниться в отдельном файле. Файл читается при помощи `std::ifstream`, и свойства записываются в граф с использованием отображения свойств `name_map` и `compile_cost_map`. Эти отображения свойств моделируют `LvaluePropertyMap`, а значит, имеют операцию `operator[]()`, которая отображает из дескрипторов вершин в ссылки на соответствующие объекты свойств вершин.

```
std::ifstream name_in("makefile-target-names.dat");
std::ifstream compile_cost_in("target-compile-costs.dat");
graph_traits<file_dep_graph2>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi) {
  name_in >> name_map[*vi];
  compile_cost_in >> compile_cost_map[*vi];
}
```

В следующих разделах мы изменим функции топологической сортировки и поиска в глубину, чтобы использовать интерфейс отображений свойств для доступа к свойствам вершин вместо жесткого прописывания доступа через указатель на массив.

3.7. Время компиляции

Следующие вопросы, на которые нам нужно ответить: «Сколько времени займет компиляция?» и «Сколько времени займет компиляция на параллельном компьютере?» На первый вопрос ответить легко. Мы просто суммируем время компиляции для всех вершин графа. Ради интереса выполним это вычисление с помощью функции `std::accumulate`. Чтобы использовать данную функцию, нужны итераторы, которые при разыменовании дают стоимость компиляции для вершины. Итераторы вершин графа не предоставляют подобной возможности, так как при разыменовании выдают дескрипторы вершин. Вместо них мы применим класс `graph_property_iter_range` (см. раздел 16.8) для генерации подходящих итераторов.

```
graph_property_iter_range<file_dep_graph2,
                          vertex_compile_cost_t>::iterator ci, ci_end;
tie(ci, ci_end) = get_property_iter_range(g, vertex_compile_cost);
std::cout << "полное время последовательной компиляции: "
          << std::accumulate(ci, ci_end, 0.0) << std::endl;
```

Вывод этого участка кода будет таким:

```
полное время последовательной компиляции: 21.3
```

Теперь предположим, что мы имеем параллельный суперкомпьютер с сотнями процессоров. При наличии независимых друг от друга целей они могут компилироваться одновременно на разных процессорах. Сколько времени будет занимать компиляция? Для ответа на этот вопрос необходимо найти критический путь через граф файловых зависимостей. Иначе говоря, нам нужно найти самый длинный путь через граф.

Вклады различных целей во время компиляции файла libfoobar.a изображены на рис. 3.3. Черными стрелками обозначены зависимости файла libfoobar.a. Предположим, что мы уже определили, когда закончится компиляция bar.o и foo.o. Тогда время компиляции для libfoobar.a будет максимумом времен компиляции для bar.o и foo.o плюс время их компоновки в файл библиотеки.

Теперь, когда известно, как вычислить «расстояние» («distance») для каждой вершины, возникает вопрос: в каком порядке мы должны обойти вершины? Понятно, что если есть ребро (u, v), принадлежащее графу, то нужно вычислить расстояние для u перед v, так как вычисление расстояния для v требует знания расстояния до u. Теперь необходимо просмотреть вершины в топологическом порядке.

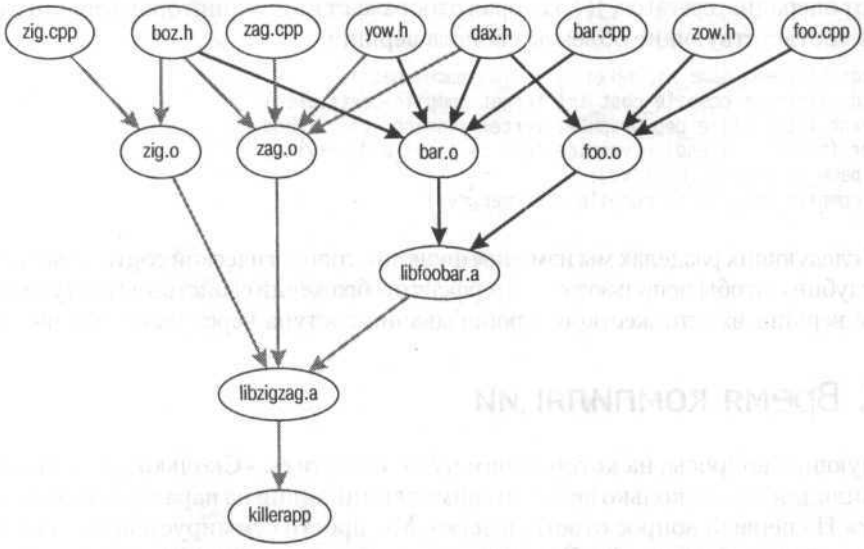


Рис. 3.3. Вклады различных целей во время компиляции libfoobar.a

3.8. Обобщенная топологическая сортировка и поиск в глубину

Так как был заменен тип графа с `file_dep_graph` на `file_dep_graph2`, стало невозможным использовать функцию `topo_sort()`, разработанную в разделе 3.4. Несоответствие касается не только графового типа, но и массива `color`, использованного в `generic_dfs_v1()`, который показывает, что дескрипторы вершин являются целыми числами (это неверно для `file_dep_graph2`). Эти проблемы ведут к созданию даже более общей версии топологической сортировки и лежащего в ее основе поиска в глубину. Параметризация функции `topo_sort()` осуществляется в несколько этапов:

1. Отдельный тип `file_dep_graph` заменяется на параметр шаблона `Graph`. Простая смена параметра шаблона ничего не даст, если нет стандартного интерфейса всех типов графов, которые мы хотим использовать с алгоритмом. Здесь приходят на помощь концепции обхода графов из BGL. Для `topo_sort()` нам нужен тип графа, который моделирует концепции VertexListGraph и IncidenceGraph.

2. Использование vertex_t* для вывода упорядочения является слишком жестким ограничением. Общий способ вывода последовательности элементов предполагает применение итератора вывода, как это происходит в алгоритмах STL. Это дает пользователю намного больше вариантов для хранения результатов.
3. Нам нужно добавить параметр для отображения цветов. Необходимо учесть только самые существенные моменты. Функция topo_sort() должна иметь возможность отображать дескриптор вершины в объект-маркер для данной вершины. Библиотека Boost Property Map Library (см. главу 15) определяет минимальный интерфейс для осуществления этого отображения. В нашем случае используется интерфейс LvaluePropertyMap. Внутреннее отображение color_map, полученное из графа в разделе 3.6, реализует интерфейс LvaluePropertyMap, так же как и массив цветов из раздела 3.3.4. Указатель на массив цветовых маркеров может быть применен как отображение свойств благодаря перегруженным функциям из boost/property_map.hpp, которые адаптируют указатели под интерфейс LvaluePropertyMap.

В листинге 3.8 приведена реализация обобщенного topo_sort(). Далее будут обсуждаться обобщенные topo_visitor и generic_dfs_v2().

Листинг 3.8. Функция topo_sort(), вторая версия

```
template <typename Graph, typename OutputIterator, typename ColorMap>
void topo_sort(const Graph& g, OutputIterator topo_order, ColorMap color)
{
  topo_visitor<OutputIterator> vis(topo_order);
  generic_dfs_v2(g, vis, color);
}
```

Сейчас класс topo_visitor является шаблоном класса (листинг 3.9), который нужно приспособить под итератор вывода. Вместо декремента вставляется инкремент итератора вывода (декремент итератора вывода запрещен). Для получения того же самого обращения, как в первой версии topo_sort(), пользователь может передать внутрь обратный итератор или, например, итератор вставки в начало списка.

Листинг 3.9. Класс topo_visitor

```
template <typename OutputIterator>
struct topo_visitor : public default_dfs_visitor {
   topo_visitor(OutputIterator& order) : topo_order(order) { }
  template <typename Graph>
  void finish_vertex(typename graph_traits<Graph>::vertex_descriptor u,
                     const Graph&)
  { *topo_order++ = u; }
  OutputIterator& topo_order;
};
```

Обобщенный поиск в глубину (листинг 3.10) изменяется через параметризацию типа графа и отображение цветов. Так как неизвестен тип цвета, необходимо запросить тип его значений (с помощью класса свойств property_traits) у ColorMap. Вместо использования констант, таких как white_color, мы применяем цветовые функции, определенные в color_traits.

Листинг 3.10. Обобщенная функция generic_dfs_v2()

```
template <typename Graph, typename Visitor, typename ColorMap>
void generic_dfs_v2(const Graph& g, Visitor vis, ColorMap color)
```

продолжение ⇩

Листинг 3.10 (*продолжение*)

```
{
  typedef color_traits<typename
                       property_traits<ColorMap>::value_type> ColorT;
  typename graph_traits<Graph>::vertex_iterator vi, vi_end;
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    color[*vi] = ColorT::white();
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    if (color[*vi] == ColorT::white())
      dfs_v2(g, *vi, color, vis);
}
```

Логическое содержание `dfs_v1` не требует изменений, однако нужно произвести небольшие модификации из-за параметризации типа графа. Вместо жесткого задания `vertex_t` в качестве дескриптора вершины мы извлекаем соответствующий дескриптор вершины из графового типа, используя класс свойств `graph_traits`. Полностью обобщенная версия функции поиска в глубину приведена в листинге 3.11. Она, в сущности, аналогична `depth_first_visit()` из BGL.

Листинг 3.11. Функция dfs_v2()

```
template <typename Graph, typename ColorMap, typename Visitor>
void dfs_v2(const Graph& g,
  typename graph_traits<Graph>::vertex_descriptor u,
  ColorMap color, Visitor vis)
{
  typedef typename property_traits<ColorMap>::value_type color_type;
  typedef color_traits<color_type> ColorT;
  color[u] = ColorT::gray();
  vis.discover_vertex(u, g);
  typename graph_traits<Graph>::out_edge_iterator ei, ei_end;
  for (tie(ei, ei_end) = out_edges(u, g); ei != ei_end; ++ei)
    if (color[target(*ei, g)] == ColorT::white()) {
      vis.tree_edge(*ei, g);
      dfs_v2(g, target(*ei, g), color, vis);
    } else if (color[target(*ei, g)] == ColorT::gray())
      vis.back_edge(*ei, g);
    else
  vis.forward_or_cross_edge(*ei, g);
  color[u] = ColorT::black();
  vis.finish_vertex(u, g);
}
```

Реальные функции BGL `depth_first_search()` и `topological_sort()` очень похожи на обобщенные функции, которые мы разработали в этом разделе. Детальный пример использования функции `depth_first_search()` рассмотрен в разделе 4.2, а полное описание `depth_first_search()` приведено в разделе 13.2.3. Документация по `topological_sort()` дана в разделе 13.2.5.

3.9. Время параллельной компиляции

Теперь, когда мы имеем обобщенные топологическую сортировку и поиск в глубину, можно решать задачу определения полного времени компиляции на параллельном компьютере. Во-первых, проведем топологическую сортировку, сохранив результаты в векторе `topo_order`. Затем мы передадим обратный итератор в `topo_sort()` для того, чтобы получить прямой топологический порядок (а не обратный).

```
std::vector<vertex_t> topo_order(num_vertices(g));
topo_sort(g, topo_order.rbegin(), color_map);
```

Перед вычислением времен компиляции необходимо подготовить отображение расстояний (его мы будем использовать для хранения результатов вычислений времени компиляции) (листинг 3.12). Для вершин, которые не имеют входящих ребер (будем считать их исходными вершинами), инициализируем их расстояние нулем, так как компиляция этих целей в make-файле может начаться сразу. Всем другим вершинам присваивается бесконечное значение расстояния. Найдем исходные вершины, помечая все вершины с входящими в них ребрами.

Листинг 3.12. Подготовка к вычислению времен компиляции

```
graph_traits<file_dep_graph2>::vertex_iterator i, i_end;
graph_traits<file_dep_graph2>::adjacency_iterator vi, vi_end;

// найти исходные вершины путем пометки всех вершин с входящими ребрами
for (tie(i, i_end) = vertices(g); i != i_end; ++i)
  color_map[*i] = white_color;
for (tie(i, i_end) = vertices(g); i != i_end; ++i)
  for (tie(vi, vi_end) = adjacent_vertices(*i, g); vi != vi_end; ++vi)
    color_map[*vi] = black_color;

// инициализируем расстояния в 0, а для исходных вершин
    присваиваем время компиляции
for (tie(i, i_end) = vertices(g); i != i_end; ++i)
  if (color_map[*i] == white_color)
    distance_map[*i] = compile_cost_map[*i];
  else
    distance_map[*i] = 0;
```

Теперь все готово для вычисления расстояния. Мы проходим через все вершины, записанные в topo_order, и для каждой из них обновляем расстояние (полное время компиляции) до каждой смежной вершины. Это несколько отличается от того, что описано выше. До этого мы говорили о том, что от каждой вершины мы смотрели «вверх»[1] по графу для вычисления расстояния до этой вершины. Сейчас же мы переформулировали вычисления так, что мы проталкиваем расстояния «вниз» по графу. Причина такого изменения состоит в том, что для прохождения «вверх» требуется доступ к входящим ребрам вершины, а этого наш тип графа не предоставляет.

```
std::vector<vertex_t>::iterator ui;
for (ui = topo_order.begin(); ui != topo_order.end(); ++ui) {
  vertex_t u = *ui;
  for (tie(vi, vi_end) = adjacent_vertices(u, g); vi != vi_end; ++vi)
    if (distance_map[*vi] < distance_map[u] + compile_cost_map[*vi])
      distance_map[*vi] = distance_map[u] + compile_cost_map[*vi];
}
```

Максимальное значение расстояния среди всех вершин дает нам искомое полное время параллельной компиляции. Для создания итераторов свойств по расстояниям вершин вновь применяется graph_property_iter_range. Функция std::max_element() используется для нахождения максимума.

```
graph_property_iter_range<file_dep_graph2,
                          vertex_distance_t>::iterator ci, ci_end;
tie(ci, ci_end) = get_property_iter_range(g, vertex_distance);
```

[1] Против направления стрелок. — *Примеч. перев.*

```
std::cout << "полное время параллельной компиляции: "
          << *std::max_element(ci, ci_end) << std::endl;
```

В результате будет выведена такая запись:

```
полное время параллельной компиляции: 11.9
```

Для каждой цели в make-файле, изображенном на рис. 3.4, показаны два числа, означающие время его собственной компиляции и время, за которое он будет скомпилирован при параллельной компиляции. Критический путь обозначен черными стрелками.

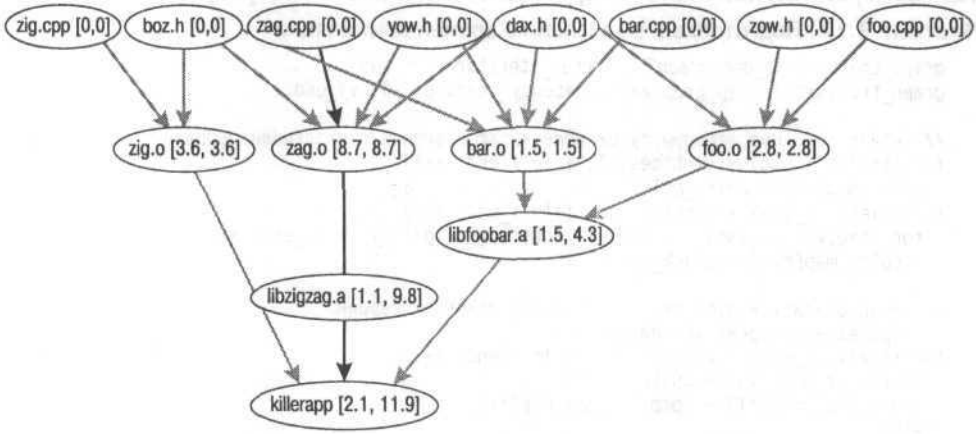


Рис. 3.4. Время компиляции и аккумулированное время компиляции для каждой вершины

3.10. Итоги

В этой главе мы применили BGL для получения ответов на вопросы, которые могут возникнуть при создании системы сборки программного обеспечения: в каком порядке нужно собирать цели make-файла? Нет ли циклических зависимостей? Сколько времени займет компиляция? Ответы на эти вопросы были получены при рассмотрении топологического упорядочения ориентированного графа и его вычислении алгоритмом поиска в глубину.

При реализации решений для представления графа файловых зависимостей мы использовали класс `adjacency_list` из BGL. Была написана простая реализация топологической сортировки и обнаружения циклов. После определения общих мест кода он был преобразован в обобщенную реализацию поиска в глубину. Мы использовали посетителей алгоритма для параметризации поиска в глубину и затем написали конкретных посетителей для реализации топологической сортировки и обнаружения циклов.

Мы также рассмотрели различные варианты класса `adjacency_list`, которые позволили приписать свойства (имя вершины и время компиляции) к вершинам графа. После этого поиск в глубину стал более обобщенным благодаря параметризации графового типа и метода доступа к свойствам. Наконец, были применены топологическая сортировка и поиск в глубину для вычисления времени компиляции всех файлов на параллельном компьютере.

Основные алгоритмы на графах

4.1. Поиск в ширину

Поиск в ширину (breadth-first search, BFS) — один из основных способов получения информации о графе, который можно применить при решении множества различных задач. Библиотека BGL имеет обобщенную реализацию поиска в ширину в виде алгоритма `breadth_first_search()`. Этот функциональный шаблон параметризован, поэтому может быть использован во многих ситуациях. В данном разделе будет описан и применен для вычисления чисел Бэкона алгоритм поиска в ширину.

4.1.1. Определения

Поиск в ширину — обход графа, который посещает все вершины, достижимые из данной исходной вершины. Порядок посещения вершин определяется расстоянием от исходной вершины до каждой вершины графа. Ближние вершины посещаются раньше, чем более удаленные.

Алгоритм поиска в ширину можно представить себе в виде волны, которая распространяется от камня, брошенного в воду. Вершины на гребне одной волны находятся на одинаковом расстоянии от исходной вершины. Поиск в ширину для простого графа показан на рис. 4.1. Порядок посещения вершин имеет последовательность $\{d\}$, $\{f, g\}$, $\{c, h, b, e\}$, $\{a\}$ (вершины сгруппированы по их расстоянию от вершины d).

При посещении вершины v ребро (u, v), которое привело к ее посещению, называется *древесным ребром*. Все вместе древесные ребра образуют *дерево поиска в ширину* с корнем в исходной вершине. Для данного древесного ребра (u, v) вершина u называется *предком*, или *родителем*, для v. Древесные ребра на рис. 4.1. обозначены черными линиями, а все остальные — серыми.

Вершины графа помечены *кратчайшим расстоянием* от исходной вершины d. Кра́тчайшее расстояние $\delta(s, v)$ от некоторой вершины s до вершины v — минимальное количество ребер в любом пути из s в v. *Кратчайший путь* — путь, длина

которого равна $\delta(s, v)$. Понятно, что кратчайший путь может быть не единственным. Главной особенностью поиска в ширину является то, что вершины с более короткими кратчайшими путями посещаются прежде вершин с большими.

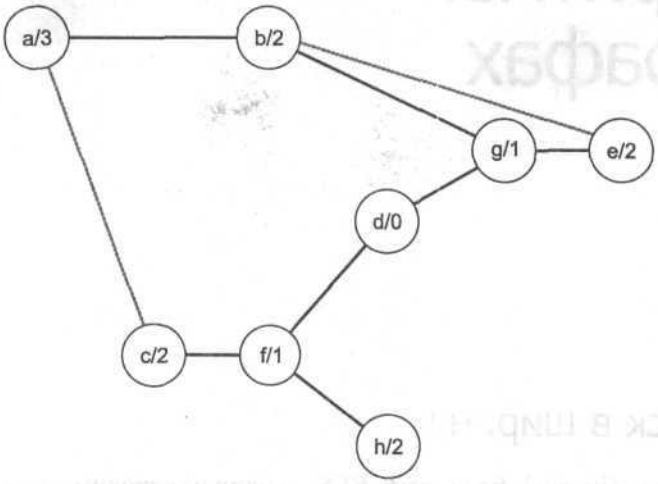


Рис. 4.1. Распространение поиска в ширину по графу

В главе 5 мы займемся вычислениями кратчайших путей, в которых длина пути определяется суммой весов, приписанных ребрам пути, а не просто числом ребер пути.

4.1.2. Шесть степеней Кевина Бэкона

Интересное приложение поиска в ширину возникает в популярной игре «Шесть степеней Кевина Бэкона». Идея игры состоит в том, чтобы связать некоего актера[1] с Кевином Бэконом через цепочку актеров, которые снимались в кино вместе, менее чем за шесть шагов. Например, Теодор Хесбург (заслуженный президент Университета Нотр-Дам в отставке) снимался в фильме «Rudy» с актером Герри Беккером, который снимался в фильме «Sleepers» вместе с Кевином Бэконом. По каким-то неизвестным нам причинам три студента, Майк Джинелли, Крэй Фасс и Брайан Тертл, решили, что Кевин Бэкон является центральной фигурой индустрии развлечений. Математики играют в похожую игру, вычисляя свое *число Эрдеша*, которое является числом работ, написанных в соавторстве, — это число отделяет их от известного Пауля Эрдеша.

Игра «Шесть степеней Кевина Бэкона» на самом деле является задачей на графе. Для того чтобы представить эту задачу в виде графа, нужно назначить вершину каждому актеру и создать ребра между двумя вершинами в том случае, если актеры снимались в одном фильме. Поскольку отношения между актерами симметричны, ребра не имеют направлений и граф получается неориентированным.

Проблема нахождения череды актеров до Кевина Бэкона становится традиционной графовой задачей нахождения пути между двумя вершинами. Поскольку

[1] Под актерами мы будем понимать и актрис.

мы хотим найти путь с длиной менее шести шагов, в идеале было бы неплохо найти кратчайший путь между вершинами. Как было показано в предыдущем разделе, поиск в ширину может быть использован для нахождения кратчайших путей. Подобно числу Эрдеша, мы будем называть длину кратчайшего пути от данного актера к Кевину Бэкону *числом Бэкона*. В следующем примере будет показано, как использовать функцию `breadth_first_search()`, чтобы вычислить числа Бэкона для группы актеров.

Входной файл и создание графа

В данном примере будет использована небольшая часть фильмов и актеров из базы данных по кинофильмам Internet Movie Database[1]. Файл `example/kevin_bacon.txt` содержит список пар актеров, которые снимались в одном и том же фильме. Как показано в следующем фрагменте, каждая строка файла содержит имя актера, название фильма и имя другого актера из этого же фильма. В качестве разделителя используется точка с запятой.

```
Patrick Stewart;Prince of Egypt, The (1998);Steve Martin
```

Для начала прочитаем файл с помощью `std::ifstream` и создадим граф на его основе.

```
std::ifstream datafile("./kevin-bacon.dat");
if (! datafile) {
  std::cerr << "No ./kevin-bacon.dat file" << std::endl;
  return EXIT_FAILURE;
}
```

Для представления графа используется `adjacency_list`, а `undirectedS` служит признаком того, что граф неориентированный. Как и в разделе 3.6, для присваивания вершинам имен актеров, а ребрам — названия фильмов класс `property` используется для добавления этих свойств вершинам и ребрам.

```
typedef adjacency_list<vecS, vecS, undirectedS,
  property<vertex_name_t, std::string>,
  property<edge_name_t, std::string> > Graph;
Graph g;
```

Для доступа к свойствам объекты отображений свойств должны быть получены из графа. Следующий код готовит эти отображения, которые позднее используются дескрипторами вершин и ребер для доступа к связанным с ними именам.

```
typedef property_map<Graph, vertex_name_t>::type actor_name_map_t;
actor_name_map_t actor_name = get(vertex_name, g);
typedef property_map<Graph, edge_name_t>::type movie_name_map_t;
movie_name_map_t connecting_movie = get(edge_name, g);
```

Файл читается построчно и разбирается в список лексем, разделенных точками с запятой. Для создания «виртуального» контейнера лексем используется библиотека лексического разбора Boost Tokenizer Library. Код для разбора файла данными приведен в листинге 4.1.

---

[1] Internet Movie Database используется факультетом информатики Университета Вирджинии для графа к их «Оракулу Бэкона».

Листинг 4.1. Разбор файла с данными

```
for (std::string line; std::getline(datafile,line); ) {
  char delimiters_separator<char> sep(false, "", ":");
  tokenizer<> line_toks(line, sep);
  tokenizer<>::iterator i = line_toks.begin();
  ⟨ Взять имя первого актера и добавить вершину к графу ⟩
  ⟨ Сохранить название фильма в переменной ⟩
  ⟨ Взять имя второго актера и добавить к графу ⟩
  ⟨ Добавить ребро, связывающее двух актеров, к графу ⟩
}
```

Каждая строка ввода соответствует ребру графа, инцидентному двум вершинам, которые определяются именами актеров. Название фильма приписывается к ребру как свойство. Одной из проблем в создании этого графа из такого формата файла является то, что он представляет собой поток ребер. Хотя добавить ребро к файлу несложно, проще добавлять к нему вершины. Вершины появляются только в контексте соединяющих их ребер, причем во входном потоке вершина может появиться несколько раз. Для обеспечения однократного включения вершины в граф используется отображение имен актеров в вершины. По мере добавления вершин к графу последующее появление той же вершины (уже в составе другого ребра) может быть связано с правильной вершиной, уже находящейся в графе. Это легко достигается использованием `std::map`.

```
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef std::map<std::string, Vertex> NameVertexMap;
NameVertexMap actors;
```

Первая лексема каждой строки — это имя актера. Если актера еще нет в отображении, вершина добавляется к графу, свойству вершины `name` присваивается имя актера, а дескриптор вершины записывается в отображение (листинг 4.2). Если актер уже находится в отображении, функция `std::map::insert()` возвращает итератор, указывающий на позицию соответствующей вершины в графе.

Листинг 4.2. Добавление имени первого актера как вершины в графе

```
⟨ Взять имя первого актера и добавить вершину к графу ⟩ ≡
std::string actors_name = *i++;
NameVertexMap::iterator pos;
bool inserted;
Vertex u, v;
tie(pos, inserted) = actors.insert(std::make_pair(actors_name, Vertex()));
if (inserted) {
  u = add_vertex(g);
  actor_name[u] = actors_name;
  pos->second = u;
} else {
  u = pos->second;
}
```

Вторая лексема (название фильма) закрепляется за ребром, соединяющим двух актеров. Однако ребро не может быть создано, пока нет дескрипторов вершин для обоих актеров. Поэтому название фильма сохраняется для последующего использования.

```
⟨ Сохранить название фильма в переменной ⟩ ≡
std::string movie_name = *i++;
```

Третьей лексемой является имя второго актера. Для включения в граф соответствующей вершины используется тот же способ, что и для первого актера.

```
⟨ Взять имя второго актера и добавить к графу ⟩ ≡
tie(pos, inserted) = actors.insert(std::make_pair(*i, Vertex()));
if (inserted) {
  v = add_vertex(g);
  actor_name[v] = *i;
  pos->second = v;
} else v = pos->second;
```

На *последнем шаге добавляем ребро между актерами и записываем название* объединяющего их фильма. Так как для параметра `EdgeList` используется `setS`, то параллельные ребра не попадают в граф.

```
⟨ Добавить ребро, связывающее двух актеров, к графу ⟩ ≡
graph_traits<Graph>::edge_descriptor e;
tie(e, inserted) = add_edge(u, v, g);
if (inserted) connecting_movie[e] = movie_name;
```

Вычисление чисел Бэкона с помощью поиска в ширину

В нашем подходе для вычисления чисел Бэкона с помощью поиска в ширину числа Бэкона вычисляются для всех вершин графа и поэтому необходимо где-нибудь их хранить. Так как мы используем `adjacency_list` с параметром `VertexList=vecS`, *дескрипторы вершин являются целыми числами из диапазона* $[0, |V|)$. Числа Бэкона могут быть записаны в `std::vector`, индексом которого будет дескриптор вершины.

```
std::vector<int> bacon_number(num_vertices(g));
```

Алгоритм `breadth_first_search()` принимает три аргумента: граф, исходную вершину и именованные параметры. Исходная вершина должна быть вершиной, соответствующей Кевину Бэкону, и может быть получена из отображения `actors` (имя-вершина). Число Бэкона для самого Кевина Бэкона равно, разумеется, нулю.

```
Vertex src = actors["Kevin Bacon"];
bacon_number[src] = 0;
```

Для вычисления чисел Бэкона фиксируются расстояния вдоль кратчайших путей. В частности, когда поиск в ширину находит древесную дугу (u, v), расстояние для v может быть вычислено как $d[v] \leftarrow d[u] + 1$. Для вставки этого действия в алгоритм поиска в ширину определим класс-посетитель `bacon_number_recorder`, моделирующий концепцию `BFSVisitor`. Вычисление расстояния будет осуществляться в функции-методе класса `tree_edge()` в событийной точке «встретилось древесное ребро». Класс `bacon_number_recorder` унаследован от `default_bfs_visitor` с реализацией по умолчанию (пустой) остальных функций-методов класса для оставшихся событийных точек (листинг 4. 3). Чтобы сделать посетителя более универсальным, для доступа к расстоянию до вершины используется обобщенный интерфейс `LvaluePropertyMap`.

Листинг 4.3. Класс bacon_number_recorder

```
template <typename DistanceMap>
class bacon_number_recorder : public default_bfs_visitor {
public:
  bacon_number_recorder(DistanceMap dist) : d(dist) { }
```

продолжение ⇩

Листинг 4.3 (*продолжение*)

```
  template <typename Edge, typename Graph>
  void tree_edge(Edge e, const Graph& g) const {
    typename graph_traits<Graph>::vertex_descriptor
      u = source(e, g), v = target(e, g);
    d[v] = d[u] + 1;
  }
private:
  DistanceMap d;
};

// Функция для удобства
template <typename DistanceMap>
bacon_number_recorder<DistanceMap>
record_bacon_number(DistanceMap d)
{
  return bacon_number_recorder<DistanceMap>(d);
}
```

Теперь все готово для вызова `breadth_first_search()`. Аргумент посетителя является именованным параметром, поэтому его необходимо передать, применив функцию `visitor()`. В качестве отображения расстояний здесь используется указатель на начало массива `bacon_number`.

```
breadth_first_search(g, src,
                     visitor(record_bacon_number(&bacon_number[0])));
```

Число Бэкона для каждого актера выводится в цикле по вершинам графа.

```
graph_traits<Graph>::vertex_iterator i, end;
for (tie(i, end) = vertices(g); i != end; ++i) {
  std::cout << actor_name[*i] << " имеет число Бэкона "
            << bacon_number[*i] << std::endl;
}
```

Ниже представлен фрагмент вывода полученной программы.

```
William Shatner имеет число Бэкона 2
Denise Richards имеет число Бэкона 1
Kevin Bacon имеет число Бэкона 0
Patrick Stewart имеет число Бэкона 2
Steve Martin имеет число Бэкона 1
...
```

4.2. Поиск в глубину

Поиск в глубину является основной частью многих графовых алгоритмов. Поиск в глубину использует и алгоритм нахождения сильно связной компоненты (см. раздел 13.5.2), и алгоритм топологической сортировки (см. раздел 13.2.5). Также поиск в глубину полезен и сам по себе. Например, его можно использовать для вычисления достижимости и обнаружения циклов в графе (см. раздел 3.4).

Последняя возможность делает поиск в глубину полезным и для оптимизирующего компилятора, которому необходимо выявлять циклы в графе потока управления программы. В данном разделе описывается, как использовать функции `depth_first_search()` и `depth_first_visit()` при рассмотрении примера обнаружения и определения границ циклов в графе потока управления.

4.2.1. Определения

Поиск в глубину посещает каждую вершину графа ровно один раз. При выборе следующего изучаемого ребра поиск в глубину всегда выбирает ход «глубже» в граф (откуда и происходит его название). То есть поиск в глубину выбирает следующую смежную еще не посещенную вершину, пока не дойдет до вершины, у которой нет смежных не посещенных вершин. Затем алгоритм возвращается к предыдущей вершине и продолжает идти вдоль еще не исследованных ребер, выходящих из этой вершины. После того как поиск в глубину посетил все достижимые вершины (из конкретной исходной вершины), он выбирает одну из еще не посещенных вершин и продолжает работу. При этом процессе создается множество деревьев поиска в глубину, которые вместе образуют лес поиска в глубину. Распространение поиска в глубину на неориентированном графе показано на рис. 4.2. Для каждого ребра обозначен порядок его прохождения.

Подобно алгоритму поиска в ширину, алгоритм поиска в глубину помечает вершины цветами для отслеживания продвижения поиска по графу. Первоначально все вершины имеют белый цвет. Когда алгоритм посещает вершину, он окрашивает ее в серый цвет. После посещения всех потомков вершины она окрашивается в черный цвет.

Поиск в глубину присваивает ребрам графа три категории: *древесное ребро* (tree edge), *обратное ребро* (back edge) и *прямое* или *поперечное* ребро (forward or cross edge). Древесное ребро — это ребро из леса поиска в глубину, который строится (явно или неявно) в процессе обхода графа. Более точно, ребро (u, v) является древесным, если v была впервые посещена после прохождения ребра (u, v). Во время поиска в глубину древесные ребра могут быть идентифицированы по белому цвету вершины v. Вершина u называется *предком*, или *родителем*, вершины v в дереве поиска, если (u, v) является древесным ребром. Обратное ребро соединяет вершину с одним из его потомков в дереве поиска.

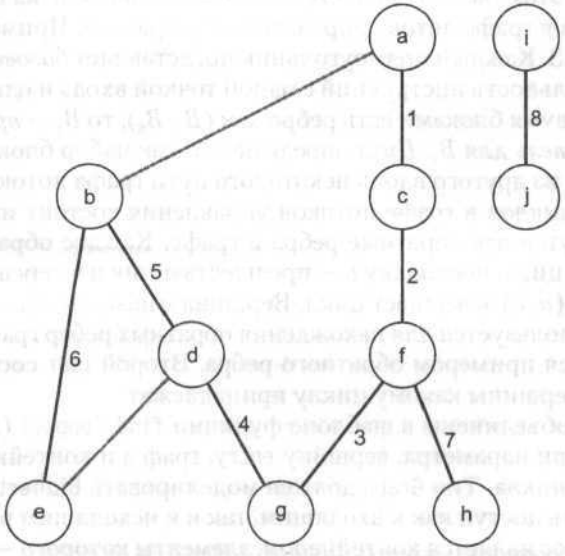


Рис. 4.2. Распространение поиска в глубину по графу

Этот тип ребра выявляется, если конечная вершина v исследуемого ребра (u, v) имеет серый цвет. Петли считаются обратными ребрами. Прямое ребро — ребро (u, v), не принадлежащее дереву поиска, которое соединяет вершину u с потомком v в дереве поиска. К поперечным относятся ребра, не входящие в предыдущие три категории. Если конечная вершина v изучаемого ребра окрашена черным, то это либо прямое ребро, либо поперечное ребро (хотя мы не знаем, какое именно).

Существует много лесов поиска в глубину для данного графа, а значит, много различных (и одинаково правильных) способов классифицировать ребра. Одним из вариантов реализовать поиск в глубину является использование стека. Поиск в глубину во время обработки вершины помещает в стек смежные вершины и извлекает одну вершину в качестве следующей для обработки. Другим (эквивалентным этому) способом является использование рекурсивных функций.

Одним из интересных свойств поиска в глубину является то, что моменты посещения и окончания обработки вершин образуют структуру скобок. Если мы будем писать открывающую скобку при посещении вершины, а закрывающую — при окончании обработки вершины, результатом будет набор правильно вложенных скобок. Здесь мы приведем скобочную структуру графа с рис. 4.2 после применения поиска в глубину. Поиск в глубину является ядром других графовых алгоритмов, включая топологическую сортировку и два алгоритма поиска связных компонент. Он может быть использован и для нахождения циклов (см. раздел 3.4).

```
(a (c (f (g (d (b (e e) b) d) g)(h h) f) c) a) (i (j j) i)
```

4.2.2. Нахождение циклов в графах потоков управления программы

Нашей задачей в этом разделе является использование поиска в глубину для нахождения циклов в графе потока управления программы. Пример графа потоков показан на рис. 4.3. Каждый прямоугольник представляет *базовый блок*, содержащий последовательность инструкций с одной точкой входа и одной точкой выхода. Если между двумя блоками есть ребро как (B_1, B_6), то B_1 — *предшественник* B_6 и B_6 — *последователь* для B_1. *Цикл* определяется как набор блоков, где все блоки достижимы один из другого вдоль некоторого пути графа потоков [32].

Нахождение циклов в графе потоков управления состоит из двух шагов. На первом шаге ищутся все обратные ребра в графе. Каждое обратное ребро (u, v) идентифицирует цикл, поскольку v — предшественник u в дереве поиска в глубину, и добавление (u, v) завершает цикл. Вершина v называется *головой цикла*. Поиск в глубину используется для нахождения обратных ребер графа. Ребро (B_7, B_1) (рис. 4.3) является примером обратного ребра. Второй шаг состоит в определении того, какие вершины какому циклу принадлежат.

Эти два шага объединены в шаблоне функции `find_loops()` (листинг 4.4). Эта функция имеет три параметра: вершину `entry`, граф `g` и контейнер для хранения вершин каждого цикла. Тип `Graph` должен моделировать `BidirectionalGraph`, чтобы можно было иметь доступ как к входящим, так и к исходящим ребрам графа (листинг 4.7). Тип `Loop` является контейнером, элементы которого — наборы вершин.

Обратные ребра из первого шага хранятся в векторе back_edges, а color_map применяется для отслеживания продвижения поиска в глубину.

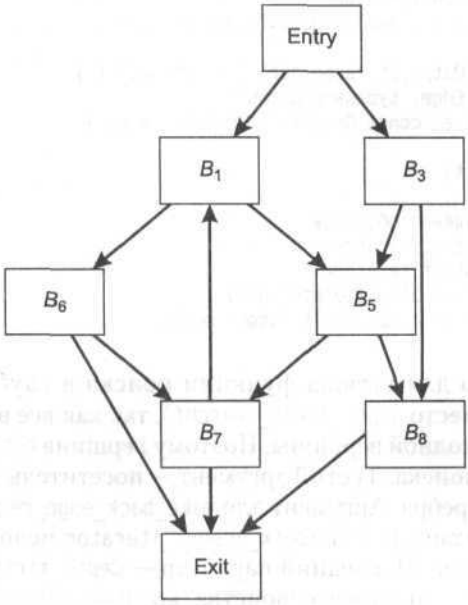


Рис. 4.3. Граф потоков управления программы

Листинг 4.4. Шаблон функции find_loops()

```
⟨ Шаблон функции find_loops ⟩ ≡
template <typename Graph, typename Loops>
void find_loops(
  typename graph_traits<Graph>::vertex_descriptor entry,
  const Graph& g,
  Loops& loops) // Контейнер с набором вершин
{
  function_requires< BidirectionalGraphConcept<Graph> >();
  typedef typename graph_traits<Graph>::edge_descriptor Edge;
  typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
  std::vector<Edge> back_edges;
  std::vector<default_color_type> color_map(num_vertices(g));
  ⟨ Найти все обратные ребра графа ⟩
  ⟨ Найти все вершины каждого цикла ⟩
}
```

На первом шаге создается back_edge_recorder из DFSVisitor, который будет записывать обратные ребра во время поиска в глубину. Чтобы сделать этот класс повторно используемым, механизм хранения обратных ребер не задан, а параметризован как OutputIterator. Класс back_edge_recorder, как обычно, наследует из default_dfs_visitor, с тем чтобы иметь пустые заглушки для функций событийных точек, которые back_edge_recorder не определяет. Реализовать нужно только функцию-метод класса back_edge(). В листинге 4.5 приведены коды шаблона класса back_edge_recorder и функции для порождения объектов этого класса.

Листинг 4.5. Класс регистратора обратных ребер

```
⟨ Класс регистратора обратных ребер ⟩ ≡
template <typename OutputIterator>
class back_edge_recorder : public default_dfs_visitor {
public:
  back_edge_recorder(OutputIterator out) : m_out(out) { }
  template <typename Edge, typename Graph>
  void back_edge(Edge e, const Graph&) { *m_out++ = e; }
private:
  OutputIterator m_out;
};
//  Функция для порождения объектов
template <typename OutputIterator>
back_edge_recorder<OutputIterator>
make_back_edge_recorder(OutputIterator out) {
  return back_edge_recorder<OutputIterator>(out);
}
```

Теперь все готово для вызова функции поиска в глубину. Мы выбираем `depth_first_visit()` вместо `depth_first_search()`, так как все вершины графа потоков достижимы из исходной вершины. Поэтому вершина `entry` передается внутрь как начальная точка поиска. Третий аргумент — посетитель, который будет регистрировать обратные ребра. Аргумент для `make_back_edge_recorder()` должен быть итератором вывода, адаптер `std::back_insert_iterator` используется для записи в вектор обратных ребер. Последний параметр — `depth_first_visit()` — предназначен для отображения цветового свойства, которое поиск в глубину будет использовать при отслеживании продвижения по графу. Это отображение создается итератором вектора `color_map` (см. раздел 15.2.2).

```
⟨ Найти все обратные ребра графа ⟩ ≡
depth_first_visit(g, entry,
  make_back_edge_recorder(std::back_inserter(back_edges)),
  make_iterator_property_map(color_map.begin(), get(vertex_index, g)));
```

На втором шаге процесса определения циклов мы устанавливаем, какие вершины входят в каждый цикл (листинг 4.6). Для каждого обратного ребра, найденного на первом шаге, вызывается функция `compute_loop_extent()`, которая находит все вершины, принадлежащие циклу.

Листинг 4.6. Нахождение всех вершин каждого цикла

```
⟨ Найти все вершины каждого цикла ⟩ ≡
for (std::vector<Edge>::size_type i = 0; i < back_edges.size(); ++i) {
  loops.push_back(typename Loops::value_type());
  compute_loop_extent(back_edges[i], g, loops.back());
}
```

Чтобы вершина v принадлежала циклу обратного ребра (t, h), v должна быть достижима из h и t должна быть достижима из v. Поэтому функция `compute_loop_extent()` (листинг 4.7) состоит из трех шагов: вычисление всех вершин, достижимых из головной вершины, вычисление всех вершин, из которых достижим хвост цикла, и пересечение этих двух множеств вершин.

Листинг 4.7. Вычисление границ цикла

```
⟨ Вычислить границы цикла ⟩ ≡
template <typename Graph, typename Set>
```

```
void compute_loop_extent(
typename graph_traits<Graph>::edge_descriptor back_edge,
const Graph& g, Set& loop_set)
{
  function_requires< BidirectionalGraphConcept<Graph> >();
  typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
  typedef color_traits<default_color_type> Color;

  Vertex loop_head, loop_tail;
  loop_tail = source(back_edge, g);
  loop_head = target(back_edge, g);

  ⟨ Вычислить границы цикла: достижимы из головы ⟩
  ⟨ Вычислить границы цикла: достижимы к хвосту ⟩
  ⟨ Вычислить границы цикла: пересечение наборов достижимости ⟩
}
```

Для вычисления всех вершин, достижимых из головной вершины цикла, снова используется `depth_first_visit()`. В этом случае не требуется определять нового посетителя, поскольку нужно только знать, какие вершины были посещены, а это может быть определено по цвету уже после запуска поиска в глубину. Вершины, окрашенные в черный или серый (но не в белый) цвет, были посещены в ходе поиска в глубину. Цветовые свойства хранятся в векторе `reachable_from_head`. Все вершины, достижимые из блока B_1, показаны на рис. 4.4.

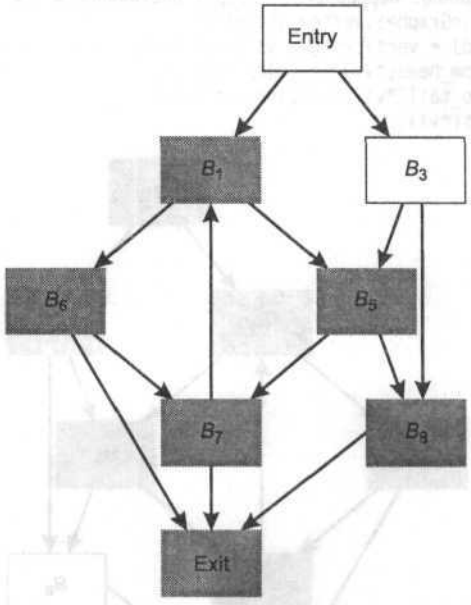


Рис. 4.4. Вершины, доступные из блока B_1

```
⟨ Вычислить границы цикла: достижимы из головы ⟩ ≡
std::vector<default_color_type>
    reachable_from_head(num_vertices(g), Color::white());
  depth_first_visit(g, loop_head, default_dfs_visitor(),
    make_iterator_property_map(reachable_from_head.begin(),
      get(vertex_index, g)));
```

На втором шаге необходимо вычислить все вершины, из которых достижим блок B_7. Для этого выполним поиск в глубину «против течения». То есть вместо обхода по исходящим ребрам каждой вершины обходятся входящие ребра. Функция `depth_first_visit()` из BGL применяет функцию `out_edges()` для доступа к следующим вершинам, однако она может быть применена к нашей ситуации, если мы используем адаптер `reverse_graph`. Этот адаптер берет `BidirectionalGraph` и выдает представление графа, в котором исходящие и входящие ребра поменялись местами.

Следующий код показывает, как это может быть сделано. Все вершины, из которых доступен блок B_7, отображены на рис. 4.5.

```
⟨ Вычислить границы цикла: достижимы к хвосту ⟩ ≡
std::vector<default_color_type> reachable_to_tail(num_vertices(g));
reverse_graph<Graph> reverse_g(g);
depth_first_visit(reverse_g, loop_tail, default_dfs_visitor(),
  make_iterator_property_map(reachable_to_tail.begin(),
    get(vertex_index, g)));
```

На последнем шаге вычисления вершин, принадлежащих циклу, осуществляется пересечение двух полученных ранее множеств достижимости. Вершина вносится в `loop_set` в случае, если она достижима из головной вершины и если «хвост» достижим из нее.

```
⟨ Вычислить границы цикла: пересечение наборов достижимости ⟩ ≡
typename graph_traits<Graph>::vertex_iterator vi, vi_end;
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    if (reachable_from_head[*vi] != Color::white()
      && reachable_to_tail[*vi] != Color::white())
      loop_set.insert(*vi);
```

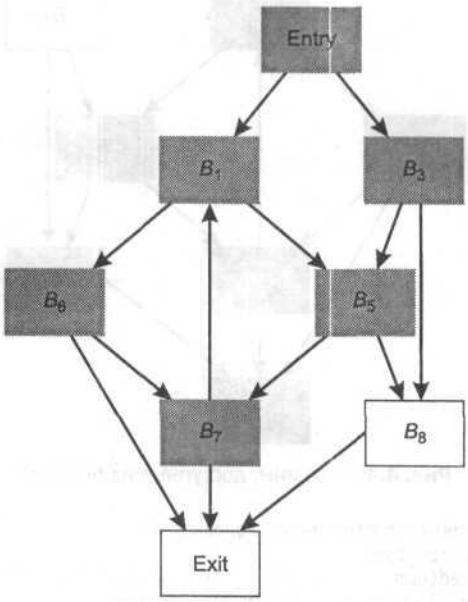


Рис. 4.5. Вершины, из которых доступен блок B_7

Задачи нахождения кратчайших путей

В этой главе мы решаем некоторые задачи маршрутизации пакетов с использованием алгоритмов нахождения кратчайших путей. В первом разделе проблема кратчайшего пути объясняется в общем виде и напоминаются некоторые определения. Второй раздел является кратким введением в пакетную маршрутизацию. Третий и четвертый разделы описывают два наиболее часто используемых протокола маршрутизации пакетов и представляют реализацию их основных алгоритмов с помощью BGL.

5.1. Определения

Путем называется последовательность вершин $<v_0, v_1, ..., v_k>$ в графе $G = (V, E)$, такая, что каждое из ребер (v_i, v_{i+1}) находится в наборе E (каждая вершина соединена со следующей вершиной последовательности). В задаче нахождения кратчайшего пути каждое ребро (u, v) имеет вес $w(u, v)$. *Вес пути* (или *длина пути*) определяется как сумма весов каждого ребра пути:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

Вес $\delta(u, v)$ *кратчайшего пути* из вершины u в вершину v — это минимум весов всех возможных путей из u в v. Если пути из u в v не существует, $\delta(u, v) = \infty$:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \to v\} \\ \infty \end{cases}.$$

Кратчайшим путем называется любой путь, вес которого равен весу кратчайшего пути. Пример кратчайшего пути показан на рис. 5.1.

Задача кратчайшего пути между двумя вершинами состоит в нахождении кратчайшего пути, соединяющего две данные вершины. *Задача кратчайших путей из*

одной вершины заключается в нахождении кратчайшего пути от заданной вершины до всех других вершин графа. Набор кратчайших путей, исходящих из одной вершины, называется *деревом кратчайших путей*. *Задача кратчайшего пути между всеми парами вершин* графа состоит в нахождении кратчайших путей из каждой вершины графа в каждую другую.

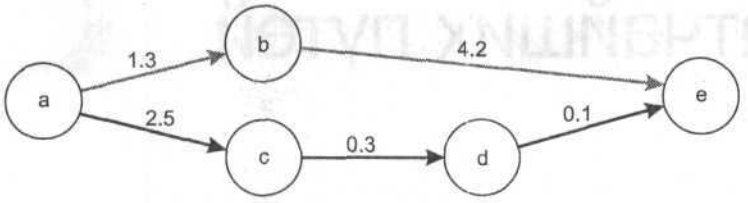


Рис. 5.1. Кратчайший путь от вершины *a* до *e* обозначен черными стрелками

Как оказалось, для решения задачи кратчайшего пути между двумя вершинами не существует алгоритмов, которые были бы асимптотически более быстрыми, чем алгоритмы для решения задачи кратчайших путей из одной вершины. Библиотека BGL включает в себя два классических метода для решения задачи кратчайшего пути из одной вершины: алгоритм Дейкстры и алгоритм Беллмана–Форда. Также BGL включает алгоритм Джонсона для нахождения кратчайших путей между всеми парами вершин.

Алгоритмы нахождения кратчайших путей широко применяются во многих областях. Одним из важных современных применений является маршрутизация пакетов в Интернете. Протоколы, которые управляют информационными пакетами при передаче через Интернет, используют алгоритмы кратчайших путей для сокращения времени прохождения пакета до его назначения.

5.2. Маршрутизация в Интернете

Когда компьютер отправляет сообщение другому компьютеру с использованием *интернет-протокола* (IP), содержимое сообщения укладывается в пакет. Каждый пакет помимо данных сообщения (полезная составляющая) включает метаданные, такие так адреса источника и приемника, длина данных, порядковый номер и т. п. Если сообщение большое, данные разбиваются на меньшие части, каждая из которых пакуется отдельно. Индивидуальные части снабжены порядковыми номерами, так что исходное сообщение может быть собрано на принимающей стороне.

Если адрес назначения для пакета находится вне локальной сети, пакет отправляется с исходной машины на *интернет-маршрутизатор* (internet router). Маршрутизатор направляет получаемые пакеты другим маршрутизаторам, исходя из таблицы маршрутизации, которая создается на основании протоколов маршрутизации. Путешествуя с одного маршрутизатора на следующий (то есть совершая *переход*, hop), пакеты прибывают на место назначения. Если сеть перегружена, некоторые пакеты могут быть потеряны. Для надежной доставки применяются протоколы более высокого уровня, например *протокол управления передачей* (Transmission Control Protocol, TCP). Он использует квитирование (установление связи) между отправителем и получателем, чтобы потерянные пакеты передавались

вновь. Программа системы Unix traceroute (или ее Windows-аналог tracert) может быть использована для прослеживания пути от вашего компьютера к другим узлам Интернета.

Конечная цель процесса маршрутизации состоит в доставке пакетов до пункта назначения как можно быстрее. На продолжительность доставки пакета влияет множество факторов, например количество переходов (hops), задержки внутри маршрутизаторов, задержки передачи между маршрутизаторами, пропускная способность сети и т. д. Протокол маршрутизации должен выбирать наилучшие пути между маршрутизаторами — эта информация хранится в таблице маршрутизации.

Задача маршрутизации может быть смоделирована в виде графа, в котором каждая вершина является маршрутизатором, а каждое прямое соединение между маршрутизаторами — ребром. Ребру приписаны такие данные, как задержка и пропускная способность. Граф для простой сети маршрутизаторов представлен на рис. 5.2. У соединений обозначены средние задержки передачи. Теперь задача маршрутизации сведена к задаче кратчайших путей.

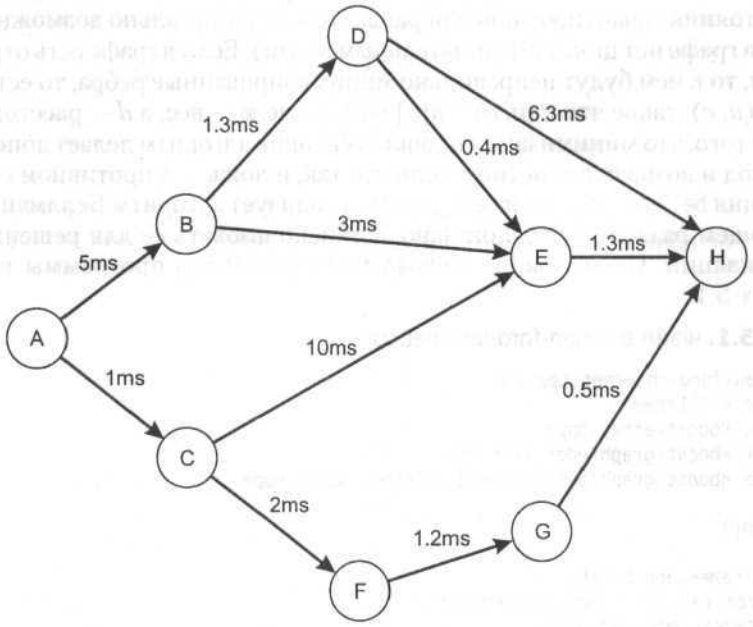


Рис. 5.2. Интернет-маршрутизаторы, соединенные друг с другом

5.3. Алгоритм Беллмана–Форда и маршрутизация с помощью вектора расстояний

Некоторые из первых интернет-протоколов маршрутизации, например *протокол маршрутной информации* (Routing Information Protocol, RIP) [19], использовали вектор расстояний. Основная идея RIP — поддерживать на каждом маршрутизаторе оценку расстояния до всех других маршрутизаторов и периодически сравнивать

эти записи со своими соседями. Если маршрутизатор узнает о более коротком пути до некоторого пункта назначения от одного из своих соседей, он обновляет запись о расстоянии до этого пункта и изменяет свою таблицу маршрутизации так, чтобы пакеты в этот пункт шли через данного соседа. По прошествии некоторого времени оцененные расстояния, хранимые таким распределенным способом, гарантированно сходятся к истинным расстояниям, давая маршрутизаторам точную информацию о наилучшем пути.

Алгоритм протокола RIP является распределенной формой алгоритма Беллмана–Форда для нахождения кратчайшего пути из одной вершины [5, 13]. Основной шаг алгоритма Беллмана–Форда называется *релаксацией ребра* и соответствует сравнению своих записей с соседями. Операция релаксации, примененная к ребру (u, v), выполняет следующее обновление:

$$d[v] = \min(w(u,v) + d[u], d[v]).$$

Алгоритм Беллмана-Форда содержит цикл по всем ребрам графа, в котором релаксация применяется к каждому ребру. Алгоритм повторяет цикл $|V|$ раз, после чего расстояния гарантированно сокращаются до минимально возможных (если, конечно, в графе нет цикла с отрицательным весом). Если в графе есть отрицательный цикл, то в нем будут неправильно минимизированные ребра, то есть найдется ребро (u, v), такое что $w(u, v) + d[u] < d[v]$, где w — вес, а d — расстояние. Для проверки того, что минимизация прошла успешно, алгоритм делает дополнительный проход и возвращает истину, если это так, и ложь — в противном случае.

Функция `bellman_ford_shortest_paths()` реализует алгоритм Беллмана–Форда. В следующем разделе будет показано, как использовать ее для решения задачи маршрутизации. Схема файла bellman-ford-internet.cpp программы приведена в листинге 5.1.

Листинг 5.1. Файл bellman-ford-internet.cpp

```
⟨ bellman-ford-internet.cpp ⟩ ≡
#include <iostream>
#include <boost/array.hpp>
#include <boost/graph/edge_list.hpp>
#include <boost/graph/bellman_ford_shortest_paths.hpp>

int main()
{
  using namespace boost;
  ⟨ Подготовить сеть маршрутизаторов ⟩
  ⟨ Присвоить веса ребрам ⟩
  ⟨ Создать хранилище для свойств вершин ⟩
  ⟨ Вызвать алгоритм Беллмана-Форда ⟩
  ⟨ Вывести расстояния и родителей ⟩
  return EXIT_SUCCESS;
}
```

Первый аргумент функции `bellman_ford_shortest_paths()` является графовым объектом. Тип этого графового объекта должен моделировать концепцию EdgeListGraph. Многие из графовых классов BGL моделируют EdgeListGraph и, следовательно, могут быть использованы с этим алгоритмом. Здесь мы применим шаблон класса `edge_list`, который является адаптером. Он позволяет представлять множество значений итератора в виде графа. Тип значений итератора должен быть

`std::pair` — пара дескрипторов вершин. Дескрипторы вершин могут быть практически *любого* типа, хотя мы используем целые числа для удобства применения их в качестве индексов в массивах.

В нашем случае ребра хранятся в `boost::array`, где каждое ребро есть `std::pair`. Каждой вершине приписан идентификационный номер, заданный с помощью перечисления. Параметры шаблона для `edge_list` содержат тип итератора, тип значения итератора и тип разности итераторов[1]. Код по подготовке сети маршрутизаторов приведен в листинге 5.2.

Листинг 5.2. Подготовка сети маршрутизаторов

```
⟨ Подготовить сеть маршрутизаторов ⟩ ≡
// Идентификационные номера для маршрутизаторов (вершин)
enum { A, B, C, D, E, F, G, H, n_vertices };
const int n_edges = 11;
typedef std::pair<int, int> Edge;

// Список соединений между маршрутизаторами, сохраненный в массиве
array<Edge, n_edges> edges = { { Edge(A, B), Edge(A, C),
  Edge(B, D), Edge(B, E), Edge(C, E), Edge(C, F), Edge(D, H),
  Edge(D, E), Edge(E, H), Edge(F, G), Edge(G, H) } };

// Указание типа графа и определение графового объекта
typedef edge_list<array<Edge, n_edges>::iterator> Graph;
Graph g(edges.begin(), edges.end());
```

Для передачи веса ребра (задержек передачи) в алгоритм нужно определить отображение весового свойства ребра, моделирующее концепцию ReadablePropertyMap. По умолчанию параметр `weight_map()` является внутренним отображением свойства веса ребра графа, которое может быть получено с помощью `get(edge_weight, g)`. Поскольку класс `edge_list` не поддерживает определенные пользователем внутренние отображения свойств, веса ребер должны храниться не в нем и аргумент отображения свойства должен быть передан в функцию явно. Класс `edge_list` предоставляет отображение свойства ребро-индекс, так что индексы ребер могут быть использованы как смещения в массиве свойств ребер. В нашем случае там хранятся задержки передачи. Приведенный ниже код создает массив значений задержек передачи.

```
⟨ Присвоить веса ребрам ⟩ ≡
// Значения задержек передачи для каждого ребра
array<float, n_edges> delay =
  { { 5.0, 1.0, 1.3, 3.0, 10.0, 2.0, 6.3, 0.4, 1.3, 1.2, 0.5 } };
```

Массив `delay` обеспечивает хранение весов ребер, но не предоставляет интерфейс отображения свойств, требуемый алгоритмом для отображения дескрипторов ребер в вес. Необходимый интерфейс отображения свойств предоставляется классом-адаптером `iterator_property_map` из библиотеки Boost Property Map Library. Этот класс преобразует итератор (такой, как итератор для массива значений задержек) в LvaluePropertyMap. Удобным средством для создания адаптера является вспомогательная функция `make_iterator_property_map()`. Первый аргумент —

[1] Для компиляторов с работающей версией std::iterator_traits шаблонные параметры edge_list типа значения и типа разности не нужны, так как имеются правильные параметры по умолчанию.

итератор, второй — отображение из ребер в индексы ребер, и третий аргумент — объект типа значений итератора, который нужен только для вывода типа. Вызов функции make_iterator_property_map() может быть таким:

```
⟨ Создать отображение свойства для задержек ⟩ ≡
make_iterator_property_map(delay.begin(), get(edge_index, g), delay[0])
```

В качестве значения, возвращаемого функцией, служит созданный объект-адаптер, который затем передается функции Беллмана–Форда. Функция get() извлекает отображение edge_index из объекта-графа и является частью интерфейса PropertyGraph.

Вершинам графа приписано несколько свойств. Как и на рис. 5.2, вершины обозначены буквами (это их имена). Метки расстояний требуются для записи длин кратчайших путей. Наконец, отображение предков parent используется для записи дерева кратчайших путей. Для каждой вершины в графе отображение предков записывает родителя данной вершины в соответствии с деревом кратчайших путей, то есть каждое ребро (parent[u], u) является ребром в дереве кратчайших путей.

Класс edge_list не обеспечивает методы для задания свойств вершин (вершины — только целые числа). Свойства хранятся в отдельных массивах с номером вершины в качестве индекса (листинг 5.3). Начальным значением расстояний является бесконечность, и родителем каждой вершины первоначально устанавливается она сама.

Листинг 5.3. Создание хранилища свойств вершин

```
⟨ Создать хранилище для свойств вершин ⟩ ≡
// Определение хранилищ для некоторых "внешних" свойств вершин
char name[ ] = "ABCDEFGH";
array<int, n_vertices> parent;
for (int i = 0; i < n_vertices; ++i)
  parent[i] = i;
array<float, n_vertices> distance;
distance.assign(std::numeric_limits<float>::max());
// Обозначим A как исходную вершину
distance[A] = 0;
```

Поскольку описатели вершин графа edge_list являются целыми числами, указатели на массивы свойств подходят в качестве отображений свойств, так как Boost Property Map Library включает специализации для встроенных типов указателей (см. раздел 15.2.1).

Ниже показан вызов bellman_ford_shortest_paths(). Расстояния кратчайших путей записаны в векторе расстояний, а родитель каждой вершины (в соответствии с деревом кратчайших путей) записан в вектор родителей.

```
⟨ Вызвать алгоритм Беллмана-Форда ⟩ ≡
bool r = bellman_ford_shortest_paths(g, int(n_vertices),
  weight_map( ⟨ Создать отображение свойства для задержек ⟩ ).
  distance_map(&distance[0]).
  predecessor_map(&parent[0]));
```

Программа завершается выводом предков и расстояний для каждого маршрутизатора в сети или выводит предупреждение пользователю об отрицательном цикле в сети.

```
⟨ Вывести расстояния и родителей ⟩ ≡
if (r)
  for (int i = 0; i < n_vertices; ++i)
    std::cout << name[i] << ": " << distance[i]
              << " " << name[parent[i]] << std::endl;
else
  std::cout << "отрицательный цикл" << std::endl;
```

Для нашего примера программа выдаст следующее:

```
A: 0      A
B: 5      A
C: 1      A
D: 6.3    B
E: 6.7    D
F: 3      C
G: 4.2    F
H: 4.7    G
```

Таким образом, работая в обратную сторону через предков, мы можем видеть, что кратчайший путь от маршрутизатора A до маршрутизатор H есть $\langle A, C, F, G, H \rangle$.

5.4. Маршрутизация с учетом состояния линии и алгоритм Дейкстры

Уже к началу 1980-х годов появились сомнения в масштабируемости маршрутизации по вектору расстояний. Проблемы вызывали следующие два аспекта:

- В среде, где топология сети часто изменяется, маршрутизация по вектору расстояний сходилась слишком медленно, чтобы поддерживать точную информацию о расстояниях.
- Сообщения с обновлениями содержали расстояния до всех узлов, так что размер сообщения рос вместе с размером всей сети.

Для решения этих проблем была разработана *маршрутизация с учетом состояния линии* (Link-State Routing) [28, 37]. При такой маршрутизации каждый маршрутизатор хранит графовое представление топологии всей сети и вычисляет свою таблицу маршрутизации по этому графу, используя алгоритм Дейкстры. Для поддержания графа в актуальном состоянии маршрутизаторы совместно используют информацию о состоянии линий: какие линии открыты («up»), а какие нет («down»). Когда возможности связи изменяются, по всей сети распространяется информация в виде *объявления состояния линии*.

Поскольку совместно использовать нужно только локальную информацию (связь с соседями), маршрутизация с учетом состояния линии не страдает от проблем с большим размером сообщения, как это происходит при маршрутизации по вектору расстояний. Кроме того, так как каждый маршрутизатор вычисляет свои собственные кратчайшие расстояния, реакция на изменения в сети и перевычисление точных таблиц маршрутизации занимает намного меньше времени. Недостатком маршрутизации с учетом состояния линии является то, что она требует от маршрутизатора больше вычислений и памяти. Но даже с учетом этого данный вид маршрутизации признан эффективным и формализован в виде *протокола маршрутизации с определением кратчайшего маршрута* (Open Shortest Path First

protocol, OSPF) [33]. Сейчас он является одним из предпочитаемых протоколов для внутренней маршрутизации между шлюзами.

Алгоритм Дейкстры находит все кратчайшие пути из исходной вершины до каждой вершины графа, последовательно наращивая множество вершин S, для которых известен кратчайший путь. На каждом шаге алгоритма вершина из множества $V - S$ с наименьшей меткой расстояния добавляется к S. Затем исходящие ребра вершины релаксируются с использованием того же способа, что и в алгоритме Беллмана–Форда, по формуле $d[v] = \min(w(u, v) + d[u], d[v])$. Затем алгоритм повторяет цикл, обрабатывая следующую вершину из $V - S$ с наименьшей меткой расстояния. Алгоритм завершается, когда в S оказываются все вершины, достижимые из исходной.

В оставшейся части этого раздела мы покажем, как использовать функцию `dijkstra_shortest_paths()` из BGL при решении задачи поиска кратчайшего пути из одной вершины в сети маршрутизаторов и как вычислять таблицу маршрутизации. Пример сети, описанной в RFC 1583, изображен на рис. 5.3. Интернет-маршрутизаторы используют единый протокол маршрутизации. RT обозначает маршрутизатор (router), N — сеть (network), под которой понимается блок адресов, трактуемых как единый пункт назначения, H — хост. Веса ребер указывают стоимость передачи.

Для демонстрации алгоритма Дейкстры мы вычислим дерево кратчайших путей для маршрутизатора RT6. Основные шаги программы показаны в листинге 5.4.

Листинг 5.4. Вычисление дерева кратчайших путей для маршрутизатора RT6

```
⟨ ospf-example.cpp ⟩ ≡
#include <fstream>                    // для файлового ввода-вывода
#include <boost/graph/graphviz.hpp>   // для read/write_graphviz()
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/lexical_cast.hpp>
int main()
{
  using namespace boost;
  ⟨ Читать орграф из dot-файла Graphviz ⟩
  ⟨ Копировать орграф, преобразуя строковые метки в целые веса ⟩
  ⟨ Найти шестой маршрутизатор ⟩
  ⟨ Подготовить отображение свойства родителей
    для записи дерева кратчайших путей ⟩
  ⟨ Выполнить алгоритм Дейкстры ⟩
  ⟨ Покрасить все ребра дерева кратчайших путей черным ⟩
  ⟨ Записать новый граф в dot-файл Graphviz ⟩
  ⟨ Записать таблицу маршрутизации для шестого маршрутизатора ⟩
  return EXIT_SUCCESS;
}
```

На первом шаге создается граф. Граф на рис. 5.3 представлен как dot-файл Graphviz. Пакет Graphviz предоставляет инструменты для автоматической компоновки и рисования графов. Он доступен на сайте www.graphviz.org. Программы Graphviz используют специальный формат файла для графов, называемый dot-файлами. Библиотека BGL включает в себя анализатор для чтения этого формата в граф BGL. Анализатор можно использовать с помощью функции `read_graphviz()`, определенной в `boost/graph/graphviz.hpp`. Так как граф ориентированный, мы используем тип `GraphvizDigraph`. Для неориентированного графа применяется тип `GraphvizGraph`.

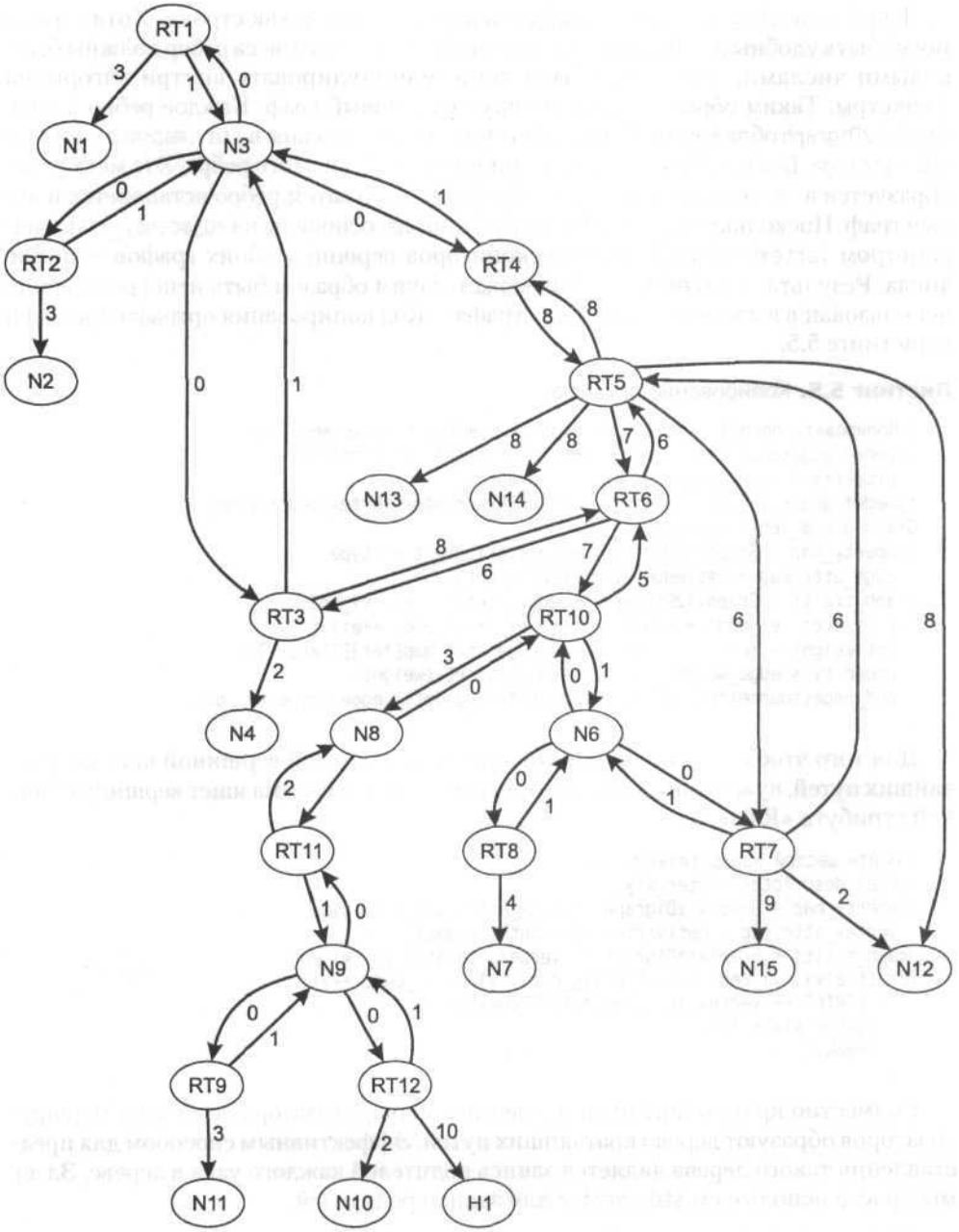


Рис. 5.3. Интернет-маршрутизаторы в виде ориентированного графа

```
⟨ Читать орграф из dot-файла Graphviz ⟩ ≡
GraphvizDigraph g_dot;
read_graphviz("figs/ospf-graph.dot", g_dot);
```

Тип GraphvizDigraph хранит свойства вершин и ребер как строки. Хотя строки могут быть удобны для файлового ввода-вывода и печати, веса ребер должны быть целыми числами, чтобы ими было легко манипулировать внутри алгоритма Дейкстры. Таким образом, g_dot копируется в новый граф. Каждое ребро в типе GraphvizDigraph обладает рядом атрибутов, которые записаны в std::map<std::string, std::string>. Веса ребер хранятся в атрибуте «label» каждого ребра. Эта метка преобразуется в int при помощи boost::lexical_cast, и затем ребро вставляется в новый граф. Поскольку типы Graph и GraphvizDigraph основаны на adjacency_list с параметром VertexList=vecS, типы дескрипторов вершин у обоих графов — целые числа. Результат source(*ei, g_dot) может, таким образом быть непосредственно использован в вызове add_edge() для графа g. Код копирования орграфа приведен в листинге 5.5.

Листинг 5.5. Копирование орграфа

```
⟨ Копировать орграф, преобразуя строковые метки в целые веса ⟩ ≡
typedef adjacency_list < vecS, vecS, directedS, no_property,
  property < edge_weight_t, int > > Graph;
typedef graph_traits < Graph >::vertex_descriptor vertex_descriptor;
Graph g(num_vertices(g_dot));
property_map < GraphvizDigraph, edge_attribute_t >::type
  edge_attr_map = get(edge_attribute, g_dot);
graph_traits < GraphvizDigraph >::edge_iterator ei, ei_end;
for (tie(ei, ei_end) = edges(g_dot); ei != ei_end; ++ei) {
  int weight = lexical_cast < int >(edge_attr_map[*ei]["label"]);
  property < edge_weight_t, int >edge_property(weight);
  add_edge(source(*ei, g_dot), target(*ei, g_dot), edge_property, g);
}
```

Для того чтобы шестой маршрутизатор стал исходной вершиной поиска кратчайших путей, нужно определить его дескриптор. Программа ищет вершину с меткой атрибута «RT6».

```
⟨ Найти шестой маршрутизатор ⟩ ≡
vertex_descriptor router_six;
property_map < GraphvizDigraph, vertex_attribute_t >::type
  vertex_attr_map = get(vertex_attribute, g_dot);
graph_traits < GraphvizDigraph >::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g_dot); vi != vi_end; ++vi)
  if ("RT6" == vertex_attr_map[*vi]["label"]) {
    router_six = *vi;
    break;
  }
```

Совместно кратчайшие пути от шестого маршрутизатора до других маршрутизаторов образуют дерево кратчайших путей. Эффективным способом для представления такого дерева является запись родителей каждого узла в дереве. Здесь мы просто используем std::vector для записи родителей.

```
⟨ Подготовить отображение свойства родителей
 для записи дерева кратчайших путей ⟩ ≡
std::vector < vertex_descriptor > parent(num_vertices(g));
//  Все вершины вначале являются своими родителями
typedef graph_traits < Graph >::vertices_size_type size_type;
```

```
for (size_type p = 0; p < num_vertices(g); ++p)
  parent[p] = p;
```

Теперь все готово для вызова алгоритма Дейкстры. Мы передаем ему массив родителей через именованный параметр predecessor_map().

```
⟨ Выполнить алгоритм Дейкстры ⟩ ≡
dijkstra_shortest_paths(g, router_six, predecessor_map(&parent[0]));
```

Для подготовки к выводу графа в dot-файл Graphviz мы «красим» цвета древесных ребер в черный цвет. Древесные ребра были записаны в массив parent. Для каждой вершины i ребро (parent[i], i) является древесным ребром, если не выполняется равенство parent[i] = i. В противном случае i — корневая или недостижимая вершина.

```
⟨ Покрасить все ребра дерева кратчайших путей в черный ⟩ ≡
graph_traits < GraphvizDigraph >::edge_descriptor e;
for (size_type i = 0; i < num_vertices(g); ++i)
  if (parent[i] != i) {
    e = edge(parent[i], i, g_dot).first;
    edge_attr_map[e]["color"] = "black";
}
```

Теперь мы можем записать граф в dot-файл. Для ребер цветом по умолчанию выберем серый (для недревесных ребер). Вычисленное дерево кратчайших путей для шестого маршрутизатора представлено на рис. 5.4.

```
⟨ Записать новый граф в dot-файл Graphviz ⟩ ≡
graph_property < GraphvizDigraph, graph_edge_attribute_t >::type &
  graph_edge_attr_map = get_property(g_dot, graph_edge_attribute);
graph_edge_attr_map["color"] = "grey";
write_graphviz("figs/ospf-sptree.dot", g_dot);
```

На последнем шаге вычисляется таблица маршрутизации для шестого маршрутизатора. Таблица маршрутизации имеет три колонки: пункт назначения (destination), следующий переход (hop) в сторону узла назначения и полное расстояние до узла назначения. Для заполнения таблицы маршрутизации создаются записи для каждого пункта назначения в сети. Информация для каждого узла может быть получена обратным прохождением кратчайшего пути от назначения до шестого маршрутизатора по отображению родителей. Узлы, являющиеся своими собственными родителями, пропускаются, так как это либо сам шестой маршрутизатор, либо недостижимый узел.

```
⟨ Записать таблицу маршрутизации для шестого маршрутизатора ⟩ ≡
std::ofstream rtable("routing-table.dat");
rtable << "Dest    Next Hop    Total Cost" << std::endl;
for (tie(vi, vi_end) = vertices(g_dot); vi != vi_end; ++vi)
  if (parent[*vi] != *vi) {
    rtable << vertex_attr_map[*vi]["label"] << "     ";
    ⟨ Следовать по пути назад к шестому маршрутизатору по родителям ⟩
    }
```

Во время следования по пути из конечного пункта к шестому маршрутизатору веса путей суммируются в path_cost. Мы также записываем дочерний узел текущей вершины, так как по завершении цикла эта вершина будет использоваться в качестве следующего перехода.

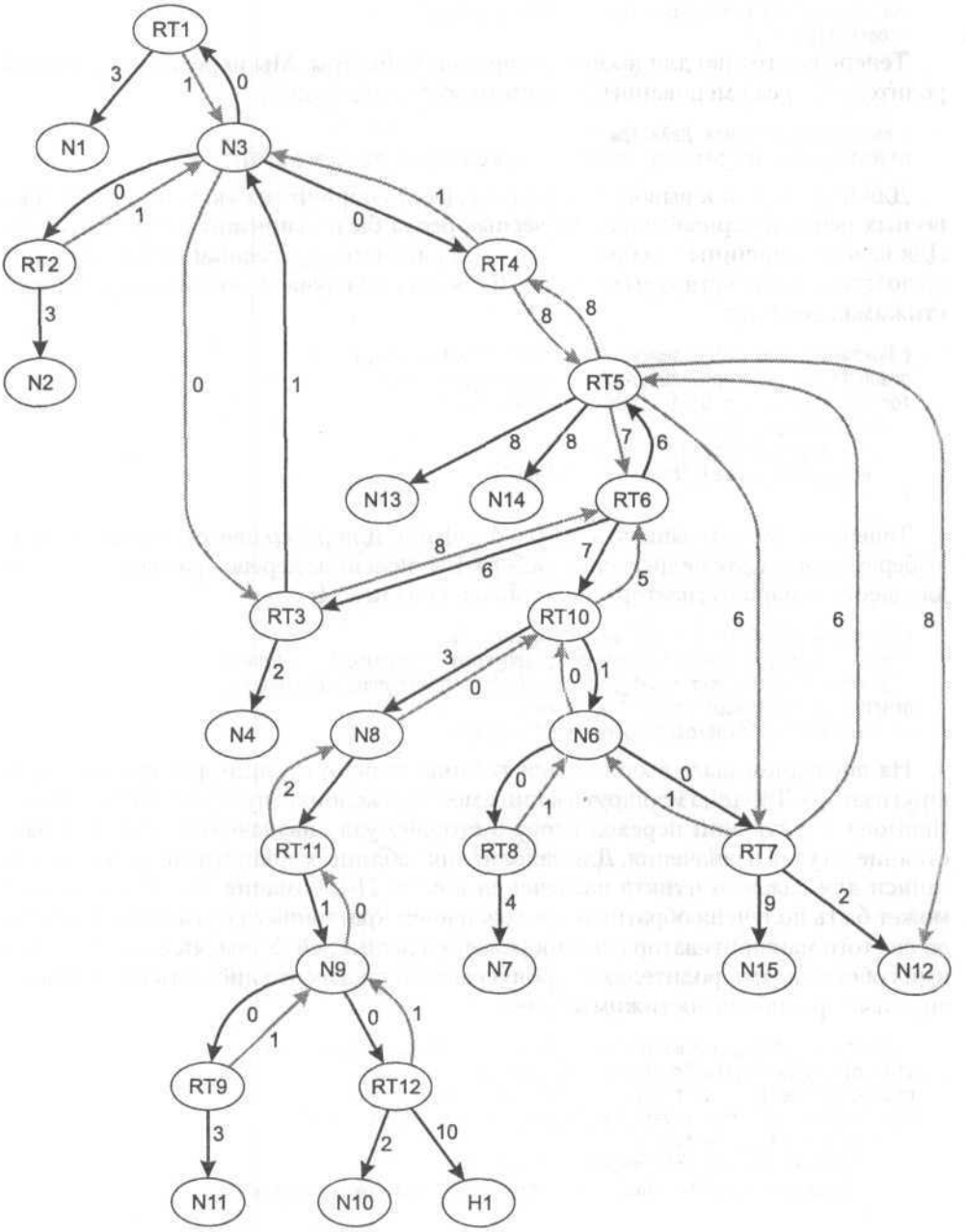


Рис. 5.4. Дерево кратчайших путей для шестого маршрутизатора

```
⟨ Следовать по пути назад к шестому маршрутизатору по родителям ⟩ ≡
vertex_descriptor v = *vi, child;
int path_cost = 0;
property_map < Graph, edge_weight_t >::type
```

```
weight_map = get(edge_weight, g);
do {
  path_cost += get(weight_map, edge(parent[v], v, g).first);
  child = v;
  v = parent[v];
} while (v != parent[v]);
rtable << vertex_attr_map[child]["label"] << "      ";
rtable << path_cost << std::endl;
```

Результирующая таблица маршрутизации выглядит так:

Назначение	След.переход	Полная стоимость
Dest	Next Hop	Total Cost
RT1	RT3	7
RT2	RT3	7
RT3	RT3	6
RT4	RT3	7
RT5	RT5	6
RT7	RT10	8
RT8	RT10	8
RT9	RT10	11
RT10	RT10	7
RT11	RT10	10
RT12	RT10	11
N1	RT3	10
N2	RT3	10
N3	RT3	7
N4	RT3	8
N6	RT10	8
N7	RT10	12
N8	RT10	10
N9	RT10	11
N10	RT10	13
N12	RT10	10
N13	RT5	14
N14	RT5	14
N15	RT10	17
H1	RT10	21

Задача минимального остовного дерева

6

Библиотека Boost Graph Library реализует два классических алгоритма для нахождения минимального остовного дерева: алгоритм Краскала [23] и алгоритм Прима [38]. Задача нахождения минимального остовного дерева появляется во многих приложениях, таких как планирование телефонной сети, построение монтажной электрической схемы, упаковка данных. В этой главе мы применяем алгоритмы BGL к задаче планирования телефонной сети.

6.1. Определения

Задачу минимального остовного дерева можно определить следующим образом. Для данного неориентированного графа $G = (V, E)$ необходимо найти ациклическое подмножество его ребер $T \subseteq E$, которое соединяет все вершины в графе и общий вес которого минимален. Общий вес остовного дерева — это сумма весов ребер из T:

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

Ациклическое подмножество ребер, которое соединяет все вершины графа, называется *остовным деревом*. *Минимальным остовным деревом* называется остовное дерево T с минимальным полным весом.

6.2. Планирование телефонной сети

Предположим, что в ваши обязанности входит проведение телефонных линий для некоторого отдаленного региона. Регион состоит из нескольких городов и дорожной сети. Проведение телефонной линии требует возможности подъезда на грузовике, а значит, необходимости дороги вдоль всего маршрута линии. Ваш бюд-

жет достаточно невелик и строительства новых дорог не предусматривает: телефонные линии должны идти вдоль существующих дорог. Также желательно минимизировать общую длину телефонного кабеля, который потребуется для соединения всех населенных пунктов региона.

Поскольку регион малонаселен, такие факторы, как пропускная способность, не рассматриваются. Пример отдаленного региона с городами, соединенными между собой дорогами, в виде взвешенного графа представлен на рис. 6.1. Длины дорог обозначены в милях. Нашей целью является нахождение оптимального расположения телефонных линий. Сначала мы решим эту задачу, используя алгоритм Краскала, а затем с помощью алгоритма Прима.

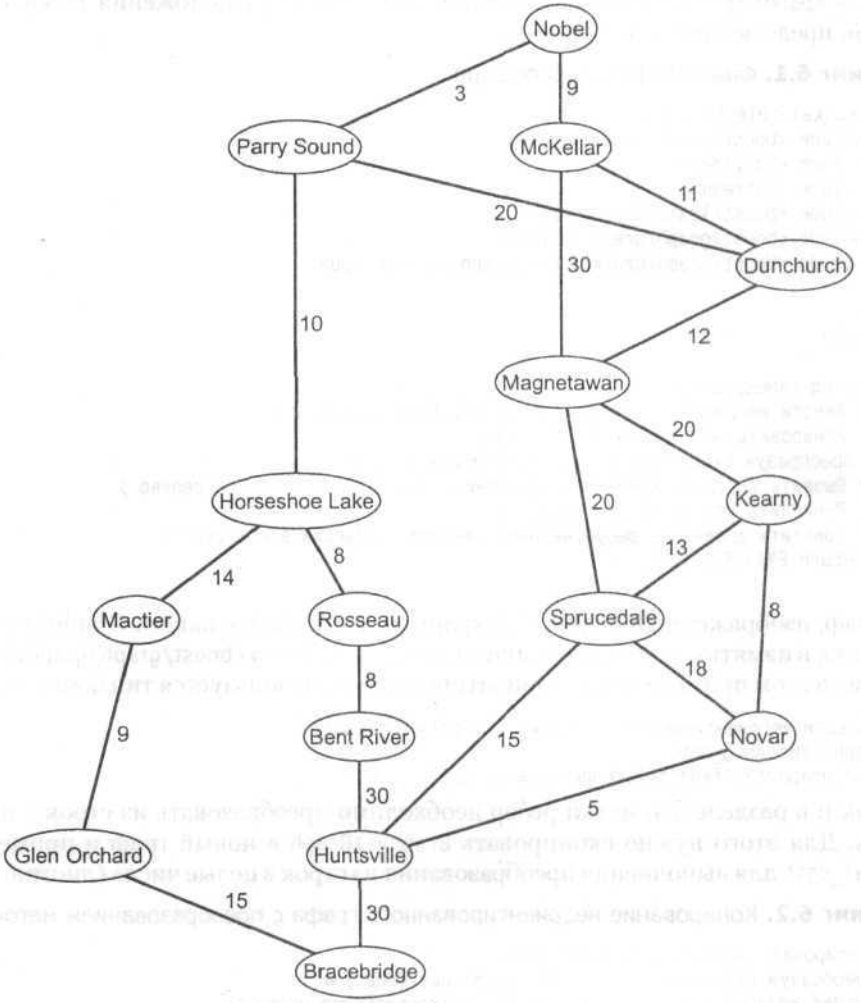


Рис. 6.1. Города, соединенные сетью дорог, в виде взвешенного графа

6.3. Алгоритм Краскала

Алгоритм Краскала стартует с того, что каждая вершина является сама по себе деревом, без ребер из множества T, которое и составит минимальное остовное дерево. Затем алгоритм проверяет каждое ребро графа в порядке увеличения веса ребра. Если ребро соединяет две вершины в разных деревьях, алгоритм сливает эти деревья в одно и добавляет ребро к множеству T. После того как все ребра будут просмотрены, дерево T покроет граф (если он связный) и станет минимальным остовным деревом графа.

Схема файла kruskal-telephone.cpp, в котором применяется функция `kruskal_minimum_spanning_tree()` для вычисления наилучшего расположения телефонных линий, представлена в листинге 6.1.

Листинг 6.1. Файл kruskal-telephone.cpp

```
⟨ kruskal-telephone.cpp ⟩ ≡
#include <boost/config.hpp>
#include <iostream>
#include <fstream>
#include <boost/lexical_cast.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/kruskal_min_spanning_tree.hpp>

int
main()
{
  using namespace boost;
  ⟨ Ввести неориентированный граф из dot-файла Graphviz ⟩
  ⟨ Копировать неориентированный граф,
   преобразуя строковые метки в целочисленные веса ⟩
  ⟨ Вызвать алгоритм Краскала и сохранить минимальное остовное дерево ⟩
  ⟨ Вычислить вес остовного дерева ⟩
  ⟨ Пометить древесные ребра черными линиями и вывести в dot-файл ⟩
  return EXIT_SUCCESS;
}
```

Граф, изображенный на рис. 6.1, хранится в dot-файле пакета Graphviz и считывается в память с помощью функции `read_graphviz()` из boost/graph/graphviz.hpp. Так как в этом примере граф неориентированный, используется тип `GraphvizGraph`.

```
⟨ Ввести неориентированный граф из dot-файла Graphviz ⟩ ≡
GraphvizGraph g_dot;
read_graphviz("figs/telephone-network.dot", g_dot);
```

Как и в разделе 5.4, метки ребер необходимо преобразовать из строк в целые числа. Для этого нужно скопировать `GraphvizGraph` в новый граф и применить `lexical_cast` для выполнения преобразования из строк в целые числа (листинг 6.2).

Листинг 6.2. Копирование неориентированного графа с преобразованием меток

```
⟨ Копировать неориентированный граф,
 преобразуя строковые метки в целочисленные веса ⟩ ≡
typedef adjacency_list < vecS, vecS, undirectedS, no_property,
  property < edge_weight_t, int > > Graph;
Graph g(num_vertices(g_dot));
property_map < GraphvizGraph, edge_attribute_t >::type
```

```
  edge_attr_map = get(edge_attribute, g_dot);
graph_traits < GraphvizGraph >::edge_iterator ei, ei_end;
for (tie(ei, ei_end) = edges(g_dot); ei != ei_end; ++ei) {
  int weight = lexical_cast < int >(edge_attr_map[*ei]["label"]);
  property < edge_weight_t, int >edge_property(weight);
  add_edge(source(*ei, g_dot), target(*ei, g_dot), edge_property, g);
}
```

Вызов алгоритма Краскала требует, чтобы тип графа был одновременно VertexListGraph и EdgeListGraph. В разделе 14.1.1 в описании `adjacency_list` показывает, что выбранный нами тип `Graph` прекрасно подходит. Для хранения вывода алгоритма (ребер минимального остовного дерева) мы используем `std::vector mst` и применяем `std::back_inserter()` для создания из него итератора вывода. Алгоритм Краскала имеет несколько именованных параметров. В нашем примере мы ими не пользуемся, поэтому все они принимают значения по умолчанию. Отображение весов и отображение индексов вершин по умолчанию извлекаются из графа (это внутренние свойства). Имя `edge_weight_t` объявлено как свойство типа `Graph`, и отображение индексов вершин уже есть в `adjacency_list` с `VertexList=vecS`. Отображения предшественников и рангов (используются только в пределах алгоритма Краскала) по умолчанию создаются внутри алгоритма.

```
⟨ Вызвать алгоритм Краскала и сохранить минимальное остовное дерево ⟩ ≡
std::vector < graph_traits < Graph >::edge_descriptor > mst;
kruskal_minimum_spanning_tree(g, std::back_inserter(mst));
```

Когда алгоритм завершил свою работу, минимальное остовное дерево хранится в `mst`. Полный вес этого дерева вычисляется сложением весов ребер в `mst`. В нашем примере полный вес для минимального остовного дерева составляет 145 миль.

```
⟨ Вычислить вес остовного дерева ⟩ ≡
property_map < Graph, edge_weight_t >::type_weight = get(edge_weight, g);
int total_weight = 0;
for (int e = 0; e < mst.size(); ++e)
  total_weight += get(weight, mst[e]);
std::cout << "полный вес: " << total_weight << std::endl;
```

Древесные ребра затем окрашиваются в черный цвет, и граф сохраняется в dot-файле (листинг 6.3).

Листинг 6.3. Отмечаем древесные ребра для вывода

```
⟨ Пометить древесные ребра черными линиями и вывести в dot-файл⟩ ≡
typedef graph_traits < Graph >::vertex_descriptor Vertex;
for (int i = 0; i < mst.size(); ++i) {
  Vertex u = source(mst[i], g), v = target(mst[i], g);
  edge_attr_map[edge(u, v, g_dot).first]["color"] = "black"; //  черный цвет
}
std::ofstream out("figs/telephone-mst-kruskal.dot");
graph_property < GraphvizGraph, graph_edge_attribute_t >::type &
  graph_edge_attr_map = get_property(g_dot, graph_edge_attribute);
graph_edge_attr_map["color"] = "gray";   //  серый цвет
graph_edge_attr_map["style"] = "bold";   //  полужирный стиль
write_graphviz(out, g_dot);
```

Результирующее минимальное остовное дерево показано на рис. 6.2. Оптимальное расположение телефонных линий отображается черными линиями.

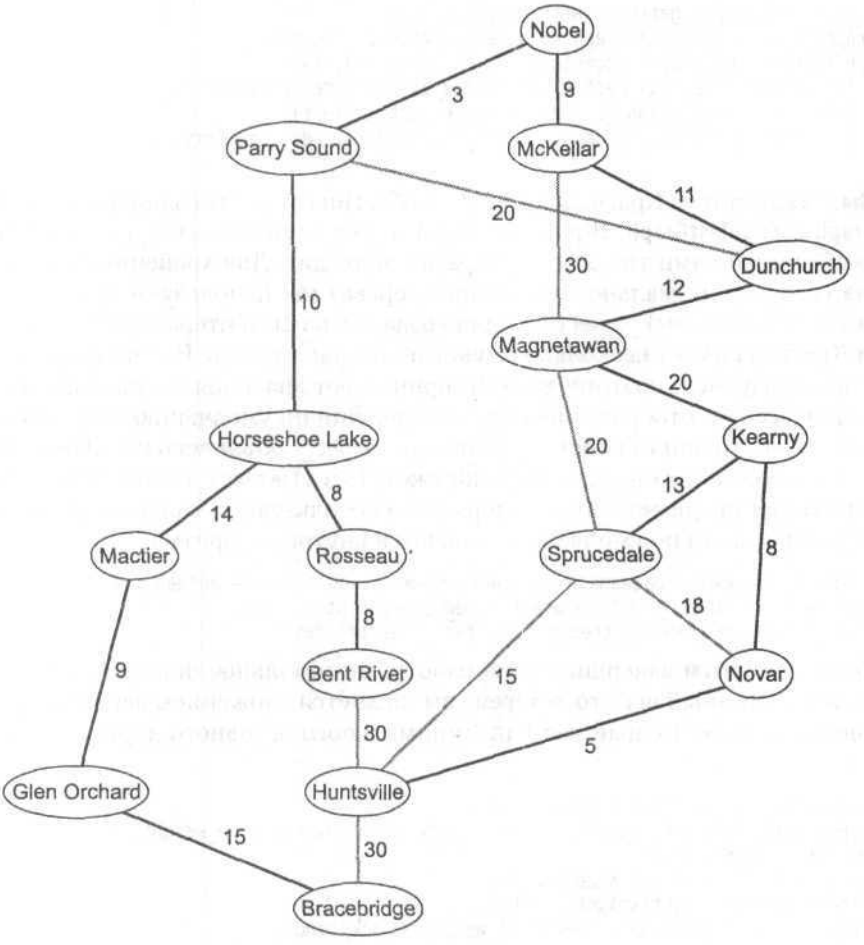


Рис. 6.2. Минимальное остовное дерево для алгоритма Краскала

6.4. Алгоритм Прима

Алгоритм Прима наращивает минимальное остовное дерево по одной вершине за раз (а не одно ребро за раз, как в алгоритме Краскала). Основная идея алгоритма Прима состоит в последовательном добавлении вершин к минимальному остовному дереву. Очередная вершина должна иметь общее ребро с минимальным весом, соединяющее ее с уже находящейся в дереве вершиной. Алгоритм Прима очень похож на алгоритм Дейкстры. (На самом деле реализация алгоритма Прима в BGL есть просто вызов алгоритма Дейкстры со специально подобранными функциями сравнения расстояний и объединения.)

В этом разделе алгоритм `prim_minimum_spanning_tree()` применяется к той же самой задаче планирования телефонной сети (см. рис. 6.1). Схема файла **prim-telephone.cpp** (листинг 6.4) подобна использованной в предыдущем разделе, хотя имеется некоторая разница в том, как алгоритм Прима выводит ребра остовного дерева.

Листинг 6.4. Файл prim-telephone.cpp

```
⟨ prim-telephone.cpp ⟩ ≡
#include <iostream>
#include <fstream>
#include <vector>
#include <boost/lexical_cast.hpp>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/prim_minimum_spanning_tree.hpp>

int main()
{
  using namespace boost;
  ⟨ Ввести неориентированный граф из dot-файла Graphviz ⟩
  ⟨ Копировать неориентированный граф,
   преобразуя строковые метки в целочисленные веса ⟩
  ⟨ Вызвать алгоритм Прима
   и сохранить минимальное остовное дерево в предшественниках ⟩
  ⟨ Вычислить вес остовного дерева ⟩
  ⟨ Пометить древесные ребра черными линиями и вывести в dot-файл ⟩
  return EXIT_SUCCESS;
}
```

Первые два шага (чтение из dot-файла и копирование графа) те же самые, что и в предыдущем примере. В вызове алгоритма Прима первый параметр — граф, а второй — отображение предшественников. Минимальное остовное дерево записывается алгоритмом Прима в виде отображения предшественников. Для каждой вершины v в графе `parent[v]` является родителем v в минимальном остовном дереве. Внутри алгоритма `parent[v]` может быть многократно присвоен, но последнее присваивание гарантированно устанавливает правильного родителя. Для точной настройки функции `prim_minimum_spanning_tree()` может быть применен ряд именованных параметров (мы оставляем значения по умолчанию). В нашем случае используются отображения весов ребер и индексов вершин, внутренние для типа `Graph`, а вспомогательные отображения цветов и расстояний создаются внутри алгоритма. Значением по умолчанию для корневой вершины является `*vertices(g).first`, что допустимо, так как корень минимального остовного дерева произволен.

```
⟨ Вызвать алгоритм Прима
 и сохранить минимальное остовное дерево в предшественниках ⟩ ≡
typedef graph_traits<Graph>::vertex_descriptor Vertex;
std::vector<Vertex> parent(num_vertices(g));
prim_minimum_spanning_tree(g, &parent[0]);
```

Когда минимальное остовное дерево записано в массиве `parent`, общий вес вычисляется в цикле по всем вершинам графа сложением весов всех ребер `(parent[v], v)`. Если `parent[v] = v`, это означает, что либо v является корневой вершиной дерева, либо она не была в той же компоненте связности, что и остальные вершины. В любом случае `(parent[v], v)` не является ребром остовного дерева и должно быть пропущено. Вычисление для графа, изображенного на рис. 6.1, дает тот же результат — 145 миль.

```
⟨ Вычислить вес остовного дерева ⟩ ≡
property_map<Graph, edge_weight_t>::type_weight = get(edge_weight, g);
int total_weight = 0;
for (int v = 0; v < num_vertices(g); ++v)
  if (parent[v] != v)
    total_weight += get(weight, edge(parent[v], v, g).first);
std::cout << "полный вес: " << total_weight << std::endl;
```

Для наглядности ребра минимального остовного дерева отмечены черными линиями и после записываются в dot-файл. Результирующее минимальное остовное дерево показано на рис. 6.3. Заметьте, что дерево здесь немного отличается от полученного алгоритмом Краскала. Вместо ребра между Magnetawan и Kearny имеется ребро между Magnetawan и Sprucedale. Это подчеркивает тот факт, что минимальные остовные деревья не уникальны: для конкретного графа их может быть более одного.

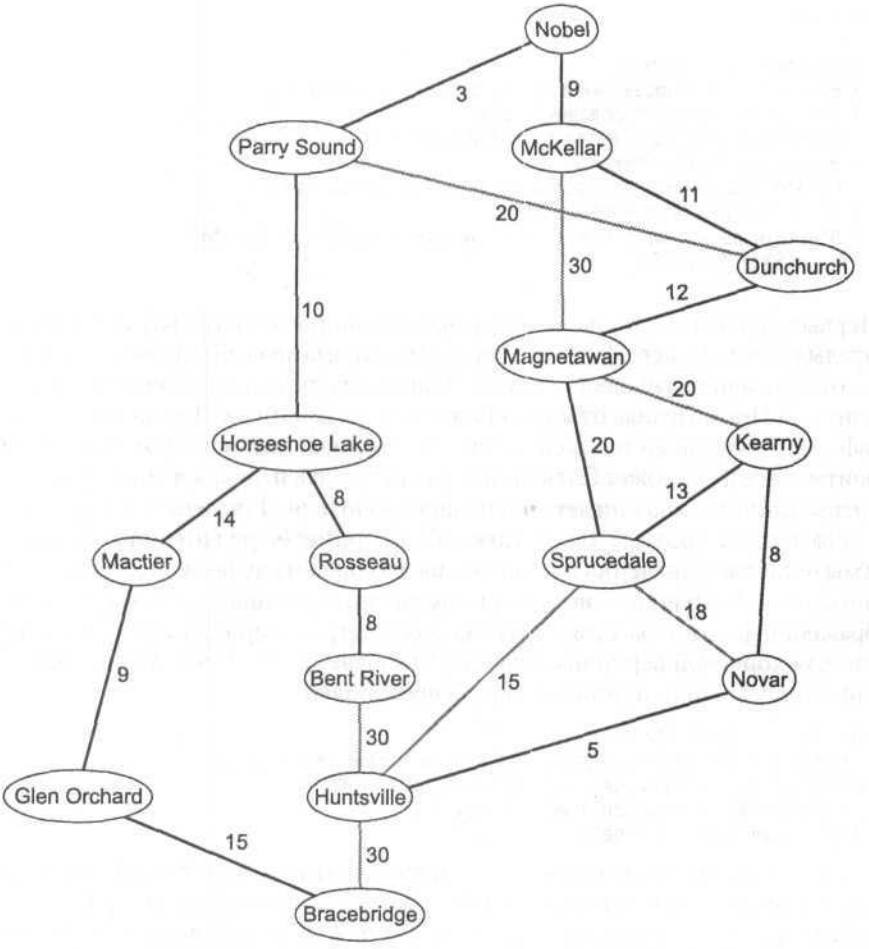


Рис. 6.3. Минимальное остовное дерево для алгоритма Прима

```
⟨ Пометить древесные ребра черными линиями и вывести в dot-файл ⟩ ≡
for (int u = 0; u < num_vertices(g); ++u)
  if (parent[u] != u)
    edge_attr_map[edge(parent[u], u, g_dot).first]["color"] = "black";
std::ofstream out("figs/telephone-mst-prim.dot");
graph_property < GraphvizGraph, graph_edge_attribute_t >::type &
  graph_edge_attr_map = get_property(g_dot, graph_edge_attribute);
graph_edge_attr_map["color"] = "gray";
write_graphviz(out, g_dot);
```

Компоненты связности 7

Основным вопросом для сети является вопрос о достижимости одних вершин из других. Например, хорошо спроектированный веб-сайт должен иметь достаточно ссылок между страницами, чтобы все страницы были достижимы с главной страницы. Кроме того, хорошим тоном является наличие ссылок на главную страницу или хотя бы на предыдущую страницу в серии последовательных страниц. В ориентированном графе группы вершин, достижимые друг из друга, называются *сильными компонентами связности*[1].

Изучение 200 миллионов веб-страниц показывает, что 56 миллионов страниц Интернета составляют одну большую сильную компоненту связности [7]. Это исследование также показало, что если рассматривать данный граф как неориентированный, большую компоненту связности составляют 150 миллионов страниц и 50 миллионов страниц оторваны от этой большой компоненты (они находятся в своих, гораздо меньших, компонентах связности).

Библиотека BGL предоставляет две функции для вычисления всех компонент связности: первая для однократного вычисления (граф не меняется) и вторая для случая, когда граф может расти. Библиотека BGL также реализует алгоритм Тарьяна для вычисления сильных компонент связности графа за линейное время.

В следующем разделе мы рассмотрим некоторые определения и затем применим функции из BGL для компонентов связности к Всемирной паутине WWW (World Wide Web).

7.1. Определения

Путем называется последовательность вершин, в которой каждая вершина пути соединена со следующей ребром. Если существует путь из вершины u в w, то мы будем говорить, что вершина w *достижима* из вершины u. *Компонента связности* — это группа вершин неориентированного графа, в которой каждая вершина

[1] Иногда они называются *бикомпонентами, сильными компонентами. — Примеч. перев.*

достижима из любой другой. *Сильная компонента связности* — это группа вершин ориентированного графа, которые обоюдно достижимы друг из друга. Отношение достижимости для неориентированных графов является *отношением эквивалентности*: оно рефлексивно, симметрично и транзитивно. Множество объектов, для которых выполняется отношение эквивалентности, образуют класс эквивалентности. Одна компонента связности является, таким образом, классом эквивалентности по отношению достижимости. Аналогично, сильная компонента связности является классом эквивалентности для отношения обоюдной достижимости. В результате эти два отношения разбивают вершины графа на непересекающиеся подмножества.

7.2. Связные компоненты и связность Интернета

Компоненты связности неориентированного графа вычисляются с применением поиска в глубину. Идея заключается в том, чтобы прогнать поиск в глубину на графе и отметить все вершины в одном дереве поиска в глубину как принадлежащие одной компоненте связности. Реализация функции `connected_components()` из BGL содержит вызов `depth_first_search()` со специальным объектом-посетителем, который присваивает каждой обнаруженной вершине номер текущей компоненты и увеличивает номер на единицу в событийной точке «начальная вершина».

На рис. 7.1 представлена сеть интернет-маршрутизаторов, у которой ребра соответствуют прямым соединениям. Шаги для вычисления компонент связности такой сети следующие: 1) прочитать данные о сети в память; 2) представить их в виде графа BGL; 3) вызвать функцию `connected_components()`. Каждой вершине графа присваивается целое число, отмечающее компоненту, к которой вершина принадлежит. Схема файла cc-internet.cpp приведена в листинге 7.1.

Листинг 7.1. Файл cc-internet.cpp

```
⟨ cc-internet.cpp ⟩ ≡
#include <fstream>
#include <vector>
#include <string>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/graphviz.hpp>

int main()
{
  using namespace boost;
  ⟨ Считать граф в память ⟩
  ⟨ Создать вектор для назначения компонент ⟩
  ⟨ Вызвать функцию connected_components() ⟩
  ⟨ Окрасить вершины в зависимости от компонент и записать в dot-файл ⟩
}
```

Данные для графа на рис. 7.1 считываются из файла cc-internet.dot, записанного в формате dot-файла Graphviz. Выбран тип `GraphvizGraph`, а не `GraphvizDigraph`, так как граф является неориентированным.

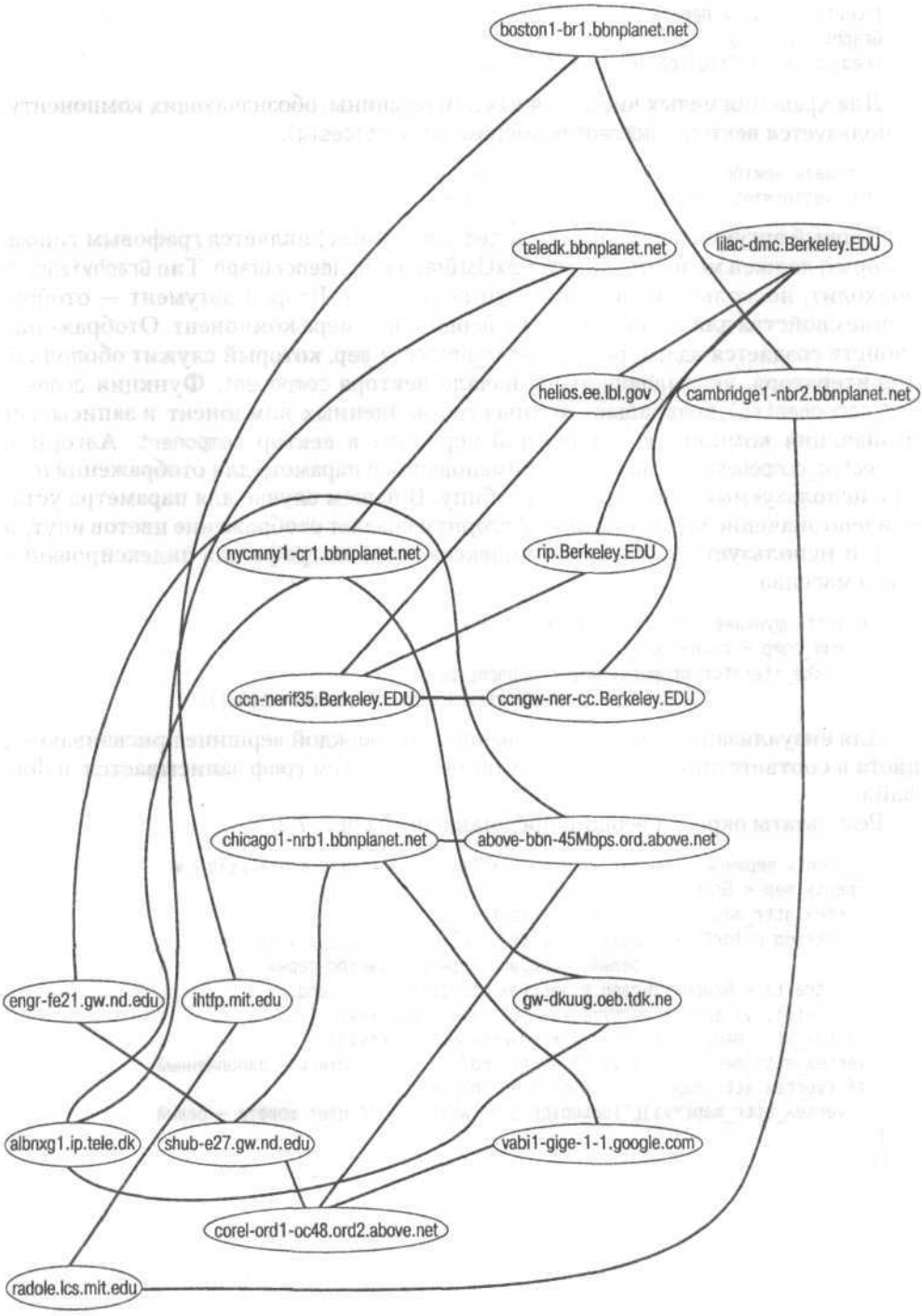


Рис. 7.1. Сеть интернет-маршрутизаторов

```
⟨ Считать граф в память ⟩ ≡
GraphvizGraph g;
read_graphviz("figs/cc-internet.dot", g);
```

Для хранения целых чисел для каждой вершины, обозначающих компоненту, используется вектор component размером num_vertices(g).

```
⟨ Создать вектор для назначения компонент ⟩ ≡
std::vector<int> component(num_vertices(g));
```

Первый аргумент в вызове connected_components() является графовым типом, который должен моделировать VertexListGraph и IncidenceGraph. Тип GraphvizGraph подходит, поскольку моделирует эти концепции. Второй аргумент — отображение свойства для преобразования вершин в номера компонент. Отображение свойств создается адаптером iterator_property_map, который служит оболочкой для итератора, указывающего на начало вектора component. Функция connected_components() возвращает количество найденных компонент и записывает назначения компонент для каждой вершины в вектор component. Алгоритм connected_components() также имеет именованный параметр для отображения цветов, используемый при поиске в глубину. В нашем случае для параметра установлено значение по умолчанию. Алгоритм создает отображение цветов внутри себя и использует отображение индексов вершин графа для индексирования этого массива.

```
⟨ Вызвать функцию connected_components() ⟩ ≡
int num_comp = connected_components(g,
     make_iterator_property_map(component.begin(),
                                get(vertex_index, g), component[0]));
```

Для визуализации результатов вычисления каждой вершине присваиваются цвета в соответствии с номером компоненты. Затем граф записывается в dot-файл.

Результаты окраски вершин представлены на рис. 7.2.

```
⟨ Окрасить вершины в зависимости от компонент и записать в dot-файл ⟩ ≡
property_map < GraphvizGraph, vertex_attribute_t >::type
  vertex_attr_map = get(vertex_attribute, g);
std::string color[] = {"white", "gray", "black", "lightgray"};
                    //  белый,   серый, черный,  светло-серый
graph_traits < GraphvizGraph >::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi) {
  vertex_attr_map[*vi]["color"] = color[component[*vi]];
  vertex_attr_map[*vi]["style"] = "filled";       // стиль — заполненный
  if (vertex_attr_map[*vi]["color"] == "black")
    vertex_attr_map[*vi]["fontcolor"] = "white";  // цвет шрифта — белый
}
write_graphviz("figs/cc-internet-out.dot", g);
```

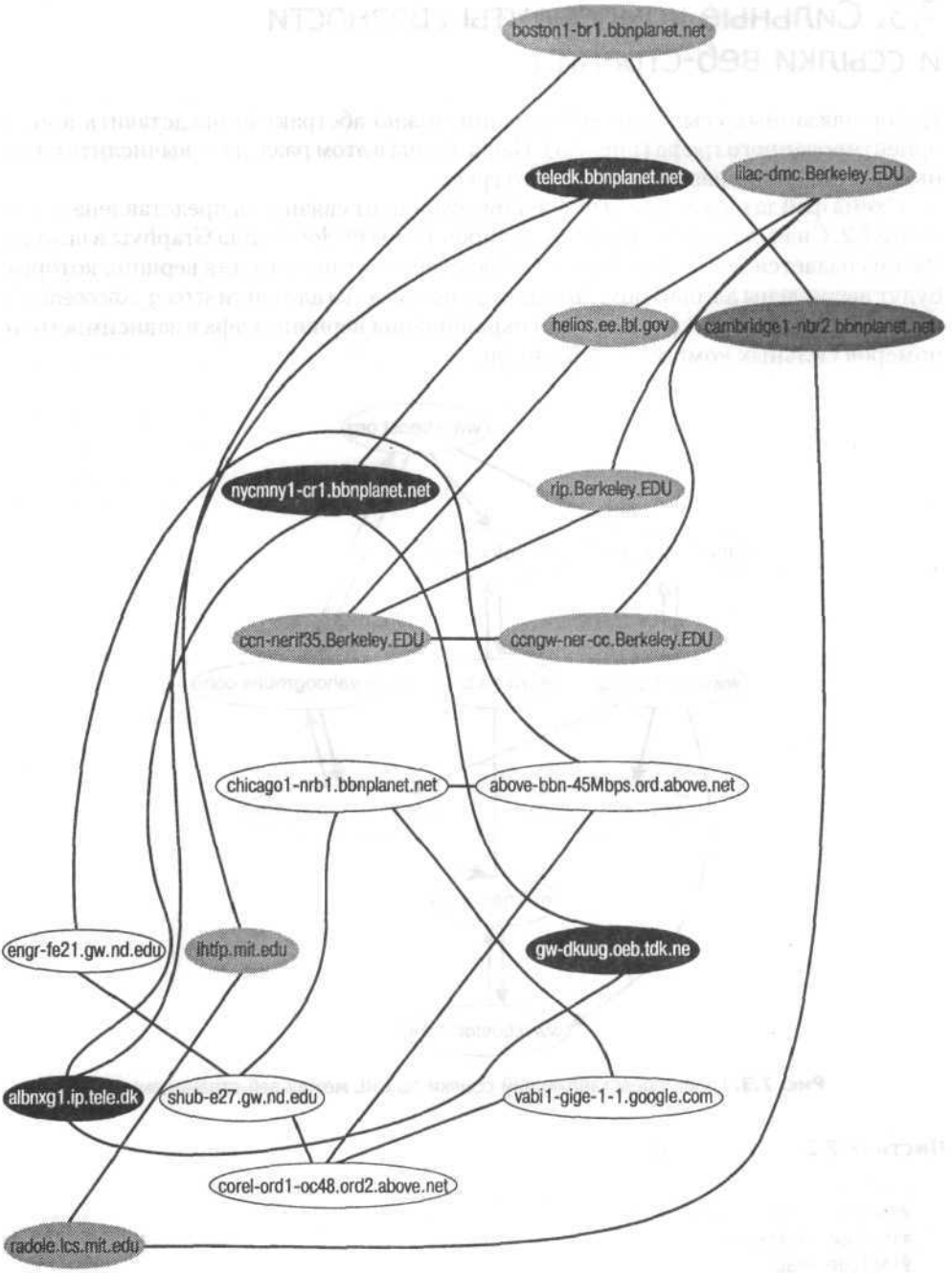


Рис. 7.2. Компоненты связности

7.3. Сильные компоненты связности и ссылки веб-страниц

Набор связанных ссылками веб-страниц можно абстрактно представить в виде ориентированного графа (рис. 7.3). Наша задача в этом разделе — вычислить сильные компоненты связности для этого графа.

Схема файла **scc.cpp** для вычисления компонент связности представлена в листинге 7.2. Сначала данные графа будут прочитаны из dot-файла Graphviz в память. Затем создается место для хранения назначений компонент для вершин, которые будут вычислены алгоритмом. После этого вызывается алгоритм `strong_components()`, и его результаты используются для окрашивания вершин графа в зависимости от номеров сильных компонент связности.

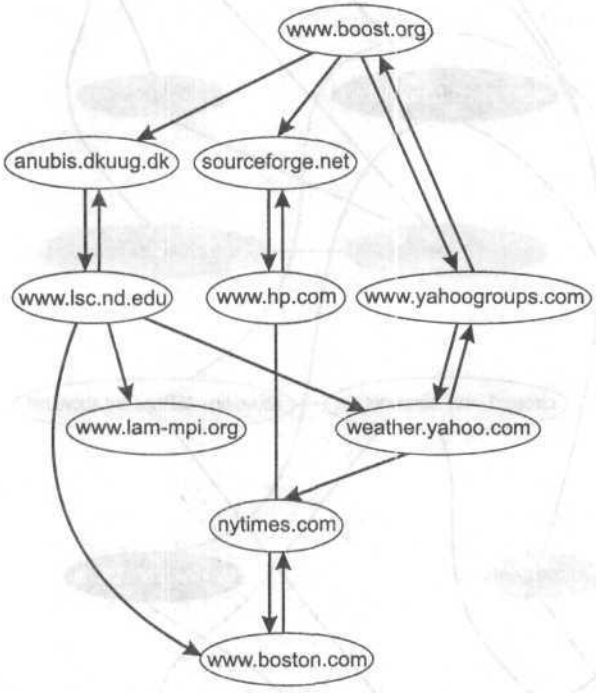


Рис. 7.3. Граф, представляющий ссылки по URL между веб-страницами

Листинг 7.2. Файл scc.cpp

```
⟨ scc.cpp ⟩ ≡
#include <boost/config.hpp>
#include <fstream>
#include <map>
#include <string>
```

```
#include <boost/graph/strong_components.hpp>
#include <boost/graph/graphviz.hpp>

int
main()
{
  using namespace boost;
  ⟨ Считать ориентированный граф в память ⟩
  ⟨ Выделить место для назначения компонент ⟩
  ⟨ Вызвать функцию strong_components() ⟩
  ⟨ Окрасить вершины в зависимости от номеров компонент
   и записать их в dot-файл ⟩
  return EXIT_SUCCESS;
}
```

Данные графа считываются из файла scc.dot с использованием типа `GraphvizDigraph` (так как граф ориентированный).

```
⟨ Считать ориентированный граф в память ⟩ ≡
GraphvizDigraph g;
read_graphviz("figs/scc.dot", g);
```

В вызове `strong_components()` адаптер `associative_property_map` используется для обеспечения интерфейса отбражения свойств, требуемого функцией. Этот адаптер создает отображение свойств из **AssociativeContainer**, например из `std::map`. Выбор `std::map` для реализации отображений свойств довольно неэффективен в нашем случае, но демонстрирует гибкость интерфейса отображения свойства. Дескриптор вершины для `GraphvizDigraph` является целым числом, поэтому он имеет требуемую `std::map` операцию «меньше».

```
⟨ Выделить место для назначения компонент ⟩ ≡
typedef graph_traits < GraphvizDigraph >::vertex_descriptor vertex_t;
std::map < vertex_t, int >component;
```

Результаты вызова `strong_components()` помещаются в массив `component`, где каждой вершине сопоставлен номер компоненты. Номера компонент идут от нуля до `num_comp - 1`. Граф, передаваемый функции `strong_components()`, должен быть моделью концепций **VertexListGraph** и **IncidenceGraph**. И действительно, он соответствует этим критериям. Второй аргумент — отображение компонент — должен быть **ReadWritePropertyMap**. Есть еще несколько именованных параметров, которые могут быть заданы, но все они предназначены для внутренних вспомогательных отображений. По умолчанию алгоритм создает массивы для этих отображений свойств и использует индекс вершины графа как смещение в этих массивах.

```
⟨ Вызвать функцию strong_components() ⟩ ≡
int num_comp = strong_components(g, make_assoc_property_map(component));
```

Программу завершает окрашивание вершин в соответствии с компонентами, которым они принадлежат. Вывод записывается в dot-файл. Граф с сильными компонентами связности представлен на рис. 7.4.

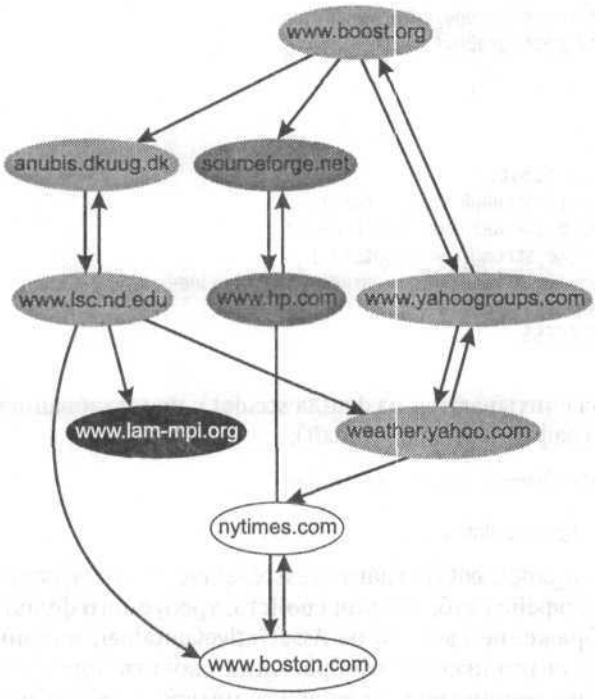


Рис. 7.4. Сильные компоненты связности

```
⟨ Окрасить вершины в зависимости от номеров компонент
  и записать их в dot-файл ⟩ ≡
property_map < GraphvizDigraph, vertex_attribute_t >::type
  vertex_attr_map = get(vertex_attribute, g);
std::string color[] = {"white", "gray", "black", "lightgray"};
                // белый,  серый, черный,  светло-серый
graph_traits < GraphvizDigraph >::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi) {
  vertex_attr_map[*vi]["color"] = color[component[*vi]];
  vertex_attr_map[*vi]["style"] = "filled";       // стиль – заполненный
  if (vertex_attr_map[*vi]["color"] == "black")
    vertex_attr_map[*vi]["fontcolor"] = "white"; // цвет шрифта – белый
}
write_graphviz("figs/scc-out.dot", g);
```

Максимальный поток

8

Задача максимального потока — это задача определения того, как много некоторого количества (например, воды) может пройти через сеть. Алгоритмы для решения задачи максимального потока имеют длинную историю. Первый алгоритм принадлежит Форду и Фалкерсону [12]. Лучший же из известных на сегодня алгоритмов общего назначения — *алгоритм проталкивания предпотока* (push-relabel algorithm) Гольдберга [9, 16, 17], основанный на понятии *предпотока* (preflow), введенном Карзановым [20]. Библиотека BGL содержит два алгоритма для вычисления максимального потока: алгоритм Эдмондса–Карпа (улучшение оригинального алгоритма Форда–Фалкерсона) и алгоритм проталкивания предпотока.

8.1. Определения

Потоковая сеть (flow network) — это ориентированный граф $G = (V, E)$ с *вершиной истока* s и *вершиной стока* t. Каждое ребро имеет положительную вещественную *пропускную способность*, и на каждой паре вершин задана *потоковая функция* f. Потоковая функция должна удовлетворять трем ограничениям:

$$f(u,v) \le c(u,v) \qquad \forall (u,v) \in V \times V \qquad \text{(ограничение пропускной способности)}$$

$$f(u,v) = -f(v,u) \qquad \forall (u,v) \in V \times V \qquad \text{(антисимметричность)}$$

$$\sum_{v \in V} f(u,v) = 0 \qquad \forall u \in V - \{s,t\} \qquad \text{(сохранение потока)}$$

Поток — чистый поток, входящий в вершину стока t. *Остаточная пропускная способность* ребра есть $r(u,v) = c(u,v) - f(u,v)$. Ребра с $r(u,v) > 0$ называются остаточными ребрами E_f и порождают граф $G_f = (V, E_f)$. Ребро с $r(u,v) = 0$ называется насыщенным.

$$|f| = \sum_{u \in V} f(u,t)$$

Задача максимального потока состоит в определении максимально возможного значения для $|f|$ и соответствующих значений для каждой пары вершин в графе.

Важным свойством потоковой сети является тот факт, что максимальный поток связан с пропускной способностью наиболее узкого места сети.

По теореме о максимальном потоке и минимальном разрезе [12] максимальное значение потока от вершины истока до вершины стока в потоковой сети равно минимальной пропускной способности среди всех (S, T) разрезов. *(S, T)-разрез* — это разделение вершин графа на два множества S и T, где $s \in S$ и $t \in T$. Любое ребро с начальной вершиной в S и конечной в T является *прямым ребром* разреза, а ребро с начальной вершиной в T и конечной в S — *обратным ребром* разреза. *Пропускная способность разреза* — это сумма пропускных способностей прямых ребер (обратные ребра игнорируются). Так что если мы рассмотрим пропускные способности всех разрезов, отделяющих s и t, и выберем разрез с минимальной пропускной способностью, она будет равна пропускной способности максимального потока в сети.

8.2. Реберная связность

При проектировании телефонной сети, сети передачи данных крупного предприятия или соединений маршрутизаторов для высокоскоростного сегмента сети Интернет очень важным вопросом, который задают себе инженеры, является устойчивость сети к повреждениям. К примеру, если кабель окажется перерезанным, имеются ли другие кабели, по которым может проходить информация? В теории графов это называются *реберной связностью* графа — минимальное число ребер, при разрезании которых получаются две несвязные компоненты (предполагается, что первоначально граф имел одну компоненту связности). Мы будем использовать $\alpha(G)$ для обозначения реберной связности графа. Множество ребер в разрезе, приводящем к увеличению числа компонент связности, называется *сечением* (minimum disconnecting set). Вершины графа разделяются по двум компонентам S^* и $\overline{S^*}$, поэтому для обозначения минимального разреза мы будем использовать $[S^*, \overline{S^*}]$. Оказывается, что вычисление реберной связности может быть сведено к последовательности решений задач о максимальном потоке. В этом разделе мы рассмотрим алгоритм вычисления реберной связности неориентированного графа [27].

Пусть $\alpha(u, v)$ обозначает минимальное число ребер, которое может быть разрезано для разъединения вершин u и v друг от друга. Если эти две вершины являются источником и стоком и пропускная способность каждого ребра равна единице, то разрез с минимальной пропускной способностью (вычисленной по алгоритму максимального потока) совпадет с минимальным разрезом. Таким образом, решив задачу максимального потока, мы также определим минимальное число ребер, которые должны быть разрезаны для разъединения двух вершин. Теперь для того, чтобы найти реберную связность графа, алгоритм максимального потока нужно запустить для каждой пары вершин. Минимум всех этих попарных минимальных разрезов будет минимальным разрезом всего графа.

Выполнять алгоритм максимального потока для каждой пары вершин слишком накладно, поэтому лучше сократить количество пар, которые требуется рас-

смотреть. Этого можно достигнуть с помощью специального свойства сечения $[S^*, \overline{S^*}]$. Пусть p — вершина минимальной степени, а δ — ее степень. Если $\alpha(G) = \delta$, то S^* есть просто p. Если $\alpha(G) \leq \delta - 1$, то оказывается, что для любого подмножества S^* (назовем его S), множество всех вершин, не являющихся соседними для вершин из S, должно быть не пусто. Это означает, что минимальный разрез может быть найден, если начать с $S = p$, выбрать вершину k из множества не соседних вершин S, вычислить $\alpha(p, k)$ и затем добавить k к S. Этот процесс повторяется, пока множество не соседних вершин S не станет пустым.

Мы реализуем алгоритм реберной связности как шаблон функции, который использует BGL-интерфейс VertexListGraph. Функция возвращает реберную связность графа, и ребра в *несвязном множестве* (disconnected set) записываются итератором вывода. Схема функции вычисления реберной связности приведена в листинге 8.1.

Листинг 8.1. Алгоритм вычисления реберной связности

```
⟨ Алгоритм вычисления реберной связности ⟩ ≡
template < typename VertexListGraph, typename OutputIterator >
typename graph_traits < VertexListGraph >::degree_size_type
edge_connectivity(VertexListGraph & g, OutputIterator disconnecting_set)
{
  ⟨ Определить типы ⟩
  ⟨ Определить переменные ⟩
  ⟨ Создать граф потоковой сети из неориентированного графа ⟩
  ⟨ Найти вершину минимальной степени и вычислить соседей S и не соседей S ⟩
  ⟨ Главный цикл ⟩
  ⟨ Вычислить прямые ребра разреза ⟩
  return c;
}
```

В первой части реализации (листинг 8.2) создаются некоторые определения типов, чтобы иметь более короткие имена для доступа к типам из свойств графа. Потоковый граф (ориентированный граф) создается на основе неориентированного входного графа g, поэтому используется графовый класс adjacency_list.

Листинг 8.2. Определение типов

```
⟨ Определить типы ⟩ ≡
typedef typename graph_traits <
        VertexListGraph >::vertex_descriptor vertex_descriptor;
typedef typename graph_traits <
        VertexListGraph >::degree_size_type degree_size_type;
typedef color_traits < default_color_type > Color;
typedef typename adjacency_list_traits < vecS, vecS,
        directedS >::edge_descriptor edge_descriptor;
typedef adjacency_list < vecS, vecS, directedS, no_property,
        property < edge_capacity_t, degree_size_type,
        property < edge_residual_capacity_t, degree_size_type,
        property < edge_reverse_t, edge_descriptor > > > > FlowGraph;
```

В листинге 8.3 мы используем std::set для множества S и множества соседей S (переменная neighbor_S), поскольку при вставке должна гарантироваться уникальность. Множество S^* (переменная S_star) и множество не соседей S (переменная nonneighbor_S) представлены std::vector в связи с тем, что мы знаем, что вставляемые элементы будут уникальными.

Листинг 8.3. Определение переменных

```
⟨ Определить переменные ⟩ ≡
vertex_descriptor u, v, p, k;
edge_descriptor e1, e2;
bool inserted;
typename graph_traits < VertexListGraph >::vertex_iterator vi, vi_end;
degree_size_type delta, alpha_star, alpha_S_k;
std::set < vertex_descriptor > S, neighbor_S;
std::vector < vertex_descriptor > S_star, nonneighbor_S;
std::vector < default_color_type > color(num_vertices(g));
std::vector < edge_descriptor > pred(num_vertices(g));
```

Граф потоковой сети создается на основе входного графа (листинг 8.4). Каждое ребро потокового графа имеет три свойства: пропускную способность, невязку и обратное ребро. Доступ к этим свойствам осуществляется через объекты-отображения свойств: cap, res_cap и rev_edge соответственно.

Листинг 8.4. Создание графа потоковой сети из неориентированного графа

```
⟨ Создать граф потоковой сети из неориентированного графа ⟩ ≡
FlowGraph flow_g(num_vertices(g));  // потоковый граф
typename property_map < FlowGraph, edge_capacity_t >::type
        cap = get(edge_capacity, flow_g);
typename property_map < FlowGraph, edge_residual_capacity_t >::type
        res_cap = get(edge_residual_capacity, flow_g);
typename property_map < FlowGraph, edge_reverse_t >::type
        rev_edge = get(edge_reverse, flow_g);

typename graph_traits < VertexListGraph >::edge_iterator ei, ei_end;
for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei) {
  u = source(*ei, g), v = target(*ei, g);
  tie(e1, inserted) = add_edge(u, v, flow_g);
  cap[e1] = 1;
  tie(e2, inserted) = add_edge(v, u, flow_g);
  cap[e2] = 1;
  rev_edge[e1] = e2;
  rev_edge[e2] = e1;
}
```

В главном алгоритме функциональные блоки выделены в отдельные функции (листинг 8.5). В первом блоке в цикле по всем вершинам графа находится вершина минимальной степени.

Листинг 8.5. Функция нахождения вершины минимальной степени

```
⟨ Функция нахождения вершины минимальной степени ⟩ ≡
template < typename Graph >
std::pair < typename graph_traits < Graph >::vertex_descriptor,
            typename graph_traits < Graph >::degree_size_type >
min_degree_vertex(Graph & g)
{
  typename graph_traits < Graph >::vertex_descriptor p;
  typedef typename graph_traits < Graph >::degree_size_type size_type;
  size_type delta = std::numeric_limits < size_type >::max();
  typename graph_traits < Graph >::vertex_iterator i, iend;
  for (tie(i, iend) = vertices(g); i != iend; ++i)
    if (degree(*i, g) < delta)
    {
      delta = degree(*i, g);
```

```
      p = *i;
    }
  return std::make_pair(p, delta);
}
```

Необходимо обеспечить возможность вставки каждого соседа вершины (и множества вершин) во множество. Это осуществляется через `adjacent_vertices()`. Мы предполагаем, что итератор вывода сходен с `std::insert_iterator` для `std::set`. Вспомогательные функции приведены в листинге 8.6.

Листинг 8.6. Вспомогательные функции вывода соседей

```
⟨ Вспомогательные функции вывода соседей ⟩ ≡
template < typename Graph, typename OutputIterator >
void neighbors(const Graph & g,
               typename graph_traits < Graph >::vertex_descriptor u,
               OutputIterator result)
{
  typename graph_traits < Graph >::adjacency_iterator ai, aend;
  for (tie(ai, aend) = adjacent_vertices(u, g); ai != aend; ++ai)
    *result++ = *ai;
}

template < typename Graph, typename VertexIterator,
           typename OutputIterator >
void neighbors(const Graph & g, VertexIterator first,VertexIterator last,
               OutputIterator result)
{
  for (; first != last; ++first)
    neighbors(g, *first, result);
}
```

На начальном шаге алгоритма (листинг 8.7) осуществляется поиск вершины минимальной степени p, затем $S = p$, а после этого вычисляются соседи и не соседи S. Мы используем `std::set_difference()` для вычисления $V - S$, где V — множество вершин графа.

Листинг 8.7. Поиск вершины и вычисление соседей и не соседей S

```
⟨ Найти вершину минимальной степени и вычислить соседей S и не соседей S ⟩ ≡
tie(p, delta) = min_degree_vertex(g);
S_star.push_back(p);
alpha_star = delta;
S.insert(p);
neighbor_S.insert(p);
neighbors(g, S.begin(), S.end(), std::inserter(neighbor_S,
          neighbor_S.begin()));
std::set_difference(vertices(g).first, vertices(g).second,
                    neighbor_S.begin(), neighbor_S.end(),
                    std::back_inserter(nonneighbor_S));
```

Итерации алгоритма заканчиваются, когда множество не соседей S становится пустым. В каждом шаге алгоритма (листинг 8.8) максимальный поток между p и не соседом k вычисляется с помощью алгоритма Эдмондса–Карпа (см. раздел 13.7.1). Вершины, окрашенные не в белый цвет, во время нахождения максимального потока соответствуют всем вершинам на одной стороне минимального разреза. Таким образом, если размер разреза на данный момент наименьший, окрашенные (не белые) вершины записываются в S^*, затем k добавляется в S, соседи и не соседи S вычисляются заново.

Листинг 8.8. Главный цикл

```
⟨ Главный цикл ⟩ ≡
while (!nonneighbor_S.empty()) {
  k = nonneighbor_S.front();
  alpha_S_k = edmunds_karp_max_flow
    (flow_g, p, k, cap, res_cap, rev_edge, &color[0], &pred[0]);
  if (alpha_S_k < alpha_star) {
    alpha_star = alpha_S_k;
    S_star.clear();
    for (tie(vi, vi_end) = vertices(flow_g); vi != vi_end; ++vi)
      if (color[*vi] != Color::white())
        S_star.push_back(*vi);
  }
  S.insert(k);
  neighbor_S.insert(k);
  neighbors(g, k, std::inserter(neighbor_S, neighbor_S.begin()));
  nonneighbor_S.clear();
  std::set_difference(vertices(g).first, vertices(g).second,
                      neighbor_S.begin(), neighbor_S.end(),
                      std::back_inserter(nonneighbor_S));
}
```

На завершающем этапе (листинг 8.9) осуществляется поиск ребер разреза, которые имеют одну вершину в S^*, а другую в $\overline{S^*}$. Эти ребра записываются через итератор вывода disconnecting_set (несвязное множество), а количество ребер в разрезе возвращается оператором return.

Листинг 8.9. Вычисление прямых ребер разреза

```
⟨ Вычислить прямые ребра разреза ⟩ ≡
std::vector < bool > in_S_star(num_vertices(g), false);
typename std::vector < vertex_descriptor >::iterator si;
for (si = S_star.begin(); si != S_star.end(); ++si)
  in_S_star[*si] = true;
degree_size_type c = 0;
for (si = S_star.begin(); si != S_star.end(); ++si) {
  typename graph_traits <VertexListGraph>::out_edge_iterator ei, ei_end;
  for (tie(ei, ei_end) = out_edges(*si, g); ei != ei_end; ++ei)
    if (!in_S_star[target(*ei, g)]) {
      *disconnecting_set++ = *ei;
      ++c;
    }
}
```

Файл edge-connectivity.cpp, в котором реализован алгоритм нахождения реберной связности, приведен в листинге 8.10.

Листинг 8.10. Файл edge-connectivity.cpp

```
⟨ edge-connectivity.cpp ⟩ ≡
#include <algorithm>
#include <utility>
#include <boost/graph/edmunds_karp_max_flow.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graphviz.hpp>

namespace boost {
  ⟨ Функция нахождения вершины минимальной степени ⟩
```

```
  ⟨ Вспомогательные функции вывода соседей ⟩
  ⟨ Алгоритм вычисления реберной связности ⟩
}

int main() {
  using namespace boost;
  GraphvizGraph g;
  read_graphviz("figs/edge-connectivity.dot", g);

  typedef graph_traits < GraphvizGraph >::edge_descriptor edge_descriptor;
  typedef graph_traits < GraphvizGraph >::degree_size_type degree_size_type;
  std::vector < edge_descriptor > disconnecting_set;
  degree_size_type c =
    edge_connectivity(g, std::back_inserter(disconnecting_set));

  std::cout << "Реберная связность: " << c << "." << std::endl;

  property_map < GraphvizGraph, vertex_attribute_t >::type
    attr_map = get(vertex_attribute, g);

  std::cout << "Сечение: {";
  for (std::vector < edge_descriptor >::iterator i =
    disconnecting_set.begin(); i != disconnecting_set.end(); ++i)
    std::cout << "(" << attr_map[source(*i, g)]["label"] << ","
              << attr_map[target(*i, g)]["label"] << ") ";
  std::cout << "}." << std::endl;
  return EXIT_SUCCESS;
}
```

В результате работы файла edge-connectivity.cpp выводится следующее:

```
Реберная связность: 2.
Сечение: { (D,E) (D,H) }.
```

Граф, к которому был применен алгоритм нахождения реберной связности (листинг 8.10), представлен на рис. 8.1.

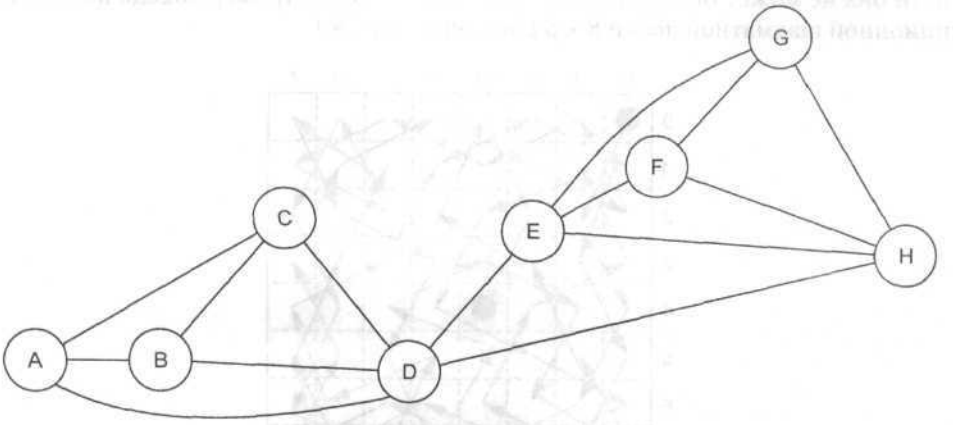


Рис. 8.1. Пример графа для реберной связности

Неявные графы: обход конем

9

Задача обхода конем состоит в нахождении такого пути для коня, чтобы он побывал на каждой клетке шахматной доски $n \times n$ ровно один раз. Обход конем является примером *гамильтонового пути*. Гамильтонов путь — это простой замкнутый путь, который проходит через каждую вершину графа только один раз. При обходе конем клетка шахматной доски считается вершиной графа. Ребра графа определяются в соответствии с шахматными правилами для хода коня (например, на две клетки вверх и на одну в сторону). В этом разделе мы используем для нахождения пути коня обобщенный алгоритм поиска с возвратом. Алгоритм с возвратом является алгоритмом грубой силы и достаточно медлителен, поэтому мы также покажем, как улучшить алгоритм, применив эвристику Варнсдорфа [46]. Задача нахождения гамильтонова цикла является NP-полной [15] (при большой размерности она не может быть решена за разумное время). Пример обхода конем традиционной шахматной доски 8×8 показан на рис. 9.1.

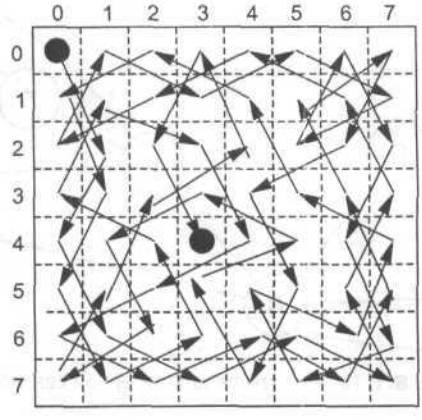


Рис. 9.1. Пример обхода конем шахматной доски

Одной из особенностей этого примера является отсутствие явной структуры (вроде класса adjacency_list) для представления графа. Вместо этого неявная структура графа knights_tour_graph следует из допустимых ходов коня на доске.

9.1. Ходы конем как граф

Граф knights_tour_graph является моделью AdjacencyGraph, поэтому нам нужно реализовать функцию adjacent_vertices(), возвращающую пару итераторов смежности. Итератор смежности трактует каждую из клеток, в которую может перейти конь, вершиной, смежной данной.

Набор возможных ходов коня хранится в массиве следующим образом:

```
typedef std::pair < int, int > Position;
Position knight_jumps[8] = { Position(2, -1), Position(1, -2),
  Position(-1, -2), Position(-2, -1), Position(-2, 1),
  Position(-1, 2), Position(1, 2), Position(2, 1) };
```

Итератор knight_adjacency_iterator содержит несколько полей данных-членов: текущую позицию на доске m_pos, индекс m_i в массиве ходов knight_jumps и указатель на граф m_g. Инкремент итератора смежности (операция operator++()) увеличивает m_i. Новая позиция может быть некорректной (за пределами доски), поэтому может понадобиться еще раз увеличить m_i, что и делает функция-метод класса valid_position(). Первая позиция тоже может быть некорректна, поэтому valid_position() вызывается также и в конструкторе итератора смежности. Указатель на шахматную доску необходим для того, чтобы можно было определить ее размер (так как доска может быть произвольного размера). При разыменовании итератора смежности (operator*()) возвращается смещение текущей позиции в текущем векторе ходов. Реализация итератора knight_adjacency_iterator приведена в листинге 9.1. Вспомогательная функция boost::forward_iterator_helper используется для автоматической реализации операции operator++(int) посредством operator++() и операции operator!=() посредством operator==().

Листинг 9.1. Итератор смежности knight_adjacency_iterator

```
struct knight_adjacency_iterator:
  public boost::forward_iterator_helper < knight_adjacency_iterator,
    Position, std::ptrdiff_t, Position *, Position > {
  knight_adjacency_iterator() { }
  knight_adjacency_iterator(int ii, Position p,
    const knights_tour_graph & g) : m_pos(p), m_g(&g), m_i(ii) {
  valid_position(); }
  Position operator *() const { return m_pos + knight_jumps[m_i]; }
  void operator++ () { ++m_i; valid_position(); }
  bool operator == (const knight_adjacency_iterator & x) const {
    return m_i == x.m_i; }
protected:
  void valid_position();
  Position m_pos;
  const knights_tour_graph * m_g;
  int m_i;
};
```

В функции-методе класса valid_position() счетчик ходов увеличивается до тех пор, пока не будет найдена позиция на доске или не будет достигнут конец массива.

```
void knight_adjacency_iterator::valid_position() {
  Position new_pos = m_pos + knight_jumps[m_i];
  while (m_i < 8 && (new_pos.first < 0 || new_pos.second < 0
                     || new_pos.first >= m_g->m_board_size
                     || new_pos.second >= m_g->m_board_size)) {
    ++m_i;
    new_pos = m_pos + knight_jumps[m_i];
  }
}
```

Функция adjacent_vertices() создает пару итераторов смежности, используя 0 для начальной позиции итератора ходов и 8 для конечной позиции итератора ходов.

```
std::pair < knights_tour_graph::adjacency_iterator,
  knights_tour_graph::adjacency_iterator >
adjacent_vertices(knights_tour_graph::vertex_descriptor v,
                  const knights_tour_graph & g) {
  typedef knights_tour_graph::adjacency_iterator Iter;
  return std::make_pair(Iter(0, v, g), Iter(8, v, g));
}
```

Класс knights_tour_graph (листинг 9.2) содержит только размер доски (как поле) и операторы typedef, требуемые для AdjacencyGraph. Функция num_vertices() возвращает число клеток на доске.

Листинг 9.2. Класс knights_tour_graph

```
struct knights_tour_graph
{
  typedef Position vertex_descriptor;
  typedef std::pair < vertex_descriptor, vertex_descriptor >
    edge_descriptor;
  typedef knight_adjacency_iterator adjacency_iterator;
  typedef void out_edge_iterator;
  typedef void in_edge_iterator;
  typedef void edge_iterator;
  typedef void vertex_iterator;
  typedef int degree_size_type;
  typedef int vertices_size_type;
  typedef int edges_size_type;
  typedef directed_tag directed_category;
  typedef disallow_parallel_edge_tag edge_parallel_category;
  typedef adjacency_graph_tag traversal_category;
  knights_tour_graph(int n): m_board_size(n) { }
  int m_board_size;
};
int num_vertices(const knights_tour_graph & g) {
  return g.m_board_size * g.m_board_size;
}
```

Теперь, когда ходы конем отображены в графовом интерфейсе Boost, мы можем рассмотреть отдельные графовые алгоритмы, которые могут быть использованы для решения задачи обхода конем шахматной доски.

9.2. Поиск с возвратом на графе

Идея алгоритма *поиска с возвратом на графе* (backtracking graph search) подобна поиску в глубину в том, что путь исследуется до тех пор, пока не будет обнаружен тупик. Поиск с возвратом отличается тем, что после достижения тупика алгоритм возвращается обратно, снимая пометки с тупикового пути, перед тем как продолжить поиск по другому пути. В листинге 9.3 поиск с возвратом реализован с помощью стека (вместо рекурсии), а порядок посещения каждой вершины записывается с помощью отображения свойства. Стек состоит из пар отметка времени — вершина, так что правильная отметка посещения доступна после возвращения из тупика. Поиск завершается, когда все вершины посещены или все возможные пути исчерпаны.

Хотя граф, определенный в предыдущем разделе, был неявным и представлял, в частности, обход шахматной доски конем, он тем не менее моделирует концепцию Graph из BGL, то есть алгоритм поиска с возвратом реализован для концепции Graph, а не для конкретного графа обхода конем. Получившийся в результате алгоритм можно повторно использовать для любой структуры данных, моделирующей Graph.

Листинг 9.3. Алгоритм поиска с возвратом

```
template < typename Graph, typename TimePropertyMap >
bool backtracking_search(Graph & g,
        typename graph_traits < Graph >::vertex_descriptor src,
        TimePropertyMap time_map)
{
  < Создать стек и инициализировать отметку времени >
  S.push(std::make_pair(time_stamp, src));
  while (!S.empty()) { // цикл пока стек не пуст
    < Получить вершину со стека, записать время и проверить завершение >
    < Положить все смежные вершины на стек >
    < Если в тупике, откатиться >
  } // while (!S.empty())
  return false;
}
```

Для записи вершин, которые нужно проверить, используется std::stack. Вершина заносится в стек вместе с отметкой посещения.

```
< Создать стек и инициализировать отметку времени > ≡
typedef typename graph_traits < Graph >::vertex_descriptor Vertex;
typedef std::pair < int, Vertex > P;
std::stack < P > S;
int time_stamp = 0;
```

Следующий шаг — записать отметку посещения для вершины графа наверху стека и проверить, не занесены ли уже в стек все вершины графа. В последнем случае алгоритм успешно завершается.

```
< Получить вершину со стека, записать время и проверить завершение > ≡
Vertex x;
tie(time_stamp, x) = S.top();
put(time_map, x, time_stamp);
// все вершины посещены, успех!
if (time_stamp == num_vertices(g) - 1)
  return true;
```

Теперь просматриваются все смежные вершины, и если смежная вершина еще не была посещена, она кладется в стек. Отсутствие смежных вершин — признак тупика.

```
⟨ Положить все смежные вершины на стек ⟩ ≡
bool deadend = true;  // переменная для индикации тупика
typename graph_traits < Graph >::adjacency_iterator i, end;
for (tie(i, end) = adjacent_vertices(x, g); i != end; ++i)
  if (get(time_map, *i) == -1) {
    S.push(std::make_pair(time_stamp + 1, *i));
    deadend = false;
  }
```

Если алгоритм достигает тупика, вершины выталкиваются из стека, пока не будет найдена еще не исследованная вершина. В процессе возврата отметки посещения для каждой вершины сбрасываются, поэтому, возможно, эти же вершины будут достигнуты по более подходящему пути.

```
⟨ Если в тупике, откатиться ⟩ ≡
if (deadend) {  // если тупик
  put(time_map, x, -1);
  S.pop();
  tie(time_stamp, x) = S.top();
  while (get(time_map, x) != -1) {
    // откатить стек до последней не исследованной вершины
    put(time_map, x, -1);
    S.pop();
    tie(time_stamp, x) = S.top();
  }
}
```

9.3. Эвристика Варнсдорфа

Эвристика Варнсдорфа для выбора следующей клетки для хода заключается в упреждающем просмотре каждого из возможных ходов для определения того, сколько дальнейших ходов возможны из этой клетки. Назовем это *числом последователей*. Клетка с наименьшим числом последователей выбирается для следующего хода. Причина, по которой эта эвристика работает, в том, что, посещая в начале наиболее ограниченные вершины, мы избегаем потенциальных тупиков. Следующая функция `number_of_successors()` в листинге 9.4 вычисляет число последователей вершины.

Листинг 9.4. Функция number_of_successors()

```
template < typename Vertex, typename Graph, typename TimePropertyMap >
int number_of_successors(Vertex x, Graph & g, TimePropertyMap time_map)
{
  int s_x = 0;
  typename graph_traits < Graph >::adjacency_iterator i, end;
  for (tie(i, end) = adjacent_vertices(x, g); i != end; ++i)
    if (get(time_map, *i) == -1)
      ++s_x;
  return s_x;
}
```

Реализация эвристики Варнсдорфа (листинг 9.5) начинается с алгоритма с возвратом, но вместо заталкивания смежных вершин в стек сначала происходит сортировка вершин по количеству последователей. Сортировка выполняется при помещении смежных вершин в очередь по приоритету (priority queue). После занесения вершин в очередь они извлекаются оттуда и помещаются в стек. Пустая очередь сигнализирует о тупике.

Листинг 9.5. Реализация эвристики Варнсдорфа

```
template < typename Graph, typename TimePropertyMap >
  bool warnsdorff(Graph & g,
                  typename graph_traits < Graph >::vertex_descriptor src,
                  TimePropertyMap time_map)
{
  ( Создать стек и инициализировать отметку времени )
  S.push(std::make_pair(time_stamp, src));
  while (!S.empty()) {
    ( Получить вершину со стека, записать время и проверить завершение )
    //  поместить смежные вершины в локальную очередь по приоритету
    std::priority_queue < P, std::vector < P >, compare_first > Q;
    typename graph_traits < Graph >::adjacency_iterator i, end;
    int num_succ;
    for (tie(i, end) = adjacent_vertices(x, g); i != end; ++i)
      if (get(time_map, *i) == -1) {
        num_succ = number_of_successors(*i, g, time_map);
        Q.push(std::make_pair(num_succ, *i));
      }
    bool deadend = Q.empty();
    //  переместить вершины из локальной очереди в стек
    for (; !Q.empty(); Q.pop()) {
      tie(num_succ, x) = Q.top();
      S.push(std::make_pair(time_stamp + 1, x));
    }
    if (deadend) {
      put(time_map, x, -1);
      S.pop();
      tie(time_stamp, x) = S.top();
      while (get(time_map, x) != -1) {
        //  откатить стек до последней не исследованной вершины
        put(time_map, x, -1);
        S.pop();
        tie(time_stamp, x) = S.top();
      }
    }
  } //  while (!S.empty())
  return false;
}
```

Взаимодействие с другими графовыми библиотеками

10

Хотя основной задачей Boost Graph Library является поддержка разработки новых приложений и графовых алгоритмов, существует довольно много кодов, которые могут получить выгоду от использования BGL. Одним из путей использования алгоритмов BGL с существующими графовыми структурами данных является копирование данных из старого формата в BGL-граф, который затем используется в алгоритмах BGL. Проблема с этим подходом заключается в том, что не всегда удобно и эффективно выполнять это копирование. Другим подходом является использование существующих структур непосредственно, используя «обертку» для BGL-интерфейса.

Паттерн Адаптер [14] — это один из механизмов для обеспечения нового интерфейса для существующего класса. Этот подход обычно требует, чтобы адаптируемый объект хранился внутри нового класса, предоставляющего желаемый интерфейс. При адаптации BGL-графа вложенности объекта не требуется, поскольку графовый интерфейс BGL полностью состоит из независимых (глобальных) функций. Такой интерфейс вместо создания нового графового класса требует только перегрузки независимых функций, из которых он состоит. В разделе 10.3 мы покажем, как это делается, во всех деталях.

Библиотека BGL включает перегруженные функции для типа `GRAPH` из LEDA [29], типа `Graph*` из Stanford GraphBase, а также для `std::vector` из STL. LEDA — популярная объектно-ориентированная библиотека для комбинаторных вычислений, включающая графовые алгоритмы и структуры данных. Stanford GraphBase, написанная Дональдом Кнутом, является набором графовых данных, генераторов графов и программ, которые выполняют алгоритмы на этих графах.

В следующих разделах мы покажем примеры использования структур данных из LEDA и SGB с алгоритмами BGL. Затем мы рассмотрим реализацию адаптирующих функций из BGL для графов LEDA, приводя пример того, как нужно писать адаптеры для других графовых библиотек.

В разделе 1.4.1 мы демонстрировали гибкость алгоритмов BGL, применяя `topological_sort()` к графам, представленным вектором списков, к графам в виде `boost::adjacency_list` и `std::vector<std::list<int>>`. Мы продолжим этот пример — планирование выполнения взаимозависимых заданий — в следующих двух

разделах, сначала используя LEDA-тип GRAPH, а затем — тип Graph от Stanford GraphBase (SGB).

10.1. Использование топологической сортировки из BGL с графом из LEDA

Заголовочный файл boost/graph/leda_graph.hpp содержит перегруженные функции, которые адаптируют параметризованный тип GRAPH из LEDA к интерфейсу BGL. BGL-интерфейс к LEDA GRAPH описан в разделе 14.3.5. Интерфейс между LEDA и BGL был протестирован с версией LEDA 4.1, одной из свободно распространяемых версий LEDA. В дополнение к leda_graph.hpp библиотека LEDA должна быть установлена, настроены пути к заголовочным и библиотечным файлам, а LEDA-библиотеки использованы при сборке программы. Более детальный материал находится в документации LEDA.

В листинге 10.1 приведена схема программы планирования заданий, на этот раз используя GRAPH из LEDA для представления зависимостей между заданиями.

Листинг 10.1. Схема программы планирования заданий

```
⟨ topo-sort-with-leda.cpp ⟩ ≡
#include <vector>
#include <string>
#include <boost/graph/topological_sort.hpp>
#include <boost/graph/leda_graph.hpp>
// Отменить определение макросов LEDA, конфликтующих с C++ Standard Library
#undef string
#undef vector

int main() {
  using namespace boost;
  ⟨ Создать граф LEDA с вершинами, обозначенными заданиями ⟩
  ⟨ Добавить ребра к графу LEDA ⟩
  ⟨ Выполнить топологическую сортировку графа LEDA ⟩
  return EXIT_SUCCESS;
}
```

Класс GRAPH из LEDA позволяет пользователю присоединить объекты-свойства к вершинам и ребрам графа, поэтому здесь мы прикрепляем имена (в форме std::string) к вершинам. Мы используем обычную функцию add_vertex() для добавления вершин к leda_g и передаем имена заданий как объект-свойство для присоединения к вершине. Дескрипторы вершин, возвращенные из add_vertex(), хранятся в векторе, так что можно быстро использовать нужную вершину при добавлении ребер. Код для создания графа LEDA приведен в листинге 10.2.

Листинг 10.2. Создание графа LEDA для задачи планирования заданий

```
⟨ Создать граф LEDA с вершинами, обозначенными заданиями ⟩ ≡
typedef GRAPH < std::string, char >graph_t;
graph_t leda_g;
typedef graph_traits < graph_t >::vertex_descriptor vertex_t;
std::vector < vertex_t > vert(7);
vert[0] = add_vertex(std::string("забрать детей из школы"), leda_g);
vert[1] = add_vertex(std::string("купить продукты"), leda_g);
vert[2] = add_vertex(std::string("получить деньги в банкомате"), leda_g);
vert[3] = add_vertex(std::string("привести детей на тренировку"), leda_g);
vert[4] = add_vertex(std::string("приготовить ужин"), leda_g);
```

продолжение ⇩

Листинг 10.2 (*продолжение*)

```
vert[5] = add_vertex(std::string("забрать детей с тренировки"), leda_g);
vert[6] = add_vertex(std::string("съесть ужин"), leda_g);
```

Следующий шаг — добавление ребер к графу. Вновь мы используем обычную функцию `add_edge()`.

```
⟨ Добавить ребра к графу LEDA ⟩ ≡
add_edge(vert[0], vert[3], leda_g);
add_edge(vert[1], vert[3], leda_g);
add_edge(vert[1], vert[4], leda_g);
add_edge(vert[2], vert[1], leda_g);
add_edge(vert[3], vert[5], leda_g);
add_edge(vert[4], vert[6], leda_g);
add_edge(vert[5], vert[6], leda_g);
```

Теперь, когда граф построен, можно вызвать `topological_sort()`. Благодаря интерфейсу между LEDA и BGL `GRAPH` из LEDA может быть использован в неизменном виде с BGL-функцией. Мы просто передаем объект `leda_g` в этот алгоритм. Функция `topological_sort()` требует отображения свойства окраски вершин, поэтому мы используем массив вершин LEDA `node_array` для отображения вершин в цвета. Функция `leda_node_property_map()` также определена в `boost/graph/leda_graph.hpp` и создает адаптер, который удовлетворяет концепции `LvaluePropertyMap` в контексте `node_array`. В вектор `topo_order` записывается обратное топологическое упорядочение. Затем вектор переворачивается, и выводится упорядочение. Операция `operator[]()` класса `GRAPH` из LEDA используется для доступа к названию задания для каждой вершины. Код топологической сортировки приведен в листинге 10.3.

Листинг 10.3. Выполнение топологической сортировки

```
⟨ Выполнить топологическую сортировку графа LEDA ⟩ ≡
std::vector < vertex_t > topo_order;
node_array < default_color_type > color_array(leda_g);

topological_sort(leda_g, std::back_inserter(topo_order),
                 color_map(make_leda_node_property_map(color_array)));

std::reverse(topo_order.begin(), topo_order.end());
int n = 1;
for (std::vector < vertex_t >::iterator i = topo_order.begin();
     i != topo_order.end(); ++i, ++n)
  std::cout << n << ": " << leda_g[*i] << std::endl;
```

10.2. Использование топологической сортировки из BGL с графом из SGB

Библиотека Stanford GraphBase определяет структуру `Graph`, которая реализует структуру данных в стиле списка смежности. Перегруженные функции в `boost/graph/stanford_graph.hpp` адаптируют структуру `Graph` к интерфейсу BGL. Помимо включения заголовочного файла `stanford_graph.hpp` необходимо, чтобы была установлена библиотека SGB и применен файл изменения `PROTOTYPES` (из дистрибутива SGB). Это необходимо потому, что оригинальные заголовочные файлы SGB не определяют прототипы функций согласно стандарту ANSI, что требуется для компилятора C++. При компиляции программы с использованием SGB-BGL интерфейса необходимо установить пути к заголовочным файлам и библиотекам для

SGB, нужно также собрать программу с библиотекой SGB. Интерфейс между BGL и SGB задокументирован в разделе 14.3.4. Пример топологической сортировки для графа из SGB приведен в листинге 10.4.

Листинг 10.4. Топологическая сортировка графа из SGB

```
⟨ topo-sort-with-sgb.cpp ⟩ ≡
#include <vector>
#include <string>
#include <iostream>
#include <boost/graph/topological_sort.hpp>
#include <boost/graph/stanford_graph.hpp>

int main() {
  using namespace boost;
  ⟨ Создать SGB-граф ⟩
  ⟨ Создать метки для заданий ⟩
  ⟨ Добавить ребра к SGB-графу ⟩
  ⟨ Выполнить топологическую сортировку на SGB-графе ⟩
  gb_recycle(sgb_g);
  return EXIT_SUCCESS;
}
```

Мы создаем SGB-граф вызовом SGB-функции `gb_new_graph()`.

```
⟨ Создать SGB-граф ⟩ ≡
const int n_vertices = 7;
Graph *sgb_g = gb_new_graph(n_vertices);
```

Далее мы записываем метки для заданий (вершины) в граф. От дескриптора вершины в SGB легко перейти к целому числу, используя `sgb_vertex_id_map`, определенный в **stanford_graph.hpp**, поэтому хранение меток в массиве удобно.

```
⟨ Создать метки для заданий ⟩ ≡
const char *tasks[] = {
  "забрать детей из школы",
  "купить продукты",
  "получить деньги в банкомате",
  "привести детей на тренировку",
  "приготовить ужин",
  "забрать детей с тренировки",
  "съесть ужин"
};
const int n_tasks = sizeof(tasks) / sizeof(char *);
```

SGB-граф хранит вершины графа в массиве, поэтому можно получить доступ к любой вершине по индексу в массиве. Функция `gb_ne_edge()` принимает два аргумента `Vertex*` и вес ребра (нам он не важен).

```
⟨ Добавить ребра к SGB-графу ⟩ ≡
gb_new_arc(sgb_g->vertices + 0, sgb_g->vertices + 3, 0);
gb_new_arc(sgb_g->vertices + 1, sgb_g->vertices + 3, 0);
gb_new_arc(sgb_g->vertices + 1, sgb_g->vertices + 4, 0);
gb_new_arc(sgb_g->vertices + 2, sgb_g->vertices + 1, 0);
gb_new_arc(sgb_g->vertices + 3, sgb_g->vertices + 5, 0);
gb_new_arc(sgb_g->vertices + 4, sgb_g->vertices + 6, 0);
gb_new_arc(sgb_g->vertices + 5, sgb_g->vertices + 6, 0);
```

Затем осуществляется топологическая сортировка. Мы передаем в алгоритм сам SGB-граф. На этот раз вместо явного создания отображения цветов мы позволим алгоритму самому создать его. Однако для этого функция `topological_sort()` нуждается в отображении вершин в целые числа. Интерфейс между SGB и BGL обеспечивает такое отображение свойства. Отображение вершин в индексы можно

получить вызовом get(vertex_index, sgb_g). Код топологической сортировки приведен в листинге 10.5.

Листинг 10.5. Вычисление топологической сортировки SGB-графа

```
⟨ Выполнить топологическую сортировку на SGB-графе ⟩ ≡
typedef graph_traits < Graph * >::vertex_descriptor vertex_t;
std::vector < vertex_t > topo_order;
topological_sort(sgb_g, std::back_inserter(topo_order),
  vertex_index_map(get(vertex_index, sgb_g)));
int n = 1;
for (std::vector < vertex_t >::reverse_iterator i = topo_order.rbegin();
     i != topo_order.rend(); ++i, ++n)
  std::cout << n << ": " << tasks[get(vertex_index, sgb_g)[*i]]
  << std::endl;
```

10.3. Реализация адаптеров графов

Написать адаптер для других графовых библиотек и структур данных совсем не сложно. В качестве примера создания новых адаптеров этот раздел предлагает детальное объяснение реализации интерфейса BGL для графа из LEDA.

Первое, с чем нужно определиться, — это какие из концепций будет реализовывать BGL граф. Следующие концепции легко реализовать поверх классов из LEDA: VertexListGraph, BidirectionalGraph, VertexMutableGraph и EdgeMutableGraph.

Все типы, ассоциированные с графовым классом из BGL, можно получить с помощью класса graph_traits. Этот класс свойств может быть частично специализирован для графового класса GRAPH из LEDA[1] (листинг 10.6). Типы node и edge являются эквивалентами дескрипторов вершин и ребер. Класс GRAPH предназначен для ориентированных графов, поэтому мы выбираем тег directed_tag для directed_category. Так как класс GRAPH автоматически не запрещает вставку параллельных ребер, в нашем случае устанавливается allow_parallel_edge_tag для edge_parallel_category. Функция number_of_nodes() из LEDA возвращает целое число, поэтому данный тип указан для vertices_size_type. Тип тега, используемый для traversal_category, должен отражать моделируемые графом концепции обхода, поэтому мы создаем теговый класс, наследующий от bidirectional_graph_tag, adjacency_graph_tag и vertex_list_graph_tag. Типы итераторов описаны далее в этом разделе.

Листинг 10.6. Свойства графов в LEDA

```
⟨ Свойства графов для LEDA-графа ⟩ ≡
namespace boost {
  struct leda_graph_traversal_category :
    public virtual bidirectional_graph_tag,
    public virtual adjacency_graph_tag,
    public virtual vertex_list_graph_tag { };

  template < typename V, typename E>
  struct graph_traits< GRAPH<V,E> > {
    typedef node vertex_descriptor;
    typedef edge edge_descriptor;
    typedef directed_tag directed_category;
```

[1] Некоторые нестандартные компиляторы, например Visual C++ 6.0, не поддерживают частичную специализацию. Для доступа к ассоциированным типам в этом случае класс свойств должен быть полностью специализирован для конкретных типов вершин и ребер. В альтернативном варианте может быть применен класс-оболочка, содержащий LEDA-граф и требуемые вложенные определения типов.

```
    typedef allow_parallel_edge_tag edge_parallel_category;
    typedef leda_graph_traversal_category traversal_category;
    typedef int vertices_size_type;
    typedef int edges_size_type;
    typedef int degree_size_type;

    ⟨ Тип итератора исходящих вершин ⟩
    // другие typedef для итераторов ...
  };
} // namespace boost
```

Сначала мы напишем функции source() и target() для концепции IncidenceGraph, которая является частью концепции BidirectionalGraph. Мы используем тип GRAPH из LEDA в качестве параметра для графа и graph_traits для задания параметра ребра и возвращаемого типа для вершины. Хотя типы LEDA могут быть применены для вершины и ребра, на практике лучше использовать graph_traits. Тогда, если придется изменить ассоциированный тип вершины или ребра, это можно сделать в одном месте внутри специализации graph_traits, а не во всем коде программы. Так как LEDA предоставляет функции source() и target(), мы просто их вызываем.

```
⟨ Получение начальной и конечной вершины ребра для графа из LEDA ⟩ ≡
template <class vtype, class etype>
typename graph_traits< GRAPH<vtype,etype> >::vertex_descriptor
source(typename graph_traits< GRAPH<vtype,etype> >::edge_descriptor e,
       const GRAPH<vtype,etype>& g)
{
  return source(e);
}
// по аналогии для конечной вершины
```

Следующая функция из IncidenceGraph — out_edges(). Эта функция возвращает пару итераторов по исходящим ребрам. Поскольку в LEDA применяются итераторы в стиле STL, их нужно реализовать. Написание итераторов, которые совместимы со стандартом C++, может быть сложным процессом. К счастью, в Boost имеется удобная утилита для реализации итераторов — класс iterator_adaptor. Этот класс позволяет пользователю создавать совместимые со стандартами итераторы, просто предоставляя классы правил (policy class). В листинге 10.7 приведен класс правил для итератора исходящих ребер. В LEDA сам объект-ребро используется как итератор. Он имеет функции Succ_Adj_Edge() и Pred_Adj_Edge() для перемещения к следующему или предыдущему ребру.

Листинг 10.7. Правила для итератора исходящих ребер

```
⟨ Правила для итератора исходящих ребер ⟩ ≡
struct leda_out_edge_iterator_policies
{
  static void initialize(leda_edge& ) { }

  template <typename Iter>
  static void increment(Iter& i)
  { i.base() = Succ_Adj_Edge(i.base(), 0); }

  template <typename Iter>
  static void decrement(Iter& i)
  { i.base() = Pred_Adj_Edge(i.base(), 0); }

  template <typename Iter>
  static leda_edge dereference(const Iter& i)
{ return i.base(); }
```

продолжение ⇩

Листинг 10.7 (*продолжение*)

```
  template <typename Iter>
  static bool equal(const Iter& x, const Iter& y)
  { return x.base() == y.base(); }
};
```

Теперь iterator_adaptor используется в качестве типа edge_iterator. Первые два параметра шаблона для edge_iterator — адаптируемый класс и класс правил. Следующие параметры указывают ассоциированные типы итератора, такие как тип значения и тип ссылки.

```
⟨ Тип итератора исходящих вершин ⟩ ≡
typedef iterator_adaptor<leda_edge, leda_out_edge_iterator_policies,
  leda_edge, const leda_edge&, const leda_edge*,
  std::forward_iterator_tag, std::ptrdiff_t
  > out_edge_iterator;
```

После определения итератора исходящих ребер в классе свойств можно определить функцию out_edges(). В следующем определении (листинг 10.8) возвращаемое значение должно быть парой итераторов исходящих вершин, так что мы используем std::pair и затем graph_traits для доступа к итераторным типам. В теле функции мы создаем итераторы исходящих ребер, передавая первое ребро для первого итератора и ноль — для второго (в LEDA с помощью нуля обозначается окончание последовательности). Ноль в качестве аргумента к First_Adj_Edge() говорит LEDA, что мы хотим обрабатывать исходящие ребра, а не входящие.

Листинг 10.8. Функция out_edges() для LEDA

```
⟨ Функция out_edges() для LEDA ⟩ ≡
template <typename V, typename E>
std::pair<typename graph_traits< GRAPH<V,E> >::out_edge_iterator,
          typename graph_traits< GRAPH<V,E> >::out_edge_iterator >
out_edges(typename graph_traits< GRAPH<V,E> >::vertex_descriptor u,
          const GRAPH<V,E>& g)
{
  typedef typename graph_traits< GRAPH<V,E> >::out_edge_iterator Iter;
  return std::make_pair( Iter(First_Adj_Edge(u,0)), Iter(0) );
}
```

Остальные типы итераторов и интерфейсных функций пишутся аналогично. Полный код для интерфейса оболочки к LEDA находится в boost/graph/leda_graph.hpp. В листинге 10.9 мы используем проверку концепций BGL, чтобы быть уверенными в правильности реализации интерфейса с BGL. Эти проверки не тестируют поведение времени исполнения (это тестируется в test/graph.cpp).

Листинг 10.9. Проверка концепции реализованного интерфейса

```
⟨ leda-concept-check.cpp ⟩ ≡
#include <boost/graph/graph_concepts.hpp>
#include <boost/graph/leda_graph.hpp>

int main()
{
  using namespace boost;
  typedef GRAPH<int, int> Graph;
  function_requires < VertexListGraphConcept<Graph> >();
  function_requires < BidirectionalGraphConcept<Graph> >();
  function_requires < VertexMutableGraphConcept<Graph> >();
  function_requires < EdgeMutableGraphConcept<Graph> >();
  return EXIT_SUCCESS;
}
```

Руководство по производительности 11

В этой главе мы обсудим влияние того или иного выбранного графа из семейства графов BGL adjacency_list на производительность. Цель этой главы — дать пользователям BGL некоторые базовые сведения о том, какие из типов графов могут быть наиболее эффективны в различных ситуациях. Мы представим серию тестов, отображающих производительность различных базовых операций нескольких разновидностей adjacency_list из BGL. Исследованы скорости выполнения операций над разреженными и плотными графами с использованием двух различных компиляторов (Microsoft Visual C++ и GNU C++).

Как основной компонент BGL-графа, adjacency_list помогает пользователям контролировать фактические структуры данных, применяемые для внутренних структур графа. Первые два параметра шаблона, EdgeList и VertexList, используются для выбора фактических контейнеров для представления последовательностей исходящих ребер и вершин соответственно. Пользователи могут применять vecS, listS или setS для EdgeList для выбора контейнеров std::vector, std::list или std::set соответственно. Они также могут указать vecS или listS для выбора std::vector или std::list соответственно в качестве основы.

11.1. Сравнения графовых классов

Мы сравнивали производительность различных вариантов adjacency_list. Эксперименты покрывают большинство базовых операций над графами: вставка и удаление вершин и ребер, обход графа по вершинам, ребрам и исходящим ребрам каждой вершины. Тесты были выполнены с разреженными и плотными графами малого (100 вершин), среднего (1000 вершин) и большого (10 000 вершин) размера. Для разреженного графа число ребер в 10 раз больше числа

вершин. Для плотного графа полное число ребер равно квадрату числа вершин.

Замеры времени проводились на компьютере Dell с двумя процессорами по 733 МГц, с памятью 512 Мбайт. Тесты были продублированы для двух компиляторов: Microsoft Visual C++ 6.0 и GNU C++ 2.95.3 (под cygwin). Для Visual C++ был установлен режим оптимизации по скорости. Для GNU C++ были установлены опции `-O3` и `-funroll-loops`. Заметим, что реализация `adjacency_list` использует компоненты из STL, которая поставляется вместе с компилятором.

В таймере применялась переносимая POSIX-функция `clock()`, которая имеет довольно низкое разрешение. По этой причине эксперименты выполнялись в цикле до тех пор, пока затраченное время не превысило минимальное разрешение по крайней мере в сто раз. Каждый тест был повторен 3 раза, и взято минимальное время из этих трех прогонов. Мы заметили, что стандартное отклонение в измерениях времени составляет приблизительно 10 %.

Далее приведен полный набор графовых типов, использованных в тестах. Также указаны аббревиатуры, которые были применены в результирующих диаграммах.

- vec

```
adjacency_list<vecS, vecS, directedS, property<vertex_distance_t, int>,
               property<edge_weight_t, int> >
```

- list

```
adjacency_list<listS, vecS, directedS, property<vertex_distance_t, int>,
               property<edge_weight_t, int> >
```

- set

```
adjacency_list<setS, vecS, directedS, property<vertex_distance_t, int>,
               property<edge_weight_t, int> >
```

- listlist

```
adjacency_list<listS, listS, directedS, property<vertex_distance_t, int>,
               property<edge_weight_t, int> >
```

11.1.1. Результаты и обсуждение

Добавление ребер и вершин

В первом эксперименте чередуются вызовы в `add_vertex()` и `add_edge()`, пока число ребер графа не достигнет $|E|$ ребер и $|V|$ вершин. Результаты эксперимента показаны на рис. 11.1. Победитель в этом эксперименте — один из классов `adjacency_list` с параметром `VertexList=listS`.

Добавление ребер

В этом тесте добавляются $|E|$ ребер к графу, который уже имеет $|V|$ вершин. Результаты показаны на рис. 11.2. Очевидным победителем для Visual C++ является `adjacency_list` с `VertexList=vecS`, независимо от размера и разреженности графа. При использовании GNU C++ побеждает класс `adjacency_list` с `EdgeList=listS` для разреженного графа.

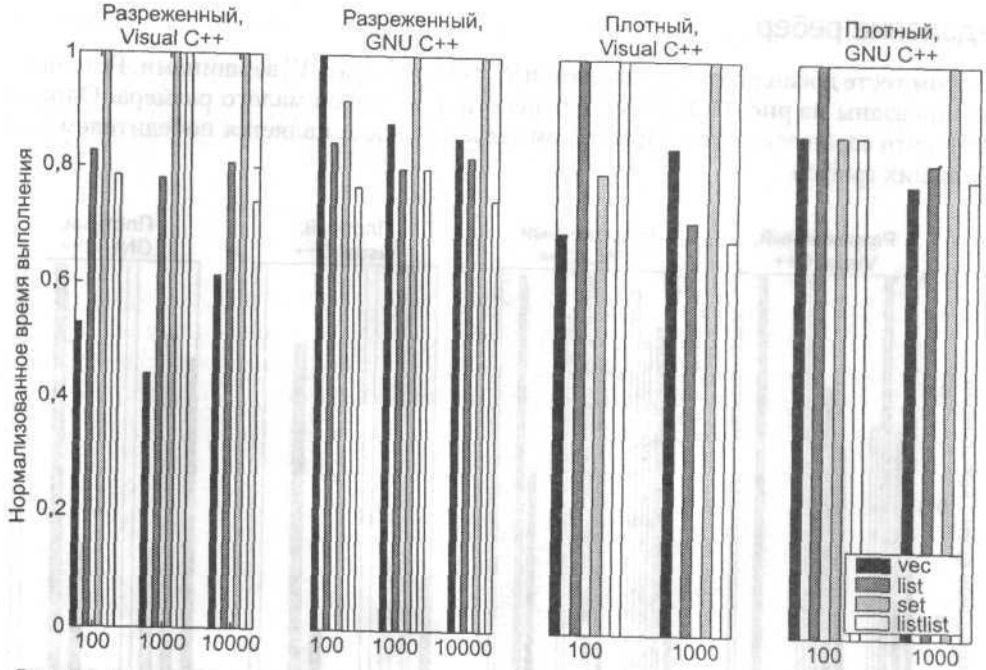


Рис. 11.1. Результаты измерения времени для эксперимента с добавлением ребер и вершин

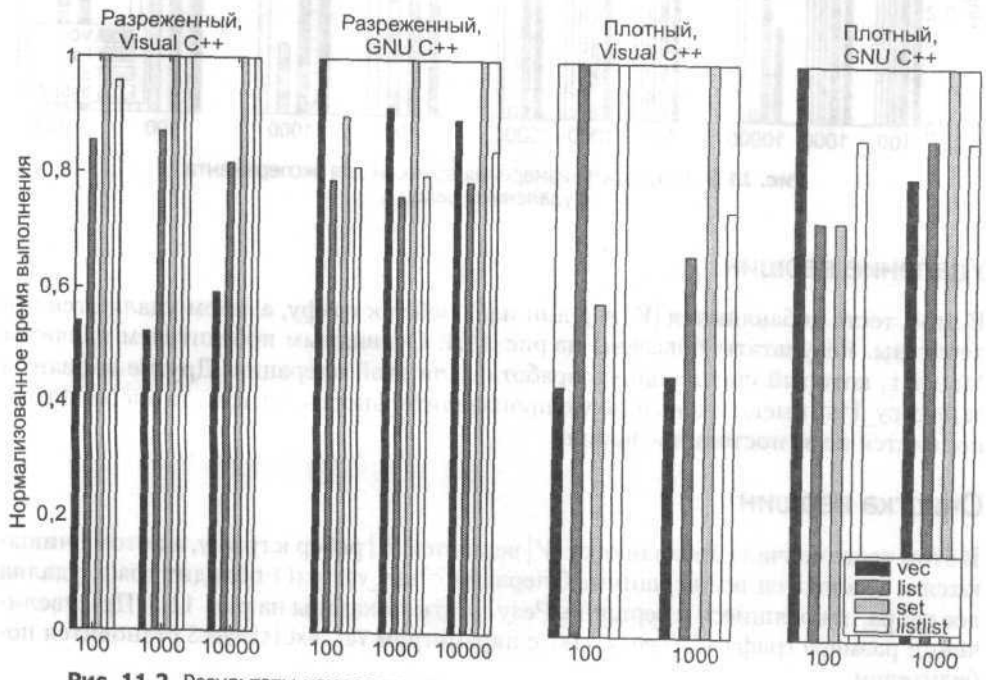


Рис. 11.2. Результаты измерения времени для эксперимента с добавлением ребер

Удаление ребер

В этом тесте добавляются и удаляются $|E|$ ребер графа с $|V|$ вершинами. Результаты показаны на рис. 11.3. Результат неясен для графов малого размера. Однако ясно, что `adjacency_list` с параметром `VertexList=setS` является победителем для больших графов.

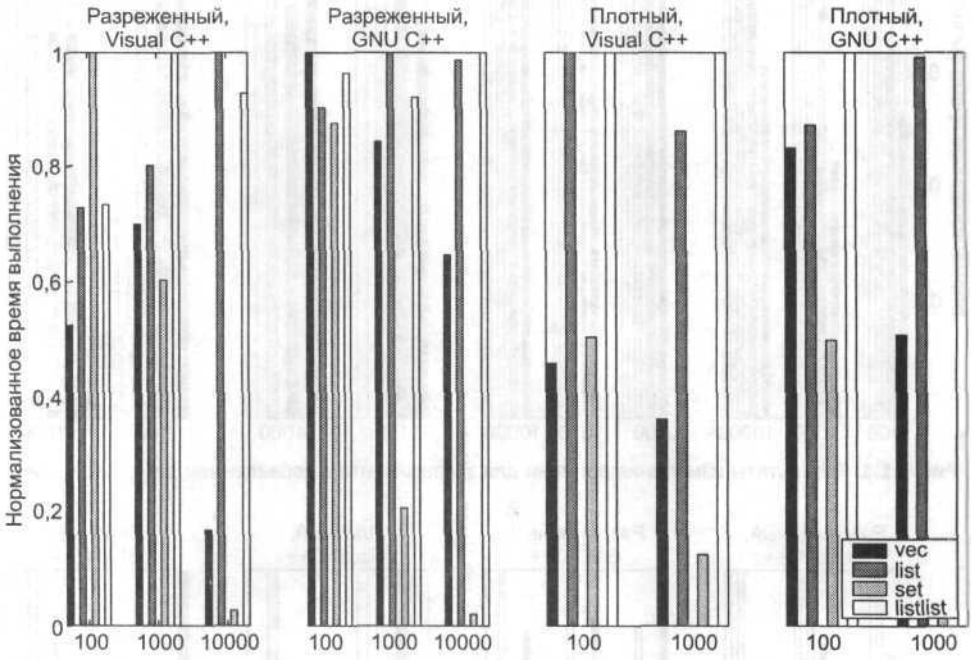


Рис. 11.3. Результаты измерения времени для эксперимента с удалением ребер

Удаление вершин

В этом тесте добавляются $|V|$ вершин и $|E|$ ребер к графу, а затем удаляются все вершины. Результаты показаны на рис. 11.4. Очевидным победителем является `listlist`, который специально разработан для этой операции. Другие варианты `adjacency_list` имеют очень плохую производительность, так как эта операция выполняется не за постоянное время.

Очистка вершин

В этом тесте сначала добавляются $|V|$ вершин и $|E|$ ребер к графу, а потом очищаются и удаляются все вершины. Операция `clear_vertex()` обходит граф, удаляя все ребра, относящиеся к вершине. Результаты показаны на рис. 11.5. При увеличении размера графа `adjacency_list` с параметром `VertexList=vecS` становится победителем.

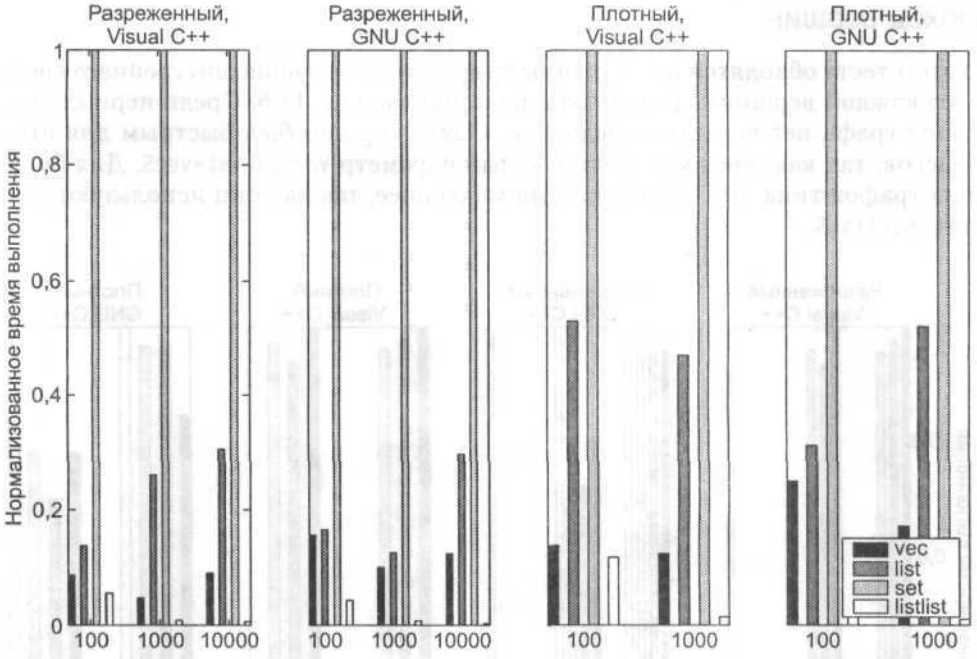


Рис. 11.4. Результаты измерения времени для эксперимента с удалением вершин

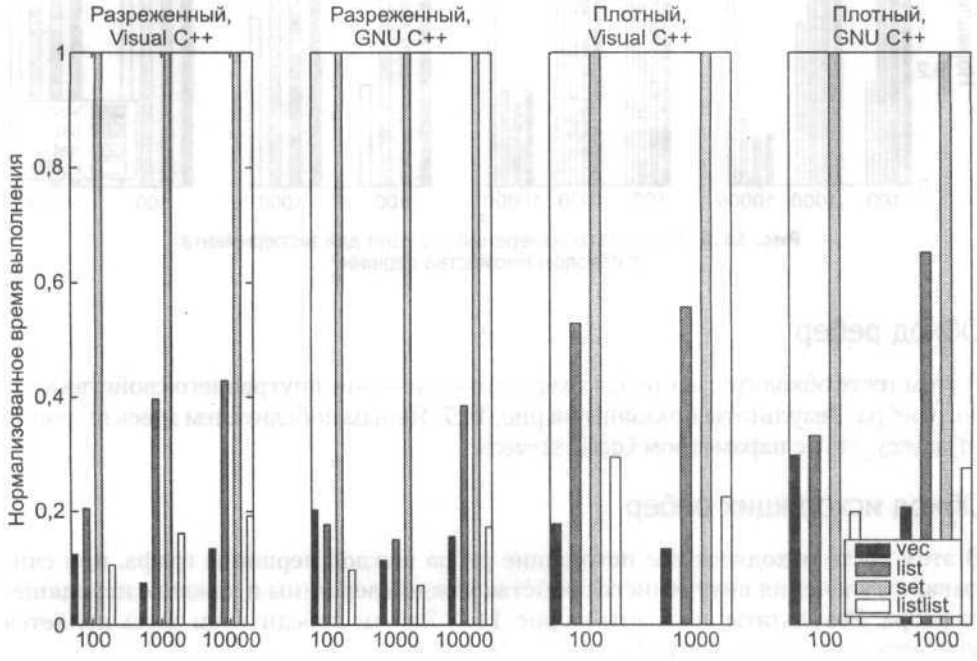


Рис. 11.5. Результаты измерения времени для эксперимента с очисткой вершин

Обход вершин

В этом тесте обходятся все вершины графа, читая значение внутреннего свойства каждой вершины. Результаты показаны на рис. 11.6. Среди первых трех типов графа нет четкого победителя. Обход вершин был быстрым для этих классов, так как они имеют одинаковый параметр `VertexList=vecS`. Для больших графов типа `listlist` обход был медленнее, так как они используют `VertexList=listS`.

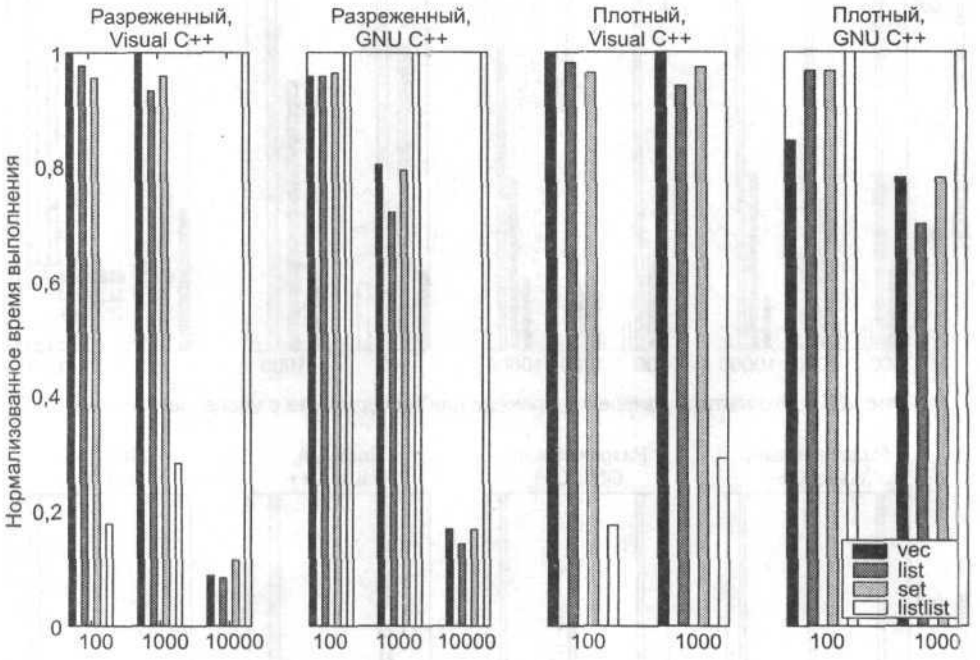


Рис. 11.6. Результаты измерения времени для эксперимента с обходом множества вершин

Обход ребер

В этом тесте обходятся все ребра графа, читая значение внутреннего свойства каждого ребра. Результаты показаны на рис. 11.7. Явным победителем здесь является `adjacency_list` с параметром `EdgeList=vecS`.

Обход исходящих ребер

В этом тесте обходятся все исходящие ребра каждой вершины графа, при считывании значения внутреннего свойства каждой вершины и каждого исходящего ребра. Результаты показаны на рис. 11.8. Явным победителем здесь является `adjacency_list` с `EdgeList=vecS`.

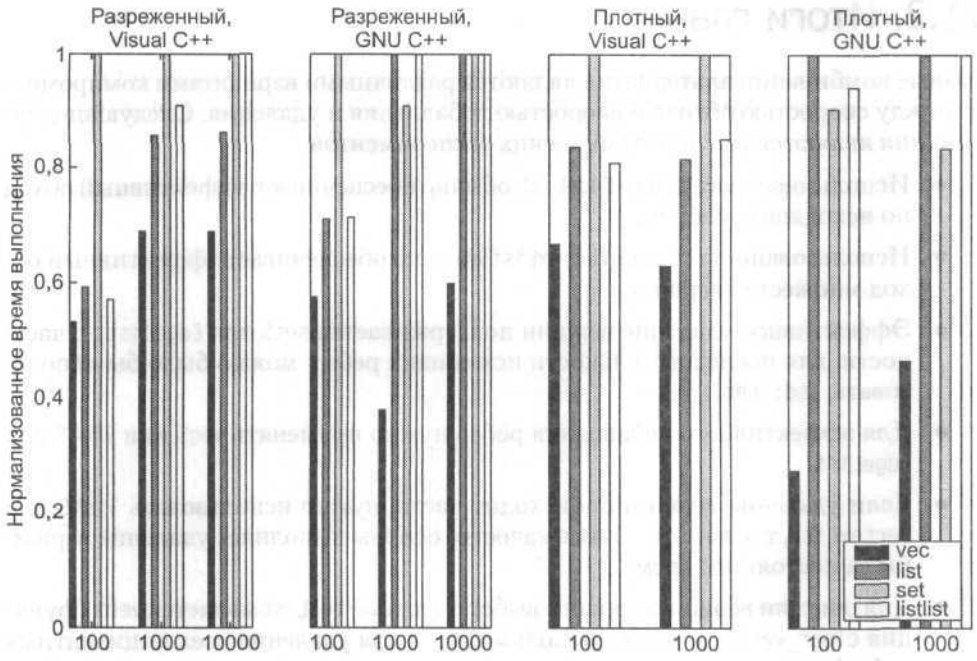


Рис. 11.7. Результаты измерения времени для эксперимента с обходом набора ребер

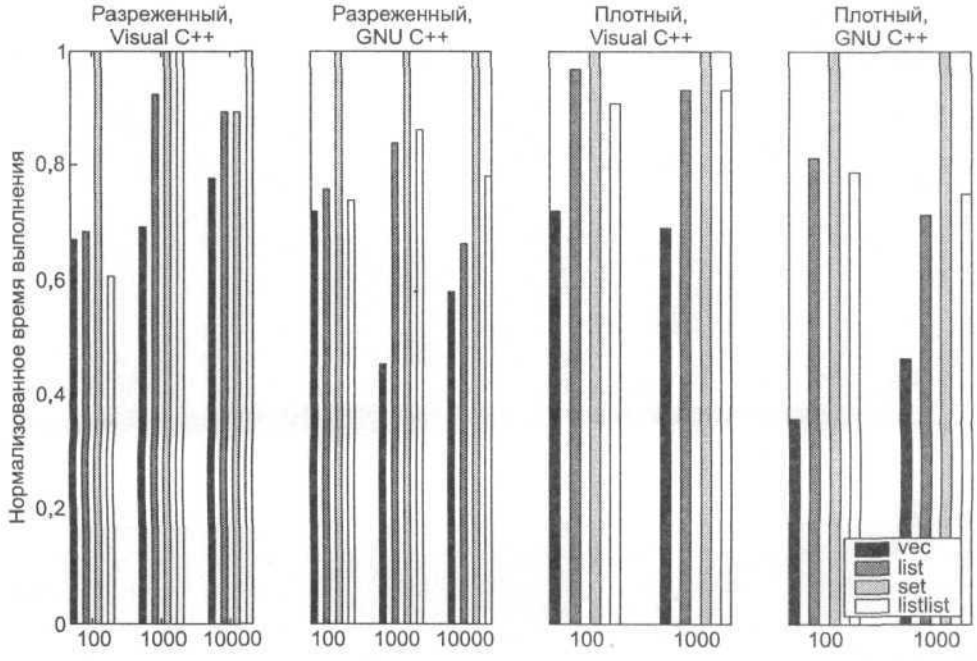


Рис. 11.8. Результаты измерения времени для эксперимента с обходом исходящих ребер

11.2. Итоги главы

Разные комбинации альтернатив являются различными вариантами компромисса между скоростью обхода и скоростью добавления и удаления. Следующие положения являются результатами наших экспериментов.

- Использование `vecS` для `EdgeList` обычно обеспечивает эффективный обход по исходящим ребрам.
- Использование `vecS` для `VertexList` обычно обеспечивает эффективный обход множества вершин.
- Эффективное удаление вершин поддерживается `setS` для `EdgeList`. В частности, для последовательности исходящих ребер можно было бы использовать `std::set`.
- Для эффективного добавления ребер нужно применять `vecS` или `listS` для `EdgeList`.
- Если удаление вершин происходит часто, нужно использовать `listS` для `VertexList`, так как `std::list` в качестве основы выполняет удаление вершины за постоянное время.
- Для очистки вершин хорошим выбором для `VertexList` является `vecS` (функция `clear_vertex()` может использоваться для удаления всех инцидентных ребер).

II Часть

Справочное руководство

Таблица 12.1 (*продолжение*)

Выражение	Возвращаемый тип или описание
BidirectionalGraph уточняет IncidenceGraph	
graph_traits<G>::in_edge_iterator	Итерация по входящим ребрам
in_edges(v, g)	std::pair<in_edge_iterator, in_edge_iterator>
in_degree(v, g)	degree_size_type
degree(e, g)	degree_size_type
AdjacencyGraph уточняет Graph	
graph_traits<G>::adjacency_iterator	Итерация по смежным вершинам.
adjacent_vertices(v, g)	std::pair<adjacency_iterator, adjacency_iterator>
VertexListGraph уточняет Graph	
graph_traits<G>::vertex_iterator	Итерация по набору вершин графа
graph_traits<G>::vertices_size_type	Целый беззнаковый тип для представления числа вершин
num_vertices(g)	vertices_size_type
vertices(g)	std::pair<vertex_iterator, vertex_iterator>
EdgeListGraph уточняет Graph	
graph_traits<G>::edge_descriptor	Тип объекта для обозначения ребер
graph_traits<G>::edge_iterator	Итерация по набору ребер графа
graph_traits<G>::edges_size_type	Целый беззнаковый тип для представления числа ребер
num_edges(g)	edges_size_type
edges(g)	std::pair<edge_iterator, edge_iterator>
source(e, g)	vertex_descriptor
target(e, g)	vertex_descriptor
AdjacencyMatrix уточняет Graph	
edge(u, v, g)	std::pair<edge_descriptor, bool>

12.1.1. Неориентированные графы

Интерфейс, который BGL предоставляет для доступа и манипуляции неориентированными графами, тот же, что и для ориентированных графов. Интерфейс один, так как есть определенная эквивалентность между неориентированными и ориентированными графами. То есть любой неориентированный граф может быть представлен как ориентированный, если неориентированное ребро (u, v) заменить двумя ориентированными (u, v) и (v, u). Такой ориентированный граф называется *ориентированной версией* неориентированного графа. На рис. 12.2 показан неориентированный граф и его ориентированная версия. Заметим, что для каждого ребра неориентированного графа ориентированный граф имеет два ребра. Таким образом, BGL использует функцию `out_edges()` (или `in_edges()`) для доступа к инцидентным ребрам в неориентированном графе. Аналогично `source()` и `target()` применяются для доступа к вершинам. Сначала это может показаться противоречивым фактом. Но, учитывая эквивалентность между неориентированными и ори-

ентированными графами, BGL позволяет применять многие алгоритмы как к ориентированным, так и к неориентированным графам.

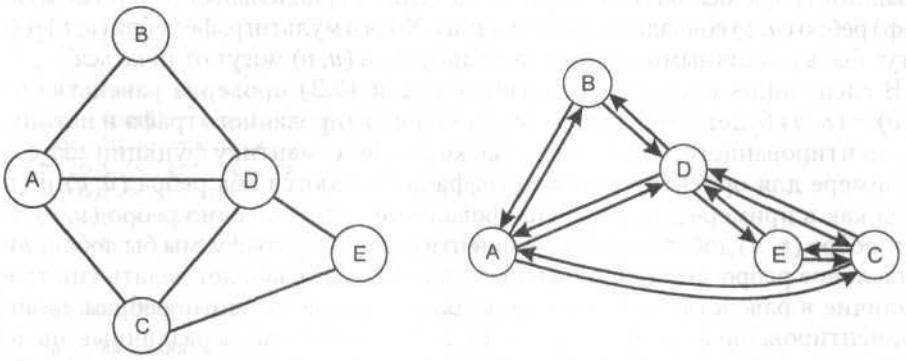


Рис. 12.2. Неориентированный граф и его ориентированный эквивалент

Следующий пример в листинге 12.1 демонстрирует использование `out_edges()`, `source()` и `target()` с неориентированным графом. Хотя обычно направление ребра не принимается во внимание для неориентированных графов, в случае применения функции `out_edges()` к вершине *u* начальная вершина для дескриптора ребра всегда *u*, а конечная — смежная с *u*. В функции `in_edges()`, соответственно, наоборот.

Листинг 12.1. Демонстрация некоторых функций для неориентированного графа

```
template <typename UndirectedGraph> void undirected_graph_demo1() {
  const int V = 3;
  UndirectedGraph undigraph(V);
  typename graph_traits<UndirectedGraph>::vertex_descriptor zero, one, two;
  typename graph_traits<UndirectedGraph>::out_edge_iterator out, out_end;
  typename graph_traits<UndirectedGraph>::in_edge_iterator in, in_end;

  zero = vertex(0, undigraph);
  one = vertex(1, undigraph);
  two = vertex(2, undigraph);
  add_edge(zero, one, undigraph);
  add_edge(zero, two, undigraph);
  add_edge(one, two, undigraph);

  std::cout << "исходящие(0): ";
  for (tie(out, out_end) = out_edges(zero, undigraph); out != out_end;
       ++out)
    std::cout << *out;
  std::cout << std::endl << "входящие(0): ";
  for (tie(in, in_end) = in_edges(zero, undigraph); in != in_end; ++in)
    std::cout << *in;
  std::cout << std::endl;
}
```

Вывод будет следующим:

```
исходящие(0): (0,1) (0,2)
входящие(0): (1,0) (2,0)
```

Хотя интерфейс для неориентированных и ориентированных графов одинаков, имеются некоторые отличия в поведении функций, поскольку равенство ребер

`graph_traits<G>::directed_category`

Теги для этой категории: `directed_tag` и `undirected_tag`.

- `graph_traits<G>::edge_parallel_category`

Описывает, позволяет ли графовый класс осуществлять вставку параллельных ребер (ребра с одной и той же парой начальной и конечной вершин). Два тега: `allow_parallel_edge_tag` и `disallow_parallel_edge_tag`.

- `graph_traits<G>::traversal_category`

Описывает виды обхода итераторами, которые поддерживает данный граф. Следующие классы тегов определены:

```
struct incidence_graph_tag { };
struct adjacency_graph_tag { };
struct bidirectional_graph_tag : public virtual incidence_graph_tag { };
struct vertex_list_graph_tag { };
struct edge_list_graph_tag { };
struct adjacency_matrix_tag { };
```

12.1.3. IncidenceGraph

Концепция IncidenceGraph обеспечивает интерфейс для эффективного доступа к исходящим ребрам каждой вершины графа. Исходящие ребра (out-edges) доступны через итераторы исходящих ребер. Функция `out_edges(v, g)` по данному дескриптору вершины `v` и графу `g` возвращает пару итераторов исходящих вершин. Первый итератор указывает на первое исходящее ребро вершины v, а второй итератор — за конец последовательности ребер. Разыменование итератора исходящих вершин возвращает дескриптор ребра. Инкремент итератора перемещает его к следующему исходящему ребру. Порядок появления исходящих ребер при итерации не фиксирован, хотя конкретная реализация графа может иметь некоторое упорядочение.

Уточнение для

Graph

Ассоциированные типы

Ниже приведены ассоциированные типы для концепции IncidenceGraph.

- `graph_traits<G>::edge_descriptor`

Дескриптор ребра соответствует уникальному ребру в графе. Он должен быть DefaultConstructible, Assignable и EqualityComparable.

- `graph_traits<G>::out_edge_iterator`

Итератор исходящих ребер для вершины v обеспечивает доступ к исходящим ребрам вершины v. Тип значения итератора — `edge_descriptor` от своего графа. Итератор исходящих вершин должен отвечать требованиям MultiPassInputIterator.

- `graph_traits<G>::degree_size_type`

Это беззнаковый целый тип, представляющий число исходящих или инцидентных ребер вершины.

Допустимые выражения

Ниже приведены допустимые выражения для концепции IncidenceGraph.

- `source(e, g)`

Тип результата: `vertex_descriptor`.

Семантика: возвращает дескриптор вершины u для ребра (u, v), представленного через `e`.

Предусловие: `e` — допустимый дескриптор ребра графа `g`.

- `target(e, g)`

Тип результата: `vertex_descriptor`.

Семантика: возвращает дескриптор вершины для ребра v (u, v), представленного через `e`.

Предусловие: `e` — допустимый дескриптор ребра графа `g`.

- `out_edges(v, g)`

Тип результата: `std::pair<out_edge_iterator,out_edge_iterator>`.

Семантика: возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам (для ориентированных графов) или инцидентным ребрам (для неориентированных графов) вершины v. Вершина v появляется как начальная во всех исходящих ребрах. Вершины, смежные с v, являются конечными в исходящих ребрах (неважно, ориентированный граф или нет).

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

- `out_degree(v, g)`

Тип результата: `degree_size_type`.

Семантика: возвращает число исходящих ребер (для ориентированных графов) или число инцидентных ребер (для неориентированных графов) вершины v.

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

Гарантии сложности

Функции `source()`, `target()` и `out_edges()` должны выполняться за постоянное время. Функция `out_degree()` должна выполняться за линейное время по количеству исходящих ребер вершины.

12.1.4. BidirectionalGraph

Концепция BidirectionalGraph уточняет IncidenceGraph: добавляется требование для эффективного доступа к входящим ребрам каждой вершины. Эта концепция выделена из IncidenceGraph, потому как предоставление эффективного доступа к входящим ребрам ориентированного графа обычно требует больше памяти, а многие алгоритмы не требуют доступа к таким ребрам. Для неориентированных графов это несущественно, так как дополнительная память не требуется.

Уточнение для

IncidenceGraph

Ассоциированные типы

Ниже приведен ассоциированный тип для концепции BidirectionalGraph.

- `graph_traits<G>::in_edge_iterator`

Итератор входящих ребер для вершины v обеспечивает доступ к входящим ребрам вершины v. Тип значения итератора — `edge_descriptor` от своего графа. Итератор исходящих вершин должен отвечать требованиям MultiPassInputIterator.

Допустимые выражения

Ниже приведены допустимые выражения для концепции BidirectionalGraph.

- `in_edges(v, g)`

Тип результата: `std::pair<in_edge_iterator, in_edge_iterator>`.

Семантика: возвращает пару итераторов, обеспечивающих доступ к входящим ребрам (для ориентированных графов) или инцидентным ребрам (для неориентированных графов) вершины v. Вершина v появляется как конечная во всех входящих ребрах. Вершины, для которых вершина v — смежная, являются начальными во входящих ребрах (неважно, ориентированный граф или нет).

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

- `in_degree(v, g)`

Тип результата: `degree_size_type`.

Семантика: возвращает число входящих ребер (для ориентированных графов) и число инцидентных ребер (для неориентированных графов) вершины v.

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

- `degree(v, g)`

Тип результата: `degree_size_type`.

Семантика: возвращает число входящих и исходящих ребер (для ориентированных графов) и число инцидентных ребер (для неориентированных графов) вершины v.

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

Гарантии сложности

Функция `in_edges()` должна выполняться за постоянное время, функция `in_degree()` — за линейное время, по числу входящих ребер.

12.1.5. AdjacencyGraph

Концепция AdjacencyGraph определяет интерфейс для доступа к смежным вершинам. Смежные вершины могут быть также получены как конечные вершины исходящих ребер. Однако для некоторых алгоритмов исходящие ребра не нужны и более удобно получать доступ к смежным вершинам непосредственно.

Уточнение для

Graph

Ассоциированные типы

Ниже приведен ассоциированный тип для концепции AdjacencyGraph.

- `graph_traits<G>::adjacency_iterator`

Итератор смежности для вершины v предоставляет доступ к вершинам, смежным с вершиной v. Тип значения итератора смежности — дескриптор вершины своего графа. Итератор должен отвечать требованиям MultiPassInputIterator.

Допустимые выражения

Ниже приведено допустимое выражение для концепции AdjacencyGraph.

- `adjacent_vertices(v, g)`

Тип результата: `std::pair<adjacency_iterator,adjacency_iterator>`.

Семантика: возвращает диапазон значений итератора, обеспечивающий доступ к вершинам, смежным с вершиной v. Более конкретно: это эквивалентно получению конечных вершин для каждого исходящего ребра вершины v.

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

Гарантии сложности

Функция `adjacent_vertices()` должна отрабатывать за постоянное время.

12.1.6. VertexListGraph

Концепция VertexListGraph определяет требования для эффективного обхода всех вершин графа.

Уточнение для

Graph

Ассоциированные типы

Ниже приведены ассоциированные типы для концепции VertexListGraph.

- `graph_traits<G>::vertex_iterator`

Итератор вершин (получаемый через `vertices(g)`) обеспечивает доступ ко всем вершинам графа. Тип итератора должен отвечать требованиям MultiPassInputIterator. Тип значения итератора вершин должен быть дескриптором вершины.

- `graph_traits<G>::vertices_size_type`

Это беззнаковый целый тип, которым можно представить число вершин в графе.

Допустимые выражения

Ниже приведены допустимые выражения для концепции VertexListGraph.

- `vertices(g)`

Тип результата: `std::pair<vertex_iterator, vertex_iterator>`.

Семантика: возвращает диапазон значений итератора, обеспечивая доступ ко всем вершинам графа g.

- num_vertices(g)

Тип результата: vertices_size_type.

Семантика: возвращает число вершин в графе g.

Гарантии сложности

Функция vertices() должна отрабатывать за постоянное время. Время выполнения функции num_vertices() должно линейно зависеть от числа вершин.

12.1.7. EdgeListGraph

Концепция EdgeListGraph уточняет концепцию Graph. Дополнительным требованием является обеспечение эффективного доступа ко всем ребрам графа.

Уточнение для

Graph

Ассоциированные типы

Ниже приведены ассоциированные типы для концепции EdgeListGraph.

- graph_traits<G>::edge_descriptor

Дескриптор ребра соответствует уникальному ребру в графе. Он должен быть DefaultConstructible, Assignable и EqualityComparable.

- graph_traits<G>::edge_iterator

Итератор ребер (полученный через edges(g)) обеспечивает доступ ко всем ребрам графа. Тип edge_iterator должен соответствовать требованиям InputIterator. Тип значения итератора ребер должен быть таким же, как у дескриптора ребер данного графа.

- graph_traits<G>::edges_size_type

Это беззнаковый целый тип, используемый для представления числа ребер в графе.

Допустимые выражения

Ниже приведены допустимые выражения для концепции EdgeListGraph.

- edges(g)

Тип результата: std::pair<edge_iterator, edge_iterator>.

Семантика: возвращает диапазон значений итератора для доступа ко всем ребрам графа g.

- source(e, g)

Тип результата: vertex_descriptor.

Семантика: возвращает дескриптор вершины для начальной вершины u ребра (u, v), заданного через e.

Предусловие: e — допустимый дескриптор ребра графа g.

- `target(e, g)`

Тип результата: `vertex_descriptor`.

Семантика: возвращает дескриптор вершины для конечной вершины v ребра (u, v), заданного через `e`.

Предусловие: `e` — допустимый дескриптор ребра графа `g`.

- `num_edges(g)`

Тип результата: `edges_size_type`.

Семантика: возвращает количество ребер в графе `g`.

Гарантии сложности

Функции `edges()`, `source()` и `target()` должны отрабатывать за постоянное время. Функция `num_edges()` должна выполняться за линейное по числу ребер время.

12.1.8. AdjacencyMatrix

Концепция `AdjacencyMatrix` уточняет концепцию `Graph` и имеет дополнительное требование обеспечения эффективного доступа к любому ребру в графе по заданным начальной и конечной вершинам.

Уточнение для

`Graph`

Допустимые выражения

Ниже приведено допустимое выражение для концепции `AdjacencyMatrix`.

- `edge(u, v, g)`

Тип результата: `std::pair<edge_descriptor, bool>`.

Семантика: возвращает пару, состоящую из флага, показывающего, существует ли в графе `g` ребро между `u` и `v`, и дескриптора ребра, если такое найдено.

Предусловие: `u`, `v` — допустимые для графа `g` дескрипторы вершин.

Гарантии сложности

Функция `edge()` должна возвращать результат за постоянное время.

12.2. Концепции для изменения графов

Этот раздел описывает BGL-интерфейс для модификации графа, то есть добавления и удаления вершин и ребер, изменения значений закрепленных за вершинами и ребрами свойств. Как и концепции для обхода графа, концепции модификации графа раздроблены на множество простых, для того чтобы предоставить разработчикам алгоритмов хороший выбор концепций для описания требований алгоритмов. В табл. 12.2 приведены допустимые выражения и ассоциированные типы для каждой концепции, а на рис. 12.3 показаны отношения уточнения между концепциями модификации графов. Некоторые из концепций, изображенных на рис. 12.1, также фигурируют на рис. 12.3, но все их отношения уточнения пропущены.

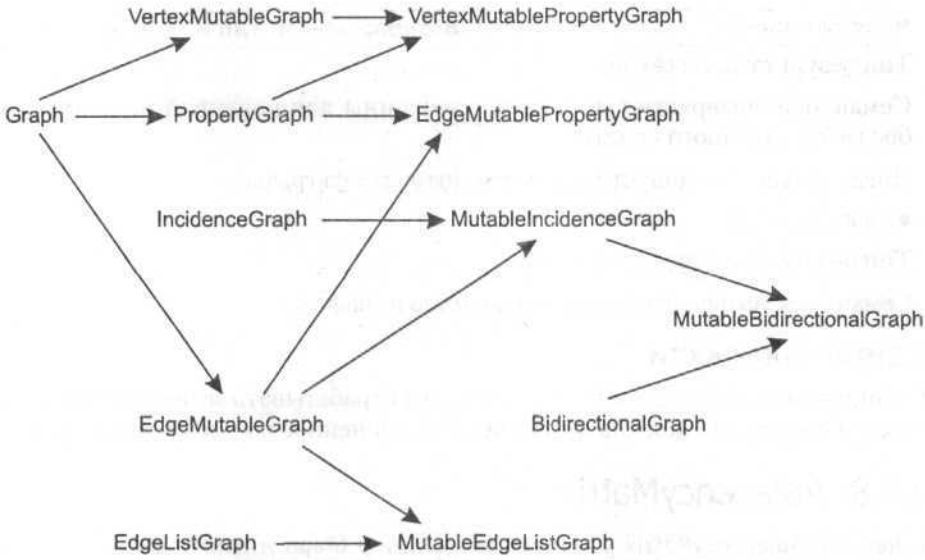


Рис. 12.3. Концепции для изменения графов и отношения уточнения между ними

Таблица 12.2. Краткая сводка концепций для изменения графа и доступа к свойствам графа

Выражение	Возвращаемый тип или описание
VertexMutableGraph уточняет Graph	
add_vertex(g)	vertex_descriptor
remove_vertex(v, g)	void
EdgeMutableGraph уточняет Graph	
clear_vertex(v, g)	void
add_edge(u, v, g)	std::pair<edge_descriptor, bool>
remove_edge(u, v, g)	void
remove_edge(e, g)	void
MutableIncidenceGraph уточняет IncidenceGraph и EdgeMutableGraph	
remove_edge(eiter, g)	void
remove_out_edge_if(u, p, g)	void
MutableBidirectionalGraph уточняет MutableIncidenceGraph и BidirectionalGraph	
remove_edge(eiter, g)	void
remove_out_edge_if(u, p, g)	void
MutableEdgeListGraph уточняет EdgeMutableGraph и EdgeListGraph	
remove_edge_if(p, g)	void
PropertyGraph уточняет Graph	
property_map<G, PropertyTag>::type	Тип для изменяемого отображения свойства вершины
property_map<G, Property-Tag>::const_type	Тип для неизменяемого отображения свойства вершины

Выражение	Возвращаемый тип или описание
get(ptag, g)	Функция для получения объекта-отображения свойства вершины
get(ptag, g, x)	Получить значение свойства для вершины или ребра x
put(ptag, g, x, v)	Положить значение свойства для вершины или ребра x равным v
VertexMutablePropertyGraph уточняет VertexMutableGraph и PropertyGraph	
add_vertex(vp, g)	vertex_descriptor
EdgeMutablePropertyGraph уточняет EdgeMutableGraph и PropertyGraph	
add_edge(u, v, ep, g)	std::pair<edge_descriptor, bool>

12.2.1. VertexMutableGraph

Концепция VertexMutableGraph — граф, модифицируемый по множеству вершин (vertex mutable), может быть изменен добавлением или удалением вершин. Управление памятью выполняется при реализации графа. Пользователь графа только вызывает `add_vertex()` и `remove_vertex()`, а реализация делает все остальное.

Уточнение для

Graph, DefaultConstructible

Допустимые выражения

Ниже приведены допустимые выражения для концепции VertexMutableGraph.

- `add_vertex(g)`

Тип результата: `vertex_descriptor`.

Семантика: добавляет новую вершину к графу. Возвращает дескриптор вершины для новой вершины.

- `remove_vertex(u, g)`

Тип результата: `void`.

Семантика: удаляет вершину *u* из множества вершин графа.

Предусловия: `u` — допустимый дескриптор вершины графа `g` и нет ребер, инцидентных вершине *u*. Функция `clear_vertex()` может быть использована для удаления всех инцидентных ребер.

Постусловия: `num_vertices(g)` уменьшается на единицу; вершина *u* больше не находится во множестве вершин графа, а ее дескриптор больше не может быть использован.

Гарантии сложности

Концепция VertexMutableGraph дает следующие гарантии сложности:

- вставка вершины происходит гарантированно за амортизированное постоянное время;
- удаление вершины должно происходить не дольше чем за $O(|E| + |V|)$.

12.2.2. EdgeMutableGraph

Концепция EdgeMutableGraph — граф, модифицируемый по множеству ребер (edge mutable), может быть изменен добавлением или удалением ребер. Управление памятью выполняется при реализации графа. Пользователь графа только вызывает `add_edge()` и `remove_edge()`, а реализация делает все остальное.

Уточнение для

Graph

Допустимые выражения

Ниже приведены допустимые выражения для концепции EdgeMutableGraph.

- `add_edge(u, v, g)`

Тип результата: `std::pair<edge_descriptor, bool>`.

Семантика: пытается вставить ребро (u, v) в граф, возвращая вставленное ребро или параллельное ребро и флаг, который показывает, было ли ребро вставлено. Эта операция не должна «портить» дескрипторы вершин или итераторы по вершинам графа, но она может сделать недействительными дескрипторы ребер или итераторы по ребрам. Порядок, в котором новое ребро появится при обходе итераторами ребер графа, не определен.

Предусловие: (u, v)являются вершинами графа.

Постусловие: (u, v) находится в наборе ребер графа. Возвращенный дескриптор ребра имеет вершину u в качестве начальной и v — в качестве конечной. Если граф позволяет иметь параллельные ребра, то возвращаемый флаг будет истиной. Если параллельные ребра в графе не разрешаются и ребро (u, v) уже есть в графе, флаг будет иметь значение «ложь». Если ребра (u, v) еще не было в графе, флаг будет иметь значение «истина».

- `remove_edge(u, v, g)`

Тип результата: `void`.

Семантика: удалить ребро (u, v) из графа. Если граф позволяет иметь параллельные ребра, функция удаляет все вхождения (u, v) в граф.

Предусловие: (u, v) является ребром из графа `g`.

Постусловие: (u, v) больше не является ребром графа `g`.

- `remove_edge(e, g)`

Тип результата: `void`.

Семантика: удалить ребро e из графа.

Предусловие: e является ребром графа `g`.

Постусловие: e больше не является ребром графа `g`.

- clear_vertex(u, g)

Тип результата: `void`.

Семантика: удалить все ребра, инцидентные вершине u графа.

Предусловие: u является допустимым дескриптором вершины для `g`.

Постусловие: u не участвует ни в одном из ребер графа в качестве начальной или конечной вершины.

Гарантии сложности

Концепция EdgeMutableGraph дает следующие гарантии сложности:

- вставка ребра должна происходить за амортизированное постоянное время или за время, равное $O(\log(|E|/|V|))$, если выполняются проверки для предотвращения добавления параллельных ребер;
- удаление ребра выполняется гарантированно за время $O(|E|)$;
- очистка вершины от инцидентных ребер выполняется не более чем за $O(|E| + |V|)$.

12.2.3. MutableIncidenceGraph

Концепция MutableIncidenceGraph обеспечивает возможность удаления ребер из списка исходящих ребер вершины.

Уточнение для

IncidenceGraph и EdgeMutableGraph

Допустимые выражения

Ниже приведены допустимые выражения для концепции MutableIncidenceGraph.

- `remove_edge(eiter, g)`

Тип результата: `void`.

Семантика: удаляет ребро, на которое указывает `eiter`, из графа, где `eiter` — итератор исходящих вершин графа.

Предусловие: `*eiter` является ребром графа.

Постусловие: `*eiter` больше не является ребром графа `g`.

- `remove_out_edge_if(u, p, g)`

Тип результата: `void`.

Семантика: удаляет все исходящие ребра вершины `u`, для которых предикат `p` возвращает истинное значение. Это выражение требуется, только если граф также моделирует IncidenceGraph.

Предусловие: `u` является допустимым дескриптором вершины графа `g`.

Постусловие: `p` возвращает ложь для всех исходящих ребер вершины *u* и все исходящие ребра, для которых `p` давал ложь, до сих пор находятся в графе.

Гарантии сложности

Концепция MutableIncidenceGraph дает следующие гарантии сложности:

- функция `remove_edge()` должна выполняться за постоянное время;
- функция `remove_out_edge_if()` должна выполняться за линейное по числу исходящих ребер время.

12.2.4. MutableBidirectionalGraph

Концепция MutableBidirectionalGraph определяет интерфейс для удаления ребер из списка входящих ребер вершины.

Уточнение для

BidirectionalGraph и MutableIncidenceGraph

Допустимые выражения

Ниже приведено допустимое выражение для концепции MutableBidirectionalGraph.

- `remove_in_edge_if(v, p, g)`

Тип результата: `void`.

Семантика: удаляет все входящие ребра вершины `v`, для которых `p` возвращает истину.

Предусловие: `v` — допустимый дескриптор вершины графа `g`.

Постусловие: `p` возвращает ложь для всех входящих ребер вершины *u* и все входящие ребра, для которых `p` давал ложь, до сих пор находятся в графе.

Гарантии сложности

Концепция MutableEdgeListGraph дает следующую гарантию сложности:

- функция `remove_in_edge_if()` выполняется за линейное по числу входящих вершин время.

12.2.5. MutableEdgeListGraph

Концепция MutableEdgeListGraph предоставляет возможность удалять ребра из списка ребер графа.

Уточнение для

EdgeMutableGraph

Допустимые выражения

Ниже приведено допустимое выражение для концепции MutableEdgeListGraph.

- `remove_edge_if(p, g)`

Тип результата: `void`.

Семантика: удаляет все ребра из графа `g`, для которых `p` возвращает истину.

Постусловие: `p` возвращает ложь для всех ребер в графе и граф до сих пор содержит все ребра, для которых `p` первоначально возвращал ложь.

Гарантии сложности

Концепция MutableEdgeListGraph дает следующую гарантию сложности:

- функция `remove_edge_if()` должна выполняться за линейное время по числу ребер в графе.

12.2.6. PropertyGraph

Концепция PropertyGraph — граф, который имеет некоторое свойство, связанное с каждой из вершин или ребер графа. Так как данный граф может иметь несколько свойств, связанных с каждой вершиной или ребром, для обозначения свойства могут быть использованы теги. В описании требований `PropertyTag` — это тип тега,

a tag — объект типа PropertyTag. Граф предоставляет функцию, которая возвращает объект—отображение свойства.

Уточнение для

Graph

Ассоциированные типы

Ниже приведены ассоциированные типы для концепции PropertyGraph.

- property_map<G, PropertyTag>::type

Тип отображения свойства для свойства, заданного PropertyTag. Этот тип должен быть изменяемым LvaluePropertyMap с таким же типом ключа, как у дескриптора вершины или ребра графа.

- property_map<G, PropertyTag>::const_type

Тип константного отображения свойства для свойства, заданного PropertyTag. Этот тип должен быть неизменяемым LvaluePropertyMap с таким же типом ключа, как у дескриптора вершины или ребра графа.

Допустимые выражения

Ниже приведены допустимые выражения для концепции PropertyGraph.

- get(ptag, g)

Тип результата: property_map<G, PropertyTag>::type, если g является изменяемым — и property_map<G, PropertyTag>::const_type — в противном случае.

Семантика: возвращает отображение свойства для свойства, заданного типом PropertyTag. Объект ptag используется только ради своего типа.

- get(ptag, g, x)

Тип результата: property_traits<PMap>::value_type.

Семантика: возвращает значение свойства (заданного типом PropertyTag), связанного с объектом x (вершина или ребро). Объект ptag используется только ради своего типа. Эта функция эквивалентна get(get(ptag, g), x).

Гарантии сложности

Функция get() должна выполняться за постоянное время.

12.2.7. VertexMutablePropertyGraph

Концепция VertexMutablePropertyGraph — это VertexMutableGraph и PropertyGraph с дополнительными функциями для указания значений свойств при добавлении вершин к графу.

Уточнение для

VertexMutableGraph и PropertyGraph

Ассоциированные типы

Ниже приведен ассоциированный тип для концепции VertexMutablePropertyGraph.

- vertex_property<G>::type

Тип объекта-свойства, закрепленного за вершиной.

Допустимые выражения

Ниже приведено допустимое выражение для концепции VertexMutablePropertyGraph.

- `add_vertex(vp, g)`

Тип результата: `vertex_descriptor`.

Семантика: добавляет новую вершину к графу и копирует `vp` в объект-свойство для новой вершины. Возвращается дескриптор вершины для новой вершины.

Гарантии сложности

Концепция VertexMutablePropertyGraph дает следующую гарантию сложности:

- `add_vertex()` гарантированно выполняется за амортизированное постоянное время.

12.2.8. EdgeMutablePropertyGraph

Концепция EdgeMutablePropertyGraph — это EdgeMutableGraph и PropertyGraph с дополнительными функциями для задания значений свойств при добавлении ребер к графу.

Уточнение для

EdgeMutableGraph и PropertyGraph

Ассоциированные типы

Ниже приведен ассоциированный тип для концепции EdgeMutablePropertyGraph.

- `edge_property<G>::type`

Тип объекта-свойства, закрепленного за ребром.

Допустимые выражения

Ниже приведено допустимое выражение для концепции EdgeMutablePropertyGraph.

- `add_edge(u, v, ep, g)`

Тип результата: `std::pair<edge_descriptor, bool>`.

Семантика: вставляет ребро (u, v) в граф и копирует объект `ep` в объект-свойство для этого ребра.

Предусловие: `u`, `v` являются допустимыми дескрипторами вершин графа `g`.

Гарантии сложности

Концепция EdgeMutablePropertyGraph дает следующую гарантию сложности:

- вставка ребра должна происходить либо за амортизированное постоянное время, либо за $O(\log(|E|/|V|))$, если при вставке проверяется недопустимость добавления параллельных ребер.

12.3. Концепции посетителей

Концепции посетителей (visitor concepts) играют такую же роль в BGL, как и функторы (functors) в STL. Функторы предоставляют механизм для расширения алгоритмов. Посетители позволяют пользователям вставлять свои собственные опе-

рации в различных точках графового алгоритма. В отличие от алгоритмов STL алгоритмы на графах обычно имеют несколько событийных точек, в которые пользователь может вставить обратный вызов (callback) через функтор. Таким образом, посетители имеют не единственный метод `operator()`, как у функтора, а несколько методов, соответствующих различным событийным точкам. В этом разделе мы определим концепции посетителей для основных алгоритмов BGL.

Как функциональные объекты в STL, посетители передаются по значению в алгоритмы BGL. Это означает, что нужно быть очень осторожным при сохранении состояния в объектах-посетителях.

Обозначения, использованные в этом разделе, приведены ниже:

- `V` — тип, моделирующий концепцию посетителя;
- `vis` — объект типа `V`;
- `G` — тип, являющийся моделью Graph;
- `g` — объект типа `G`;
- `e` — объект типа `graph_traits<G>::edge_descriptor`;
- `s`, `u` — объекты типа `graph_traits<G>::vertex_descriptor`.

12.3.1. BFSVisitor

Концепция BFSVisitor определяет интерфейс посетителя для поиска в ширину `breadth_first_search()`. Пользователи могут определить класс с интерфейсом BFSVisitor и передавать объект этого класса алгоритму `breadth_first_search()`, дополняя действия, производимые во время поиска по графу.

Уточнение для

CopyConstructible

Допустимые выражения

Ниже приведены допустимые выражения для концепции BFSVisitor.

- `vis.initialize_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается для каждой вершины графа перед началом поиска в графе.

- `vis.discover_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается, когда алгоритм встречает некоторую вершину *u* впервые. Все другие вершины, более близкие к исходной вершине, были уже просмотрены, а вершины, расположенные дальше от исходной, — еще нет.

- `vis.examine_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для каждого исходящего ребра каждой, только что посещенной вершины.

- `vis.tree_edge(e, g)`

Тип результата: `void`.

Семантика: если рассматриваемое ребро *e* входит в дерево поиска, вызывается эта функция. Вызов функции всегда предваряется вызовом функции `examine_edge()`.

- `vis.non_tree_edge(e, g)`

Тип результата: `void`.

Семантика: если рассматриваемое ребро *e* не входит в дерево поиска, вызывается эта функция. Вызов функции всегда предваряется вызовом функции `examine_edge()`. Для ориентированных графов такое ребро должно быть либо обратным, либо поперечным, для неориентированных — поперечным.

- `vis.gray_target(e, g)`

Тип результата: `void`.

Семантика: эта функция вызывается, если рассматриваемое ребро — ребро цикла и если его конечная вершина окрашена в серый цвет во время рассмотрения. Вызов функции всегда предваряется вызовом функции `cycle_edge()`. Серый цвет означает, что ребро находится в данный момент в очереди.

- `vis.black target(e, g)`

Тип результата: `void`.

Семантика: эта функция вызывается, если рассматриваемое ребро является ребром цикла и если его конечная вершина окрашена в черный цвет во время просмотра. Вызов функции всегда предваряется вызовом функции `cycle_edge()`. Черный цвет означает, что вершина уже была удалена из очереди.

- `vis.finish_vertex(u, g)`

Тип результата: `void`.

Семантика: эта функция вызывается для вершины после того, как все ее исходящие ребра были добавлены к дереву поиска и все смежные вершины просмотрены (но перед тем, как были рассмотрены их исходящие ребра).

12.3.2. DFSVisitor

Концепция `DFSVisitor` определяет интерфейс посетителя для поиска в глубину `depth_first_search()`. Пользователи могут определить класс с интерфейсом `DFSVisitor` и передавать объект этого класса алгоритму `depth_first_search()`, дополняя действия, производимые во время поиска по графу.

Уточнение для

`CopyConstructible`

Допустимые выражения

Ниже приведены допустимые выражения для концепции `DFSVisitor`.

- `vis.initialize_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается для каждой вершины графа перед началом поиска в графе.

- `vis.start_vertex(s, g)`

Тип результата: `void`.

Семантика: вызывается для исходной вершины перед началом поиска в графе.

- `vis.discover_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается, когда алгоритм встречает некоторую вершину *u* впервые.

- `vis.examine_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для каждой исходящей вершины после ее посещения.

- `vis.tree_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для каждого ребра, когда оно становится частью дерева поиска.

- `vis.back_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для обратных ребер графа. Для неориентированного графа имеется некоторая неопределенность между древесными ребрами и обратными ребрами, поскольку (u, v) и (v, u) являются одним и тем же ребром, но обе функции `tree_edge()` и `back_edge()` вызываются. Одним из способов разрешения этой неопределенности является запись древесных ребер, а затем отбрасывание обратных ребер, которые уже отмечены как древесные. Простым способом записывать древесные ребра является запись предшественников в функции `tree_edge()`.

- `vis.forward_or_cross_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для прямого или поперечного ребра в графе. Этот метод никогда не вызывается при поиске в неориентированном графе.

- `vis.finish_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается для вершины *u* после того, как `finish_vertex()` была вызвана для всех вершин дерева поиска с корнем в вершине *u*. Если вершина *u* является листом дерева поиска, функция `finish_vertex()` вызывается для вершины *u* после просмотра всех исходящих из *u* ребер.

12.3.3. DijkstraVisitor

Концепция DijkstraVisitor определяет интерфейс посетителя для `dijkstra_shortest_paths()` и подобных алгоритмов. Пользователь может создать класс, согласованный с этим интерфейсом, и затем передать объекты класса в `dijkstra_shortest_paths()` для расширения действий, производимых во время поиска по графу.

Уточнение для

CopyConstructible

Допустимые выражения

Ниже приведены допустимые выражения для концепции DijkstraVisitor.

- `vis.discover_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается, когда алгоритм встречает некоторую вершину *u* впервые.

- `vis.examine_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для каждого исходящего ребра каждой вершины после ее посещения.

- `vis.edge_relaxed(e, g)`

Тип результата: `void`.

Семантика: пусть (u, v) — это ребро e, d — отображение расстояний, а w — отображение весов. Если во время обхода $d[u] + w(u, v) < d[v]$, то ребро требует релаксации (его расстояние сокращается) и вызывается этот метод.

- `vis.edge_not_relaxed(e, g)`

Тип результата: `void`.

Семантика: если во время обхода ребро не требует релаксации, то вызывается этот метод.

- `vis.finish_vertex(u, g)`

Тип результата: `void`.

Семантика: вызывается для вершины после того, как все ее исходящие ребра были добавлены к дереву поиска и все смежные вершины были просмотрены (но перед тем, как были рассмотрены их исходящие ребра).

12.3.4. BellmanFordVisitor

Концепция BellmanFordVisitor определяет интерфейс посетителя для `bellman_ford_shortest_paths()`. Пользователь может создать класс с интерфейсом BellmanFordVisitor и затем передать объекты класса в `bellman_ford_shortest_paths()` через параметр `visitor()` для расширения действий, производимых во время поиска по графу.

Уточнение для

`CopyConstructible`

Допустимые выражения

Ниже приведены допустимые выражения для концепции BellmanFordVisitor.

- `vis.initialize_vertex(s, g)`

Тип результата: `void`.

Семантика: вызывается для каждой вершины графа перед началом поиска в графе.

- `vis.examine_edge(e, g)`

Тип результата: `void`.

Семантика: вызывается для каждого ребра графа `num_vertices(g)` раз.

- `vis.edge_relaxed(e, g)`

Тип результата: `void`.

Семантика: пусть (u, v) — ребро e, d — отображение расстояний, w — отображение весов. Если $d[u] + w(u, v) < d[v]$, то ребро требует релаксации (его расстояние сокращается) и вызывается этот метод.

- `vis.edge_not_relaxed(e, g)`

Тип результата: `void`.

Семантика: если во время обхода ребро не требует релаксации (см. выше), то вызывается этот метод.

- `vis.edge_minimized(e, g)`

Тип результата: `void`.

Семантика: после `num_vertices(g)` итераций по набору ребер графа делается одна последняя итерация для проверки того, каждое ли ребро было минимизировано. Если ребро минимизировано, то вызывается эта функция. Ребро (u, v) минимизировано, если $d[u] + w(u, v) \geq d[v]$.

- `vis.edge_not_minimized(e, g)`

Тип результата: `void`.

Семантика: если ребро не минимизировано, то вызывается эта функция. Это случается, когда в графе есть отрицательный цикл.

Алгоритмы BGL 13

13.1. Обзор

В этой главе представлена детальная информация по применению всех алгоритмов в Boost Graph Library.

Обобщенные алгоритмы BGL разделены на следующие категории:

1. Основные алгоритмы поиска.
2. Кратчайшие пути.
3. Минимальное остовное дерево.
4. Компоненты связности.
5. Максимальный поток.
6. Упорядочение вершин.

Все алгоритмы реализованы как шаблоны функций, где тип графа является параметром шаблона. Это позволяет использовать функцию с любым типом графа, который моделирует требуемые концепции. Описание каждого алгоритма содержит список требуемых графовых концепций, а описание каждого графового класса включает список концепций, которые граф моделирует. По перекрестным ссылкам через концепции можно определить, какие типы графов с какими алгоритмами могут быть использованы.

Кроме того, алгоритмы иногда параметризуются отображениями свойств, как например, отображение расстояний для алгоритмов кратчайших путей. Это дает пользователю контроль над тем, как свойства хранятся и извлекаются. Алгоритмы также параметризованы по типу посетителя, что позволяет пользователю задать обратные вызовы (call-backs), которые будут выполнены в определенных событийных точках алгоритма.

13.1.1. Информация об алгоритме

Информация об алгоритме будет представлена в виде следующих подразделов: «Прототипы», «Описание», «Где определен», «Параметры», «Именованные параметры», «Предусловия», «Сложность», «Пример».

Прототипы

Раздел справочника по каждому алгоритму начинается с прототипа функции. По именам параметров шаблона обычно можно судить о назначении параметра, а иногда также о требуемой параметром концепции. Однако точные требования для каждого параметра даны в разделе описания параметров.

Последний параметр для многих функций — `bgl_named_params`. Он служит для поддержки именованных параметров, что описано в разделе 2.7 и также рассматривается здесь. Если после `params` написано `= all defaults`, то для всех именованных параметров имеются значения по умолчанию, и эти параметры могут быть опущены.

Описание

В описании функции мы определяем задачу, которую решает эта функция, объясняем терминологию теории графов или идеи, которые необходимы для понимания задачи. Затем мы описываем семантику функций с точки зрения ее воздействия на параметры.

Где определен

В этом разделе указан заголовочный файл, который должен быть включен в программу (с помощью `#include`) для пользования функцией.

Параметры

Здесь приводится список всех обычных параметров функции (именованные рассмотрены в следующем разделе). Обычные параметры обязательны (для них нет значений по умолчанию).

Каждый параметр попадает в одну из следующих категорий:

- `IN`

  параметры читаются функцией и используются для получения информации. Функция никак не изменяет эти параметры;

- `OUT`

  параметры записываются функцией. Результаты работы функции хранятся в OUT-параметрах;

- `UTIL`

  параметры требуются для работы алгоритма, однако содержимое объектов, используемых как UTIL-параметры, обычно не интересно для пользователя. Эти параметры обычно как читаются, так и записываются.

Именованные параметры

Как сказано в разделе 2.7, BGL использует специальную технику для повышения удобства работы с функциями с большим набором параметров, когда многие

параметры имеют значения по умолчанию. В этом разделе перечисляются все именованные параметры для функции, с использованием такой же классификации, что и для обычных параметров. Кроме того, для каждого именованного параметра приведено значение по умолчанию.

Предусловия

В этом разделе мы описываем любые предусловия для функции. Обычно это включает требования к состоянию параметров отображений свойств.

Сложность

Временная сложность для каждого алгоритма дана в O-обозначениях. Сложность по памяти всегда не больше $O(|V|)$, если не указано обратное.

Пример

Для демонстрации использования алгоритма приводится простой пример.

13.2. Базовые алгоритмы

13.2.1. breadth_first_search

```
template <typename Graph, typename P, typename T, typename R>
void breadth_first_search(Graph& g,
  typename graph_traits<Graph>::vertex_descriptor s,
  const bgl_named_params<P, T, R>& params)
```

Функция `breadth_first_search()` выполняет поиск в ширину [31] ориентированного или неориентированного графа. При поиске в ширину сначала посещаются вершины, которые расположены ближе к исходной вершине, а потом более дальние вершины. В этом контексте расстояние определяется как число ребер в кратчайшем пути от исходной вершины. Функция `breadth_first_search()` может использоваться для вычисления кратчайших путей из одной вершины ко всем достижимым вершинам и длин соответствующих кратчайших путей. Определения, связанные с алгоритмом поиска в ширину, и детальный пример приведены в разделе 4.1.

Поиск в ширину использует две структуры данных для реализации обхода: цветную метку для каждой вершины и очередь. Еще не просмотренные вершины окрашены белым, серым обозначены пройденные вершины, у которых еще есть смежные белые. Черные вершины — уже посещенные вершины, причем вершины, смежные с данными, окрашены в серый или черный цвет. Алгоритм продолжает работу, удаляя вершину u из очереди и исследуя каждое исходящее ребро (u, v). Если смежная вершина v ранее не посещалась, она окрашивается в серый цвет и помещается в очередь. После рассмотрения всех исходящих ребер вершина u окрашивается в черный цвет и процесс повторяется. Псевдокод для алгоритма поиска в ширину приведен ниже. В псевдокоде показаны вычисления предшественников p, отметки посещения d и отметки окончания обработки t. По умолчанию функция `breadth_first_search()` не вычисляет эти свойства, однако имеются предопределенные посетители, которые могут быть использованы для этого.

```
ПОИСК_В_ШИРИНУ(G, s)
для каждой вершины u ∈ V[G]          ▷ инициализировать вершину u
color[u] ← БЕЛЫЙ
d[u] ← ∞
π[u] ← u
color[s] ← СЕРЫЙ
d[s] ← 0
В_ОЧЕРЕДЬ(Q, s)                      ▷ посетить вершину s
пока (Q ≠ ∅)
u ← ИЗ_ОЧЕРЕДИ(Q)                    ▷ рассмотреть вершину u
для каждой вершины v ∈ Adj[u]        ▷ рассмотреть ребро (u, v)
если (color[v] = БЕЛЫЙ)              ▷ (u, v) — древесное ребро
color[v] ← СЕРЫЙ
d[v] ← d[u] + 1
π[v] ← u
В_ОЧЕРЕДЬ(Q, v)                      ▷ посетить вершину v
иначе                                ▷ (u, v) — не древесное ребро
если (color[v] = СЕРЫЙ)
...                                  ▷ (u, v) имеет серый конец
иначе
...                                  ▷ (u, v) имеет черный конец
color[u] ← ЧЕРНЫЙ                    ▷ завершение обработки вершины u
возвратить (d,π)
```

Функция `breadth_first_search()` может быть расширена действиями определенными пользователем, которые вызываются в некоторых событийных точках. Действия должны быть представлены в форме объекта-посетителя, то есть объекта, отвечающего требованиям **BFSVisitor**. В указанном выше псевдокоде событийные точки обозначены в комментариях. По умолчанию функция `breadth_first_search()` не производит никаких действий, даже не записывает расстояния или предшественников. Однако это может быть легко добавлено определением посетителя.

Где определен

Алгоритм поиска в ширину находится в **boost/graph/breadth_first_search.hpp**.

Параметры

Ниже приведены параметры функции `breadth_first_search()`.

- `IN: Graph& g`

Ориентированный или неориентированный граф, который должен быть моделью **VertexListGraph** и **IncidenceGraph**.

- `IN: vertex_descriptor s`

Исходная вершина, с которой начинается поиск.

Именованные параметры

Ниже приведены именованные параметры функции `breadth_first_search()`.

- `IN: visitor(BFSVisitor vis)`

  Объект-посетитель, который активизируется внутри алгоритма в событийных точках, указанных в концепции **BFSVisitor**.

  По умолчанию: `default_bfs_visitor`.

- `UTIL/OUT: color_map(ColorMap color)`

  Используется алгоритмом для отслеживания продвижения по графу. Тип `ColorMap` должен быть моделью **ReadWritePropertyMap**, тип ключа — дескриптор вершины графа, тип значения `color_map` должен быть моделью **ColorValue**.

  По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа `default_color_type` размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `IN: vertex_index_map(VertexIndexMap i_map)`

  Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим только при использовании отображения свойства цвета по умолчанию. Тип `VertexIndexMap` должен моделировать **ReadablePropertyMap**. Тип значения отображения должен быть целочисленным типом. В качестве типа ключа отображения должен задаваться дескриптор вершины графа.

  По умолчанию: `get(vertex_index, g)`.

- `UTIL: buffer(Buffer& Q)`

  Очередь используется для определения порядка, в котором будут рассматриваться вершины. Если используется очередь вида FIFO («первым вошел — первым вышел»), обход графа будет происходить в обычном для поиска в ширину порядке. Могут быть использованы и другие типы очередей. Например, алгоритм Дейкстры может быть реализован с использованием очереди по приоритету. Тип `Buffer` должен моделировать концепцию **Buffer**.

  По умолчанию: `boost::queue`.

Предусловия

Очередь должна быть пуста.

Сложность

Временная сложность порядка $O(|E| + |V|)$. Пространственная сложность в наихудшем случае равна $O(|V|)$.

Пример

В примере (листинг 13.1) показано применение алгоритма поиска в ширину для графа, изображенного на рис. 13.1. Древесные ребра поиска в ширину обозначены черными линиями. Программа записывает порядок, в котором поиск в ширину посещает вершины графа. Исходный код для этого примера находится в файле example/bfs-example.cpp.

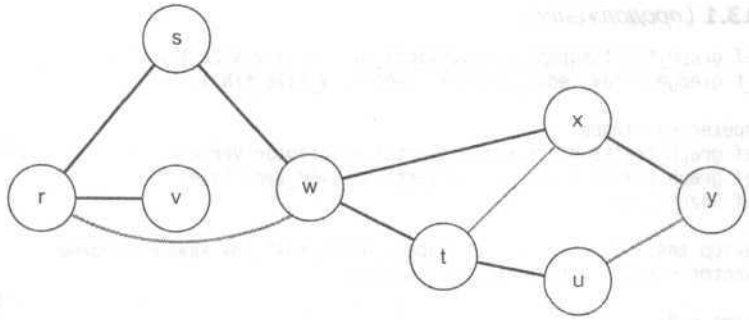


Рис. 13.1. Поиск в ширину на графе

Листинг 13.1. Применение алгоритма поиска в ширину

```
⟨ Посетитель записи времени для поиска в ширину ⟩ ≡
template < typename TimeMap > class bfs_time_visitor
:  public default_bfs_visitor {
  typedef typename property_traits < TimeMap >::value_type T;
public:
  bfs_time_visitor(TimeMap tmap, T & t):m_timemap(tmap), m_time(t) { }
  template < typename Vertex, typename Graph >
  void discover_vertex(Vertex u, const Graph & g) const {
    put(m_timemap, u, m_time++);
  }
  TimeMap m_timemap;
  T & m_time;
};

⟨ bfs-example.cpp ⟩ ≡

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/pending/indirect_cmp.hpp>
#include <boost/pending/integer_range.hpp>

#include <iostream>

using namespace boost;

⟨ Посетитель записи времени для поиска в ширину ⟩

int main() {
  using namespace boost;
  //  Выбрать графовый тип, который мы будем использовать
  typedef adjacency_list < vecS, vecS, undirectedS > graph_t;
  //  Подготовить идентификаторы вершин и имена
  enum { r, s, t, u, v, w, x, y, N };
  const char *name = "rstuvwxy";
  //  Указать ребра графа
  typedef std::pair < int, int >E;
  E edge_array[] = { E(r, s), E(r, v), E(s, w), E(w, r), E(w, t),
    E(w, x), E(x, t), E(t, u), E(x, y), E(u, y)
  };
  //  Создать графовый объект
  const int n_edges = sizeof(edge_array) / sizeof(E);
```

продолжение ⇩

Листинг 13.1 (*продолжение*)

```
typedef graph_traits<graph_t>::vertices_size_type v_size_t;
graph_t g(edge_array, edge_array + n_edges, v_size_t(N));

// Определения типов
typedef graph_traits < graph_t >::vertex_descriptor Vertex;
typedef graph_traits < graph_t >::vertices_size_type Size;
typedef Size* Iiter;

// Вектор для хранения свойства "время посещения" для каждой вершины
std::vector < Size > dtime(num_vertices(g));

Size time = 0;
bfs_time_visitor < Size * >vis(&dtime[0], time);
breadth_first_search(g, vertex(s, g), visitor(vis));

// Использовать std::sort для сортировки вершин по времени посещения
std::vector<graph_traits<graph_t>::vertices_size_type > discover_order(N);
integer_range < int >range(0, N);
std::copy(range.begin(), range.end(), discover_order.begin());
std::sort(discover_order.begin(), discover_order.end(),
          indirect_cmp < Iiter, std::less < Size > >(&dtime[0]));

std::cout << "порядок посещения: ";
for (int i = 0; i < N; ++i)
  std::cout << name[discover_order[i]] << " ";
std::cout << std::endl;

return EXIT_SUCCESS;
}
```

Вывод будет таким:

```
порядок посещения: s r w v t x u y
```

13.2.2. breadth_first_visit

```
template <typename IncidenceGraph, typename P, typename T, typename R>
void breadth_first_visit(IncidenceGraph& g,
  typename graph_traits<IncidenceGraph>::vertex_descriptor s,
  const bgl_named_params<P, T, R>& params);
```

Эта функция аналогична `breadth_first_search()` за исключением того, что цветовые маркеры не инициализируются в алгоритме. Пользователь должен сам окрасить вершины в белый цвет перед вызовом алгоритма. Поэтому требуемый тип графа — IncidenceGraph вместо VertexListGraph. Также эта разница позволяет иметь большую гибкость в отображении цветовых свойств. Например, можно использовать отображение, которое реализует функцию исключительно для подмножества вершин, что может быть более эффективно для памяти, поскольку поиск достигает только небольшой части графа.

Параметры

Ниже приведены параметры функции `breadth_first_visit()`.

- `IN: IncidenceGraph& g`

  Ориентированный или неориентированный граф, который должен быть моделью IncidenceGraph.

- IN: vertex_descriptor s

Исходная вершина, с которой начинается поиск.

Именованные параметры

Ниже приведены именованные параметры функции breadth_first_visit().

- IN: visitor(BFSVisitor vis)

Объект-посетитель, который активизируется внутри алгоритма в событийных точках, указанных в концепции BFSVisitor.

По умолчанию: bfs_visitor<null_visitor>.

- IN/UTIL/OUT: color_map(ColorMap color)

Используется алгоритмом для отслеживания продвижения по графу. Цвет каждой вершины должен быть инициализирован до вызова breadth_first_search(). Тип ColorMap должен быть моделью ReadWritePropertyMap, тип ключа — дескриптор вершины графа, тип значения color_map должен быть моделью ColorValue.

По умолчанию: get(vertex_color, g).

- UTIL: buffer(Buffer& Q)

Очередь используется для определения порядка, в котором вершины будут посещены. Если используется очередь вида FIFO («первым вошел — первым вышел»), обход графа будет происходить в обычном для поиска в ширину порядке. Могут быть использованы и другие типы очередей. Например, алгоритм Дейкстры может быть реализован с использованием очереди по приоритету. Тип Buffer должен моделировать концепцию Buffer.

По умолчанию: boost::queue.

13.2.3 depth_first_search

```
template <typename Graph, typename P, typename T, typename R>
void depth_first_search(Graph& g, const bgl_named_params<P, T, R>& params)
```

Функция depth_first_search() выполняет поиск в глубину в ориентированном или неориентированном графе. Когда возможно, поиск в глубину выбирает вершину, смежную с текущей, для следующего посещения. Если все смежные вершины уже посещены или у вершины нет смежных вершин, алгоритм возвращается на последнюю вершину, у которой есть не посещенные соседи. Когда все достижимые вершины просмотрены, алгоритм выбирает вершину из оставшихся не посещенных и продолжает обход. Алгоритм завершается, когда пройдены все вершины графа. Поиск в глубину полезен для классификации ребер графа и для наведения упорядочения вершин. В разделе 4.2 описаны различные свойства поиска в глубину и рассмотрен пример.

Подобно поиску в ширину, цветные пометки используются для отслеживания посещенных вершин. Белый цвет обозначает не посещенные вершины, серый — посещенные, но имеющие смежные не посещенные вершины. Черным обозначается посещенная вершина, у которой нет белых соседей.

Функция depth_first_search() активизирует заданные пользователем действия в определенных *событийных* точках. Это дает механизм для приспособления обобщенного алгоритма поиска в глубину ко многим ситуациям, в которых он может быть использован. В следующем ниже псевдокоде событийные точки поиска в глубину

обозначены в метках справа. Действия, определяемые пользователем, должны быть заданы в форме объекта-посетителя — объекта, чей тип удовлетворяет требованиям для DFSVisitor. В псевдокоде показаны вычисления предшественников π, отметок посещения d и отметок окончания обработки f. По умолчанию функция depth_first_search() не вычисляет этих свойств, однако пользователь может определить объекты-посетители для этого.

```
ПОИСК_В_ГЛУБИНУ(G)
для каждой вершины u ∈ V                    ▷ инициализировать вершину u
color [u] ← БЕЛЫЙ
π [u] = u
time ← 0
для каждой вершины u ∈ V
если (color [u] = БЕЛЫЙ)
call ПОИСК_
В_ГЛУБИНУ_ПОСЕЩЕНИЕ(G, u)                  ▷ исходная вершина u
возвратить (p, d, f)
ПОИСК_В_ГЛУБИНУ_ПОСЕЩЕНИЕ(G, u)
color [v] ← СЕРЫЙ                          ▷ посетить вершину u
d[u] ← time ← time + 1
для каждой вершины v ∈ Adj[u]
если (color [v] = БЕЛЫЙ)                   ▷ рассмотреть ребро (u, v)
π [v] = u
color [v] ← СЕРЫЙ
call ПОИСК_
В_ГЛУБИНУ_ПОСЕЩЕНИЕ(G, v)                  ▷ (u, v) — древесное ребро
иначе если (color [v] = СЕРЫЙ)
...                                         ▷ (u, v) — обратное ребро
иначе если (color[v] = ЧЕРНЫЙ)
...                                         ▷ (u, v) — прямое или поперечное ребро
color [u] ← ЧЕРНЫЙ                          ▷ завершение обработки вершины u
f [u] ← time ← time + 1
```

Где определен

Алгоритм поиска в глубину находится в boost/graph/depth_first_search.hpp.

Параметры

Ниже приведен параметр функции depth_first_search().

- IN: Graph& g

Ориентированный или неориентированный граф, который должен быть моделью VertexListGraph и IncidenceGraph.

Именованные параметры

Ниже приведены именованные параметры функции depth_first_search().

- IN: visitor(DFSVisitor vis)

Объект-посетитель, который активизируется внутри алгоритма в событийных точках, указанных в концепции DFSVisitor.

По умолчанию: `default_dfs_visitor`.

- `UTIL/OUT: color_map(ColorMap color)`

Используется алгоритмом для отслеживания продвижения по графу. Тип `ColorMap` должен быть моделью ReadWritePropertyMap, тип ключа — дескриптор вершины графа, тип значения `color_map` должен быть моделью ColorValue.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа `default_color_type` размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `IN: vertex_index_map(VertexIndexMap i_map)`

Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим, только когда используется отображение свойства цветовой окраски по умолчанию. Тип `VertexIndexMap` должен моделировать ReadablePropertyMap. Тип значения отображения должен быть целочисленным типом. В качестве типа ключа отображения должен использоваться тип дескриптора вершины.

По умолчанию: `get(vertex_index, g)`.

Сложность

Временная сложность порядка $O(|E| + |V|)$ и пространственная порядка $O(|V|)$.

Пример

В примере (листинг 13.2) показан поиск в глубину для графа, изображенного на рис. 13.2. Ребра леса поиска в глубину обозначены черными линиями. Исходный код этого примера находится в файле example/dfs-example.cpp.

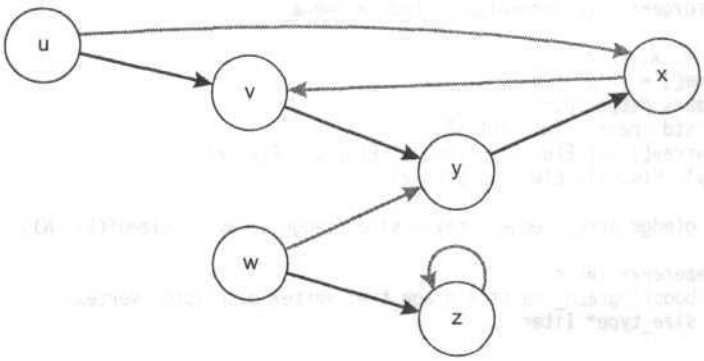


Рис. 13.2. Поиск в глубину на графе

Листинг 13.2. Применение алгоритма поиска в глубину

```
⟨ Посетитель поиска в глубину для записи времен посещения
  и окончания обработки вершин ⟩ ≡
template < typename TimeMap >
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
class dfs_time_visitor:public default_dfs_visitor {
  typedef typename property_traits < TimeMap >::value_type T;
public:
  dfs_time_visitor(TimeMap dmap, TimeMap fmap, T & t)
    : m_dtimemap(dmap), m_ftimemap(fmap), m_time(t) {
  }
  template < typename Vertex, typename Graph >
  void discover_vertex(Vertex u, const Graph & g) const {
    put(m_dtimemap, u, m_time++);
  }
  template < typename Vertex, typename Graph >
  void finish_vertex(Vertex u, const Graph & g) const {
    put(m_ftimemap, u, m_time++);
  }
  TimeMap m_dtimemap;
  TimeMap m_ftimemap;
  T & m_time;
};

⟨ dfs-example.cpp ⟩ ≡
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <boost/pending/integer_range.hpp>
#include <boost/pending/indirect_cmp.hpp>
#include <iostream>
using namespace boost;
⟨ Посетитель поиска в глубину для записи времен посещения
 и окончания обработки вершин ⟩

int main() {
  // Выбрать графовый тип, который мы будем использовать
  typedef adjacency_list < vecS, vecS, directedS > graph_t;
  typedef graph_traits < graph_t >::vertices_size_type size_type;
  // Подготовить идентификаторы вершин и имена
  enum
  { u, v, w, x, y, z, N };
  char name[] = { 'u', 'v', 'w', 'x', 'y', 'z' };
  // Указать ребра графа
  typedef std::pair < int, int >E;
  E edge_array[] = { E(u, v), E(u, x), E(x, v), E(y, x),
    E(v, y), E(w, y), E(w, z), E(z, z)
  };
  graph_t g(edge_array, edge_array + sizeof(edge_array) / sizeof(E), N);

  // Определения типов
  typedef boost::graph_traits < graph_t >::vertex_descriptor Vertex;
  typedef size_type* Iiter;

  // Векторы для хранения свойств "время посещения"
      и "время окончания обработки" для каждой вершины
  std::vector < size_type > dtime(num_vertices(g));
  std::vector < size_type > ftime(num_vertices(g));
  size_type t = 0;
  dfs_time_visitor < size_type * >vis(&dtime[0], &ftime[0], t);

  depth_first_search(g, visitor(vis));

  // Использовать std::sort для сортировки вершин по времени посещения
  std::vector < size_type > discover_order(N);
```

```
  integer_range < size_type > r(0, N);
  std::copy(r.begin(), r.end(), discover_order.begin());
  std::sort(discover_order.begin(), discover_order.end(),
            indirect_cmp < Iter, std::less < size_type > >(&dtime[0]));
  std::cout << "порядок посещения: ";
  int i;
  for (i = 0; i < N; ++i)
    std::cout << name[discover_order[i]] << " ";

  std::vector < size_type > finish_order(N);
  std::copy(r.begin(), r.end(), finish_order.begin());
  std::sort(finish_order.begin(), finish_order.end(),
            indirect_cmp < Iter, std::less < size_type > >(&ftime[0]));
  std::cout << std::endl << "порядок окончания обработки: ";
  for (i = 0; i < N; ++i)
    std::cout << name[finish_order[i]] << " ";
  std::cout << std::endl;
  return EXIT_SUCCESS;
}
```

Вывод будет таким:

```
порядок посещения: u v y x w z
порядок окончания обработки: x y v u z w
```

13.2.4. depth_first_visit

```
template <typename IncidenceGraph, typename DFSVisitor, typename ColorMap>
void depth_first_visit(IncidenceGraph& G,
  typename graph_traits<IncidenceGraph>::vertex_descriptor s,
  DFSVisitor vis, ColorMap color);
```

Функция `depth_first_visit()` — рекурсивная часть поиска в глубину. Главная задача этой функции — реализовать `depth_first_search()`, но иногда она может быть полезна сама по себе. Для дополнительной информации см. описание `depth_first_search()`.

Где определен

Функция `depth_first_visit()` находится в boost/graph/depth_first_search.hpp.

Параметры

Ниже приведены параметры функции `depth_first_visit()`.

- `IN: IncidenceGraph& g`

Ориентированный или неориентированный граф, который должен быть моделью IncidenceGraph.

- `IN: vertex_descriptor s`

Исходная вершина, с которой нужно начинать поиск.

- `IN: DFSVisitor visitor`

Объект-посетитель, который активизируется внутри алгоритма в событийных точках, указанных в концепции DFSVisitor.

- `UTIL: ColorMap color`

Используется алгоритмом для отслеживания продвижения по графу. Тип `ColorMap` должен быть моделью ReadWritePropertyMap, тип ключа — дескриптор вершины графа, тип значения `color_map` должен быть моделью ColorValue.

Сложность

Временная сложность порядка $O(|E|)$ и пространственная — порядка $O(|V|)$.

13.2.5. topological_sort

```
template <typename Graph, typename OutputIterator,
  typename P, typename T, typename R>
void topological_sort(Graph& G, OutputIterator result,
  const bgl_named_params<P, T, R>& params = all defaults)
```

Функция `topological_sort()` реализует алгоритм топологической сортировки, который создает линейное упорядочение вершин, такое, что если ребро (u, v) присутствует в графе, то u стоит в упорядочении раньше, чем v. Граф должен быть ориентированным ациклическим графом.

Обратное топологическое упорядочение записывается итератором вывода `result`, поэтому его нужно обратить для получения нормального топологического порядка. Есть несколько способов это сделать. Можно создать `std::vector` размером $|V|$ для сохранения вывода и затем использовать обратный итератор вектора для итератора `result`. Или можно использовать итератор вставки в конец `back_insert_iterator` с пустым вектором, а затем применить `std::reverse()`. Еще одна альтернатива — итератор вставки в начало `front_insert_iterator` с контейнером вроде `std::list` или `std::deque`.

Реализация включает простой вызов поиска в глубину [10]. В разделе 1.4.1 есть пример использования топологической сортировки для планирования заданий, а в главе 3 топологическая сортировка применяется при написании обобщенного графового алгоритма.

Где определен

Алгоритм топологической сортировки находится в boost/graph/topological_sort.hpp.

Параметры

Ниже приведены параметры функции `topological_sort()`.

- `IN: Graph& g`

  Ориентированный или неориентированный граф, который должен быть моделью VertexListGraph и IncidenceGraph.

- `IN: OutputIterator result`

  Вершины выводятся этим итератором в обратном топологическом порядке. Тип `OutputIterator` должен принимать дескрипторы вершин как выход, и тип итератора должен быть моделью OutputIterator.

Именованные параметры

Ниже приведены именованные параметры функции `topological_sort()`.

- `UTIL/OUT: color_map(ColorMap color)`

  Используется алгоритмом для отслеживания продвижения по графу. Тип `ColorMap` должен быть моделью ReadWritePropertyMap, тип ключа — дескриптор вершины графа, тип значения `color_map` должен быть моделью `ColorValue`.

  По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа `default_color_type`, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `IN: vertex_index_map(VertexIndexMap i_map)`

  Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим, только когда используется отображение свойств цветовой окраски по умолчанию. Тип `VertexIndexMap` должен моделировать `ReadablePropertyMap`. Тип значения отображения должен быть целочисленным типом. В качестве типа ключа отображения требуется задавать тип дескриптора вершины.

  По умолчанию: `get(vertex_index, g)`.

Сложность

Временная сложность порядка $O(|V| + |E|)$ и пространственная — $O(|V|)$.

Пример

См. раздел 1.4.1, где дан пример использования `toplogical_sort()`.

13.3. Алгоритмы кратчайших путей

13.3.1. dijkstra_shortest_paths

```
template <typename Graph, typename P, typename T, typename R>
void dijkstra_shortest_paths(const Graph& g,
  typename graph_traits<Graph>::vertex_descriptor s,
  const bgl_named_params<P, T, R>& params)
```

Алгоритм Дейкстры [10, 11] решает задачу нахождения кратчайших путей из одной вершины на взвешенном, ориентированном или неориентированном графе для неотрицательных весов ребер. Для случая, когда у некоторых ребер имеются отрицательные веса, лучше использовать алгоритм Беллмана–Форда, а когда все веса равны единице, применяется алгоритм поиска в ширину. Описание задачи кратчайшего пути приведено в разделе 5.1.

Есть два варианта получения результата из функции `dijkstra_shortest_paths()`. Первый — предоставить отображение свойства расстояний через параметр `distance_map()`. В этом случае кратчайшее расстояние от исходной вершины до любой другой в графе будет записано в отображении расстояний. Второй вариант — записать дерево кратчайших путей в отображение предшественников: для каждой вершины $u \in V$, $\pi[u]$ будет предшественником u в дереве кратчайших путей (если, конечно, не выполняется $\pi[u] = u$, так как в этом случае u либо является исходной, либо недостижима из исходной). В дополнение к этим двум вариантам пользователи могут предоставить своего собственного посетителя, который может производить определенные действия в любой из событийных точек.

Алгоритм Дейкстры находит все кратчайшие пути из одной исходной вершины к любой другой вершине, итеративно наращивая множество вершин S, к которым он знает кратчайший путь. На каждом шаге алгоритма добавление следующей вершины к S определяется очередью по приоритету. Очередь содержит вершины из множества $V - S$, а значением приоритета является их метка расстояния — длина кратчайшего пути, который известен на данный момент для каждой вершины[1].

[1] Алгоритм, использованный здесь, сохраняет память, не помещая все вершины $V - S$ в очередь по приоритету, а только те вершины из $V - S$, которые были посещены и поэтому имеют расстояние меньше бесконечности.

Вершина u в начале очереди добавляется к S, а каждое из ее исходящих ребер релаксируется. Если расстояние до вершины u в сумме с весом исходящего ребра меньше, чем метка расстояния для вершины v, то оценка расстояния для вершины v уменьшается. После этого алгоритм обрабатывает следующую вершину с головы очереди. Алгоритм завершается, когда очередь пуста.

Алгоритм использует цветовые маркеры (белый, серый и черный) для отслеживания принадлежности вершины некоторому множеству. Черные вершины принадлежат S, белые — множеству $V - S$. Белые вершины еще не посещались, а серые находятся в очереди по приоритетам. По умолчанию алгоритм заводит массив для хранения цветовых маркеров для каждой вершины графа. Вы можете предоставить свое собственное хранилище и получать доступ к цветам с помощью именованного параметра `color_map()`.

Ниже приведен псевдокод алгоритма Дейкстры для вычисления кратчайших путей из одной вершины. Здесь w обозначает вес ребра, d — метка расстояния, π — предшественник каждой вершины, используемый для кодирования дерева кратчайших путей. Q — очередь по приоритету, которая поддерживает операцию УМЕНЬШИТЬ_КЛЮЧ. Событийные точки для алгоритма обозначены рядом с метками справа.

```
ДЕЙКСТРА(G, s, w)
для каждой вершины u ∈ V                    ▷ инициализировать вершину u
d[u] ← ∞
π[u] ← u
color[u] ← БЕЛЫЙ
color[s] ← СЕРЫЙ
d[s] ← 0
ВСТАВИТЬ(Q, s)                              ▷ посетить вершину s
пока (Q ≠ ∅)
u ← ВЗЯТЬ_МИНИМАЛЬНОЕ (Q)                   ▷ рассмотреть вершину u
S ← S ∪ {u}
для каждой вершины v ∈ Adj[u]               ▷ рассмотреть ребро (u, v)
если (w(u, v) + d[u] < d[v])
d[v] ← w(u, v) + d[u]                       ▷ ребро (u, v) релаксируется
π[v] ← u
если (color[v] = БЕЛЫЙ)
color[v] ← СЕРЫЙ
ВСТАВИТЬ(Q, v)                              ▷ посетить вершину v
иначе если (color[v] = СЕРЫЙ)
УМЕНЬШИТЬ_КЛЮЧ(Q, v, w(u, v) + d[u])
иначе
...                                         ▷ ребро (u, v) не релаксируется
color[u] ← ЧЕРНЫЙ                           ▷ завершение обработки вершины u
возвратить (d, π)
```

Где определен

Алгоритм Дейкстры находится в boost/graph/dijkstra_shortest_paths.hpp.

Параметры

Ниже приведены параметры функции `dijkstra_shortest_paths()`.

- `IN: const Graph& g`

  Графовый объект, к которому применяется алгоритм. Тип `Graph` должен быть моделью VertexListGraph и IncidenceGraph.

- `IN: vertex_descriptor s`

  Исходная вершина. Все расстояния вычисляются от этой вершины, и в ней находится корень дерева кратчайших путей.

Именованные параметры

Именованные параметры функции `dijkstra_shortest_paths()` приведены ниже.

- `IN: weight_map(WeightMap w_map)`

  Вес или «длина» каждого ребра в графе. Тип `WeightMap` должен быть моделью ReadablePropertyMap. Дескриптор ребра требуется задавать в качестве типа ключа для отображения весов. Тип значения отображения весов должен быть таким же, как тип значения для отображения расстояний.

  По умолчанию: `get(edge_weight, g)`.

- `IN: vertex_index_map(VertexIndexMap i_map)`

  Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим для эффективного обновления структуры данных при релаксации ребра. Тип `VertexIndexMap` должен моделировать ReadablePropertyMap. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения.

  По умолчанию: `get(vertex_index, g)`.

- `OUT: predecessor_map(PredecessorMap p_map)`

  Отображение предшественников записывает ребра минимального остовного дерева. По завершении алгоритма ребра $(\pi[u], u)$ $\forall u \in V$ находятся в минимальном остовном дереве. Если $\pi[u] = u$, это означает, что либо u является исходной вершиной, либо u недостижима из исходной. Тип `PredecessorMap` должен быть ReadWritePropertyMap с типами ключа и вершины такими же, как тип дескриптора вершины графа.

  По умолчанию: `dummy_property_map`.

- `UTIL/OUT: distance_map(DistanceMap d_map)`

  Вес кратчайшего пути из исходной вершины в каждую вершину графа записан в этом отображении свойств. Вес кратчайшего пути — это сумма весов ребер, из которых состоит путь. Тип `DistanceMap` должен моделировать ReadWritePropertyMap. Тип дескриптора вершины графа должен задаваться в качестве типа ключа отображения расстояний. Тип значения — элемент типа `Monoid`, состоящий из функционального объекта `combine` и объекта `zero` для нейтрального элемента (см. главу 16). Также тип значения расстояния должен обеспечивать

StrictWeakOrdering (ослабленное строгое упорядочение), что и дает объект-функция compare.

По умолчанию: iterator_property_map, созданный из вектора std::vector с элементами того же типа, что и тип значения WeightMap, размером num_vertices(). Он использует i_map для отображения индексов.

- IN: distance_combine(BinaryFunction combine)

Объект-функция, являющийся операцией концепции Monoid для типа значения расстояния. Этот функциональный объект складывает длины для получения полной длины пути.

По умолчанию: closed_plus<D>, где D — тип значения отображения расстояний. Тип closed_plus определен в boost/graph/relax.hpp.

- IN: distance_compare(BinaryPredicate compare)

Объект-функция, который задает StrictWeakOrdering на значениях расстояний. Используется для определения того, какой из путей короче.

По умолчанию: std::less<D>, где D — тип значения отображения расстояний.

- IN: distance_inf(D inf)

Объект inf должен давать наибольшее значение любого объекта D. То есть compare(d, inf) == true для любого d != inf. Тип D — тип значения DistanceMap.

По умолчанию: std::numeric_limits<D>::max().

- IN: distance_zero(D zero)

Значение должно быть нейтральным элементом для Monoid, состоящего из значений расстояний и объекта-функции combine. Тип D — тип значения для DistanceMap.

По умолчанию: D.

- UTIL/OUT: color_map(ColorMap c_map)

Используется во время выполнения алгоритма для маркировки вершин. В начале алгоритма вершины имеют белый цвет и становятся серыми, когда они попадают в очередь. Они перекрашиваются в черный при удалении из очереди. В конце алгоритма все вершины, достижимые из исходной, имеют черный цвет. Остальные вершины будут белыми. Тип ColorMap должен быть моделью ReadWritePropertyMap. Дескриптор вершины требуется задавать в качестве типа ключа для отображения, и тип значения отображения должен моделировать ColorValue.

По умолчанию: iterator_property_map, созданный из вектора std::vector с элементами типа default_color_type, размером num_vertices(). Он использует i_map для отображения индексов.

- IN: visitor(Vis v)

Используйте этот параметр для задания действий, которые должны выполняться в определенных событийных точках внутри алгоритма. Тип Vis должен быть моделью DijkstraVisitor.

По умолчанию: default_dijkstra_visitor.

Сложность

Временная сложность порядка $O((|V| + |E|) \log |V|)$ или просто $O(|E| \log |V|)$, если все вершины достижимы из исходной вершины.

Пример

Исходный код примера из листинга 13.3 находится в файле example/dijkstra-example.cpp. Граф, использованный в этом примере, показан на рис. 13.3. Ребра в дереве кратчайших путей обозначены черными линиями.

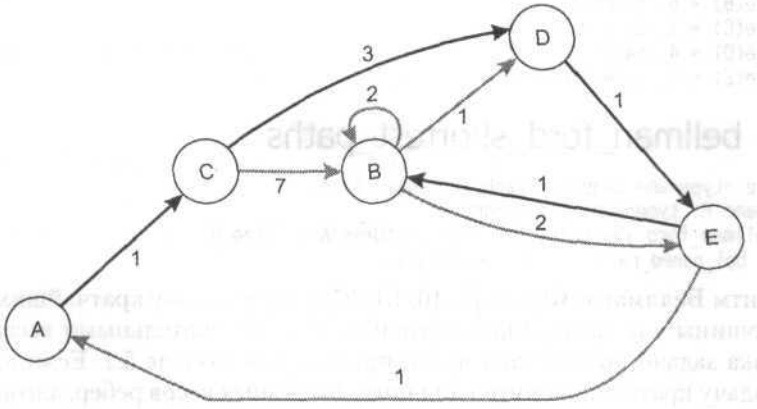


Рис. 13.3. Граф, использованный в примере для алгоритма Дейкстры

Листинг 13.3. Пример алгоритма Дейкстры

```
typedef adjacency_list < listS, vecS, directedS,
  no_property, property < edge_weight_t, int > > graph_t;
typedef graph_traits < graph_t >::vertex_descriptor vertex_descriptor;
typedef graph_traits < graph_t >::edge_descriptor edge_descriptor;
typedef std::pair<int, int> Edge;

const int num_nodes = 5;
enum nodes { A, B, C, D, E };
char name[] = "ABCDE";
Edge edge_array[] = { Edge(A, C), Edge(B, B), Edge(B, D), Edge(B, E),
  Edge(C, B), Edge(C, D), Edge(D, E), Edge(E, A), Edge(E, B)
};
int weights[] = { 1, 2, 1, 2, 7, 3, 1, 1, 1 };
int num_arcs = sizeof(edge_array) / sizeof(Edge);

graph_t g(edge_array, edge_array + num_arcs, weights, num_nodes);
property_map<graph_t, edge_weight_t>::type weightmap = get(edge_weight, g);

std::vector<vertex_descriptor> p(num_vertices(g));
std::vector<int> d(num_vertices(g));
vertex_descriptor s = vertex(A, g);

dijkstra_shortest_paths(g, s, predecessor_map(&p[0]).distance_map(&d[0]));

std::cout << "Расстояния и родители:" << std::endl;
graph_traits < graph_t >::vertex_iterator vi, vend;
```

продолжение ⇩

Листинг 13.3 (*продолжение*)

```
for (tie(vi, vend) = vertices(g); vi != vend; ++vi) {
  std::cout << "distance(" << name[*vi] << ") = " << d[*vi] << ", ";
  std::cout << "parent(" << name[*vi] << ") = " << name[p[*vi]] << std::
      endl;
}
std::cout << std::endl;
```

Программа выводит следующее:

```
Расстояния и родители:
distance(A) = 0, parent(A) = A
distance(B) = 6, parent(B) = E
distance(C) = 1, parent(C) = A
distance(D) = 4, parent(D) = C
distance(E) = 5, parent(E) = D
```

13.3.2. bellman_ford_shortest_paths

```
template <typename EdgeListGraph, typename Size,
, typename P, typename T, typename R>
bool bellman_ford_shortest_paths(EdgeListGraph& g, Size N,
  const bgl_named_params<P, T, R>& params)
```

Алгоритм Беллмана–Форда [5, 10, 13, 26] решает задачу кратчайших путей из одной вершины для графа с положительными и отрицательными весами ребер. Постановка задачи кратчайших путей приведена в разделе 5.1. Если вам нужно решать задачу кратчайших путей для положительных весов ребер, алгоритм Дейкстры предоставляет более эффективную альтернативу. Если веса всех ребер равны единице, поиск в ширину подходит еще больше.

Перед вызовом функции `bellman_ford_shortest_paths()` пользователь должен назначить исходной вершине расстояние ноль (или нейтральный элемент `Monoid`, состоящий из значений расстояния и функции `combine`), а всем другим вершинам — бесконечное расстояние (наибольшее значение расстояния согласно упорядочению, определяемому функциональным объектом `compare`). Обычно `std::numeric_limits<D>::max()` является подходящим выбором для бесконечности, где `D` — тип значения отображения расстояний. Алгоритм Беллмана–Форда выполняется в цикле по всем ребрам в графе, с применением операции релаксации к каждому ребру. В следующем псевдокоде v является вершиной, смежной с u, w отображает ребра на вес, d — отображение расстояний, которое записывает длину кратчайшего пути до каждой рассмотренной на данный момент вершины.

РЕЛАКСАЦИЯ (u, v, w, d)

если $(w(u, v) + d[u] < d[v])$

$d[v] \leftarrow w(u, v) + d[u]$

Алгоритм повторяет этот цикл $|V|$ раз, после чего гарантируется, что расстояния до каждой вершины были сокращены до минимально возможных, если в графе нет отрицательных циклов. Если отрицательный цикл присутствует, в графе будут ребра, которые не минимизированы до конца, то есть будут ребра (u, v), для которых $w(u, v) + d[u] < d[v]$. Алгоритм проходит циклом по всем вершинам еще раз, чтобы проверить, что все ребра были минимизированы, возвращая истину, если это так, и «ложь» в противном случае.

Имеется два основных способа получения результатов из функции `bellman_ford_shortest_paths()`. Если пользователь представляет отображение расстояний через параметр `distance_map()`, то кратчайшее расстояние из исходной вершины до каждой другой вершины в графе записывается в изображение расстояний (конечно, если функция возвращает истинное значение). Пользователь может также записать дерево кратчайших путей, задавая отображение свойства предшественников через параметр `predecessor_map()`. В дополнение к этим способам можно использовать своего собственного посетителя, который будет выполнять действия в любых событийных точках алгоритма (см. BellmanFordVisitor). Если вы заинтересованы только в некоторых событийных точках, создайте своего собственного посетителя из `default_bellman_visitor` для того, чтобы ненужные действия были заменены пустыми в не интересующих вас событийных точках.

Где определен

Алгоритм Беллмана–Форда находится в boost/graph/bellman_ford_shortest_paths.hpp.

Параметры

Ниже приведены параметры функции `bellman_ford_shortest_paths()`.

- `IN: EdgeListGraph& g`

  Ориентированный или неориентированный граф, который должен быть моделью EdgeListGraph.

- `IN: Size N`

  Количество вершин в графе. Тип `Size` должен быть целочисленным типом.

Именованные параметры

Именованные параметры функции `bellman_ford_shortest_paths()` приведены ниже.

- `IN: weight_map(WeightMap w)`

  Вес (также известный как «длина» или «стоимость») каждого ребра в графе. Тип `WeightMap` должен быть моделью ReadablePropertyMap. Типом ключа для этого отображения свойства должен быть дескриптор ребра графа. Тип значения для отображения весов такой же, как и тип значения отображения расстояний.

  По умолчанию: `get(edge_weight, g)`.

- `OUT: predecessor_map(PredecessorMap p_map)`

  Отображение предшественников записывает ребра минимального остовного дерева. По завершении алгоритма ребра $(\pi[u], u)$ $\forall u \in V$ находятся в минимальном остовном дереве. Если $\pi[u] = u$, то u является исходной вершиной или вершиной, которая недостижима из исходной. Тип `PredecessorMap` должен быть моделью концепции ReadWritePropertyMap, чьи типы ключа и значений такие же, как у дескриптора вершины графа.

  По умолчанию: `dummy_property_map`.

- `UTIL/OUT: distance_map(DistanceMap d_map)`

  Вес кратчайшего пути из исходной вершины s в каждую вершину графа записан в этом отображении свойства. Вес кратчайшего пути — это сумма весов

ребер, из которых состоит путь. Тип DistanceMap должен моделировать ReadWritePropertyMap. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения расстояний. Тип значения — элемент типа Monoid, состоящий из функционального объекта combine и объекта zero для нейтрального элемента. Также тип значения должен быть моделью StrictWeakOrdering (ослабленное строгое упорядочение), что обеспечивает объект-функция compare.

По умолчанию: get(vertex_distance, g).

- IN: visitor(BellmanFordVisitor v)

Объект-посетитель, который активизируется внутри алгоритма в событийных точках, указанных в концепции BellmanFordVisitor. Посетитель по умолчанию — default_bellman_visitor, не делает в событийных точках ничего.

По умолчанию: default_bellman_visitor.

- IN: distance_combine(BinaryFunction combine)

Объект-функция, являющийся операцией концепции Monoid для типа значения расстояния. Этот функциональный объект складывает длины для получения полной длины пути.

По умолчанию: closed_plus<D>, где D — тип значения отображения расстояний. closed_plus определен в boost/graph/relax.hpp.

- IN: distance_compare(BinaryPredicate compare)

Объект-функция, который задает StrictWeakOrdering на значениях расстояний. Используется для определения того, какой из путей короче.

По умолчанию: std::less<D>, где D — тип значения отображения расстояний.

Сложность

Временная сложность порядка $O(|V| \times |E|)$.

Пример

Исходный код примера из листинга 13.4 находится в файле example/bellman-example.cpp. Граф, использованный в примере, показан на рис. 13.4.

Листинг 13.4. Применение алгоритма Беллмана–Форда

```
enum { u, v, x, y, z, N };
char name[] = { 'u', 'v', 'x', 'y', 'z' };
typedef std::pair < int, int >E;
const int n_edges = 10;
E edge_array[] = { E(u, y), E(u, x), E(u, v), E(v, u),
  E(x, y), E(x, v), E(y, v), E(y, z), E(z, u), E(z,x) };
int weight[n_edges] = { -4, 8, 5, -2, 9, -3, 7, 2, 6, 7 };

typedef adjacency_list < vecS, vecS, directedS,
  no_property, property < edge_weight_t, int > > Graph;

Graph g(edge_array, edge_array + n_edges, N);

graph_traits < Graph >::edge_iterator ei, ei_end;
property_map<Graph, edge_weight_t>::type weight_pmap = get(edge_weight, g);
int i = 0;
```

```
for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei, ++i)
  weight_pmap[*ei] = weight[i];

std::vector<int> distance(N, std::numeric_limits < short >::max());
std::vector<std::size_t> parent(N);
for (i = 0; i < N; ++i)
  parent[i] = i;
distance[z] = 0;

bool r = bellman_ford_shortest_paths
  (g, int (N), weight_map(weight_pmap).distance_map(&distance[0]).
   predecessor_map(&parent[0]));

if (r)
  for (i = 0; i < N; ++i)
    std::cout << name[i] << ": " << std::setw(3) << distance[i]
      << " " << name[parent[i]] << std::endl;
    else
      std::cout << "отрицательный цикл" << std::endl;
```

Расстояние и предшественник для каждой вершины такие:

```
u:      2 v
v:      4 x
x:      7 z
y:     -2 u
z:      0 z
```

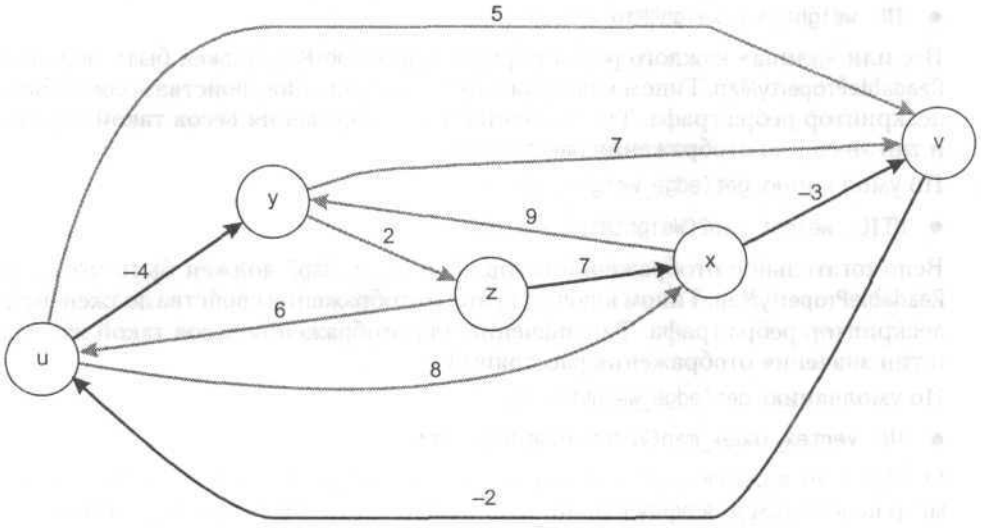


Рис. 13.4. Граф, использованный в примере алгоритма Беллмана–Форда

13.3.3. johnson_all_pairs_shortest_paths

```
template <typename Graph, typename DistanceMatrix,
  typename P, typename T, typename R>
bool johnson_all_pairs_shortest_paths(Graph& g, DistanceMatrix& D,
  const bgl_named_params<P, T, R>& params = all defaults)
```

Функция johnson_all_pairs_shortest_paths() (алгоритм Джонсона) находит кратчайшие расстояния между всеми парами вершин графа. Алгоритм возвращает false, если в графе присутствует отрицательный цикл, и true в противном случае. Расстояние между каждой парой вершин хранится в матрице расстояний D. Это один из наиболее требовательных ко времени графовых алгоритмов, так как он имеет временную сложность порядка $O(|V||E|\log|V|)$.

Где определен

Алгоритм Джонсона находится в boost/graph/johnson_all_pairs_shortest_paths.hpp.

Параметры

Параметры функции johnson_all_pairs_shortest_paths() приведены ниже.

- IN: const Graph& g

  Графовый объект, к которому применяется алгоритм. Тип Graph должен быть моделью VertexListGraph, IncidenceGraph и EdgeListGraph.

- OUT: DistanceMatrix& D

  Кратчайшая длина пути из вершины u в v хранится в $D[u][v]$.

Именованные параметры

Именованные параметры функции johnson_all_pairs_shortest_paths() приведены ниже.

- IN: weight_map(WeightMap w_map)

  Вес или «длина» каждого ребра в графе. Тип WeightMap должен быть моделью ReadablePropertyMap. Типом ключа для этого отображения свойства должен быть дескриптор ребра графа. Тип значения для отображения весов такой же, как и тип значения отображения расстояний.

  По умолчанию: get(edge_weight, g).

- UTIL: weight_map2(WeightMap2 w_map2)

  Вспомогательное отображение весов. Тип WeightMap2 должен быть моделью ReadablePropertyMap. Типом ключа для этого отображения свойства должен быть дескриптор ребра графа. Тип значения для отображения весов такой же, как и тип значения отображения расстояний.

  По умолчанию: get(edge_weight2, g).

- IN: vertex_index_map(VertexIndexMap i_map)

  Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим для эффективного обновления структуры данных при релаксации ребра. Тип VertexIndexMap должен моделировать ReadablePropertyMap. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения.

  По умолчанию: get(vertex_index, g).

- UTIL/OUT: distance_map(DistanceMap d_map)

  Вес кратчайшего пути из исходной вершины s в каждую вершину графа g записан в этом отображении свойства. Вес кратчайшего пути — это сумма весов

ребер, из которых состоит путь. Тип `DistanceMap` должен моделировать ReadWritePropertyMap. Тип дескриптора вершины графа должен задаваться в качестве типа ключа отображения расстояний. Тип значения — элемент типа Monoid, состоящий из операции сложения и объекта `zero` для нейтрального элемента. Также тип значения должен быть моделью LessThanComparable.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа, как тип значения `WeightMap`, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `IN: distance_zero(D zero)`

Значение должно быть нейтральным элементом для Monoid, со значением расстояний и операции сложения. Тип `D` — тип значения для `DistanceMap`.

По умолчанию: `D()`.

Сложность

Временная сложность порядка $O(|V||E|\log|V|)$.

Пример

Алгоритм Джонсона для кратчайших путей между всеми парами вершин (листинг 13.5) применен к графу со с. 568 «Introduction to Algorithms» [10], который также показан на рис. 13.5. Результирующая матрица `D[u][v]` дает кратчайший путь от вершины u к v.

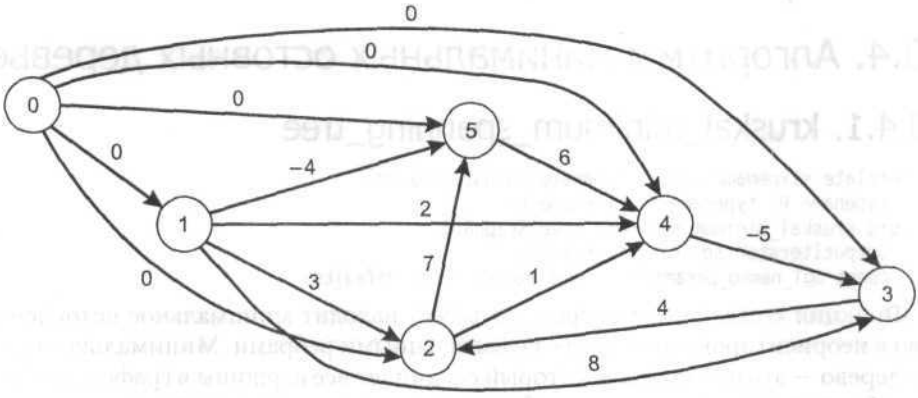


Рис. 13.5. Граф, использованный в примере алгоритма Джонсона

Листинг 13.5. Алгоритм Джонсона для кратчайших путей между всеми парами вершин

```
typedef adjacency_list<vecS, vecS, directedS, no_property,
  property< edge_weight_t, int, property< edge_weight2_t, int > > > Graph;
const int V = 6;
typedef std::pair < int, int >Edge;
  Edge edge_array[] =
    { Edge(0, 1), Edge(0, 4), Edge(0, 2), Edge(1, 3), Edge(1, 4),
    Edge(2, 1), Edge(3, 2), Edge(3, 0), Edge(4, 3)
  };
```

продолжение ⇩

Листинг 13.5 (*продолжение*)

```
const std::size_t E = sizeof(edge_array) / sizeof(Edge);

Graph g(edge_array, edge_array + E, V);

property_map < Graph, edge_weight_t >::type w = get(edge_weight, g);
int weights[] = { 3, -4, 8, 1, 7, 4, -5, 2, 6 };
int *wp = weights;

graph_traits < Graph >::edge_iterator e, e_end;
for (boost::tie(e, e_end) = edges(g); e != e_end; ++e)
  w[*e] = *wp++;

std::vector < int >d(V, std::numeric_limits < int >::max());
int D[V][V];

johnson_all_pairs_shortest_paths(g, D, distance_map(&d[0]));
```

Ниже приведена результирующая матрица расстояний:

	0	1	2	3	4	5
0	0	0	-1	-5	0	-4
1	inf	0	1	-3	2	-4
2	inf	3	0	-4	1	-1
3	inf	7	4	0	5	3
4	inf	2	-1	-5	0	-2
5	inf	8	5	1	6	0

13.4. Алгоритмы минимальных остовных деревьев

13.4.1. kruskal_minimum_spanning_tree

```
template <typename Graph, typename OutputIterator,
  typename P, typename T, typename R>
void kruskal_minimum_spanning_tree(Graph& g,
  OutputIterator spanning_tree_edges,
  const bgl_named_params<P, T, R>& params = all defaults)
```

Функция `kruskal_minimum_spanning_tree()` находит минимальное остовное дерево в неориентированном графе со взвешенными ребрами. Минимальное остовное дерево — это набор ребер, который соединяет все вершины в графе, где общий вес ребер дерева минимизирован. Функция `kruskal_minimum_spanning_tree()` выводит ребра остовного дерева итератором `spanning_tree_edges`, используя алгоритм Краскала [10, 18, 23, 44].

Алгоритм Краскала начинается с того, что каждая вершина является сама по себе деревом, без ребер из множества T, в котором будет строиться минимальное остовное дерево. Затем алгоритм рассматривает каждое ребро графа в порядке увеличения веса ребра. Если ребро соединяет две вершины в разных деревьях, алгоритм сливает эти деревья в одно и добавляет ребро к множеству T. Мы используем объединение по рангу и эвристики сжатия пути для обеспечения быстрой реализации операций над непересекающимися множествами (СДЕЛАТЬ-МНОЖЕСТВО, НАЙТИ-МНОЖЕСТВО и ОБЪЕДИНИТЬ-МНОЖЕСТВА). Ниже приведен псевдокод алгоритма Краскала.

КРАСКАЛ_МИН_ОСТ_ДЕРЕВО(G, w)
для каждой вершины $u \in V$
СДЕЛАТЬ_МНОЖЕСТВО(S, u)
$T \leftarrow \varnothing$
для каждого ребра $(u, v) \in E$ в порядке неуменьшения веса
если НАЙТИ_МНОЖЕСТВО(S, u) $\neq$ НАЙТИ_МНОЖЕСТВО(S, v)
ОБЪЕДИНИТЬ_МНОЖЕСТВА(S, u, v)
$T \leftarrow T \cup \{(u, v)\}$
возвратить T

Где определен

Алгоритм Краскала находится в boost/graph/kruskal_minimum_spanning_tree.hpp.

Параметры

Ниже приведены параметры функции `kruskal_minimum_spanning_tree()`.

- `IN: const Graph& g`

Неориентированный граф. Тип `Graph` должен быть моделью VertexListGraph и EdgeListGraph.

- `IN: OutputIterator spanning_tree_edges`

Ребра минимального остовного дерева выводятся этим `OutputIterator`.

Именованные параметры

Именованные параметры функции `kruskal_minimum_spanning_tree()` приведены ниже.

- `IN: weight_map(WeightMap w_map)`

Вес или «длина» каждого ребра в графе. Тип `WeightMap` должен быть моделью ReadablePropertyMap, и тип его значения — `LessThanComparable`. Типом ключа для этого отображения свойства должен быть дескриптор ребра графа.

По умолчанию: `get(edge_weight, g)`.

- `UTIL: rank_map(RankMap r_map)`

Тип `RankMap` должен быть моделью ReadWritePropertyMap. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения ранга. Тип значения отображения ранга должен быть целого типа.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами целого типа, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `UTIL: predecessor_map(PredecessorMap p_map)`

Тип `PredecessorMap` должен быть моделью ReadWritePropertyMap. Тип ключа и тип значения отображения предшественников должны быть типом дескриптора вершин графа.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами — дескрипторами вершин, размером `num_vertices(g)`. Он использует `i_map` для отображения индексов.

- `IN: vertex_index_map(VertexIndexMap i_map)`

Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим, если значения по умолчанию использованы для отображений ранга и предшественников. Тип `VertexIndexMap` должен моделировать `ReadablePropertyMap`. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения. По умолчанию: `get(vertex_index, g)`.

Сложность

Временная сложность порядка $O(|E| \log |E|)$.

Пример

Исходный код примера из листинга 13.6 находится в файле example/kruskal-example.cpp. Граф, использованный в этом примере, показан на рис. 13.6.

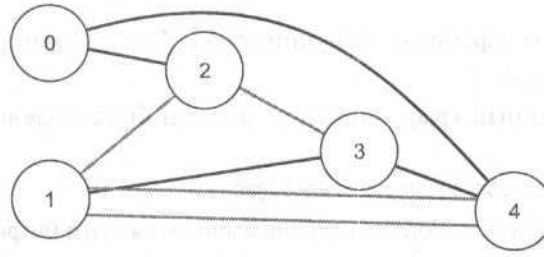


Рис. 13.6. Граф, использованный в примере алгоритма Краскала

Листинг 13.6. Пример использования алгоритма Краскала

```
typedef adjacency_list < vecS, vecS, undirectedS
  no_property, property < edge_weight_t, int > > Graph;
typedef graph_traits < Graph >::edge_descriptor Edge;
typedef graph_traits < Graph >::vertex_descriptor Vertex;
typedef std::pair<int, int> E;

const int num_nodes = 5;
E edge_array[] = { E(0, 2), E(1, 3), E(1, 4), E(2, 1), E(2, 3),
  E(3, 4), E(4, 0), E(4, 1)
};
int weights[] = { 1, 1, 2, 7, 3, 1, 1, 1 };
int num_edges = sizeof(edge_array) / sizeof(E);

Graph g(edge_array, edge_array + num_edges, weights, num_nodes);

property_map < Graph, edge_weight_t >::type weight = get(edge_weight, g);
std::vector < Edge > spanning_tree;

kruskal_minimum_spanning_tree(g, std::back_inserter(spanning_tree));

std::cout << "Ребра остовного дерева:" << std::endl;
for (std::vector < Edge >::iterator ei = spanning_tree.begin();
  ei != spanning_tree.end(); ++ei) {
    std::cout << source(*ei, g) << " <--> " << target(*ei, g)
    << " с весом " << weight[*ei] << std::endl;
}
```

Вывод этот программы следующий:

```
Ребра остовного дерева:
0 <--> 2 с весом 1
3 <--> 4 с весом 1
4 <--> 0 с весом 1
1 <--> 3 с весом 1
```

13.4.2. prim_minimum_spanning_tree

```
template <typename Graph, typename PredecessorMap,
  typename P, typename T, typename R>
void prim_minimum_spanning_tree(Graph& G, PredecessorMap p_map,
  const bgl_named_params<P, T, R>& params = all defaults)
```

Функция `prim_minimum_spanning_tree()` находит минимальное остовное дерево в неориентированном графе со взвешенными ребрами. Минимальное остовное дерево — это набор ребер, который соединяет все вершины в графе, где общий вес ребер дерева минимизирован. Функция `prim_minimum_spanning_tree()` выводит ребра остовного дерева в отображении предшественников: для каждой вершины $v \in V$, $p[v]$ является родителем v в вычисленном минимальном остовном дереве. Реализация использует алгоритм Прима [10, 18, 38, 44].

Алгоритм Прима наращивает минимальное остовное дерево по одной вершине за раз, что очень похоже на построение кратчайших путей алгоритмом Дейкстры[1]. На каждом шаге алгоритм выбирает ребро для добавления к минимальному остовному дереву. Это ребро — кратчайшее из всех, что соединяют любые вершины в дереве с вершинами вне дерева. Алгоритм использует очередь по приоритету для того, чтобы делать этот выбор эффективно. Если вершина u первая в очереди по приоритету, то ребро $(p[u], u)$ является следующим наикратчайшим ребром и добавляется к дереву. Псевдокод этого алгоритма приведен ниже.

ПРИМ_МИН_ОСТ_ДЕРЕВО(G, r, w)

для каждой вершины $u \in V$ ▷ инициализировать вершину u
$d[u] \leftarrow \infty$
$\pi[u] \leftarrow u$
$color[u] \leftarrow$ *БЕЛЫЙ*
$color[r] \leftarrow$ *СЕРЫЙ*
$d[r] \leftarrow 0$
В_ОЧЕРЕДЬ(Q, r) ▷ посетить вершину r
пока $Q \neq \varnothing$
$u \leftarrow$ ИЗ_ОЧЕРЕДИ(Q) ▷ рассмотреть вершину u
для каждой вершины $v \in Adj[u]$ ▷ рассмотреть ребро (u, v)
если ($w[u, v] < d[v]$)

[1] Фактически реализация BGL алгоритма Прима — это вызов алгоритма Дейкстры с особенными аргументами для параметров distance_compare() и distance_combine().

$d[v] \leftarrow w[u, v]$ ▷ релаксация ребра (u, v)
$\pi[v] \leftarrow u$
если ($color[v] = БЕЛЫЙ$)
$color[v] \leftarrow СЕРЫЙ$
В_ОЧЕРЕДЬ(Q, v) ▷ посетить вершину v
иначе если ($color[v] = СЕРЫЙ$)
ОБНОВИТЬ(PQ, v)
иначе
... ▷ ребро (u, v) не релаксируется
$color[u] \leftarrow ЧЕРНЫЙ$ ▷ завершение обработки вершины u
возвратить (π)

Где определен

Алгоритм Прима находится в boost/graph/prim_minimum_spanning_tree.hpp.

Параметры

Ниже приведены параметры функции `prim_minimum_spanning_tree()`.

- `IN: const Graph& g`

  Графовый объект, к которому применяется алгоритм. Тип `Graph` должен быть моделью VertexListGraph и IncidenceGraph.

- `OUT: PredecessorMap p_map`

  Отображение предшественников записывает ребра минимального остовного дерева. По завершении алгоритма ребра $(\pi[u], u)$ $\forall u \in V$ находятся в минимальном остовном дереве. Если $\pi[u] = u$, то u является исходной вершиной или вершиной, которая недостижима из исходной. Тип `PredecessorMap` должен быть моделью концепции ReadWritePropertyMap, чьи типы ключа и значений такие же, как у дескриптора вершины графа.

Именованные параметры

Именованные параметры функции `prim_minimum_spanning_tree()` приведены ниже.

- `IN: root_vertex(vertex_descriptor r)`

  Вершина, которая будет корнем минимального остовного дерева. Выбор корневой вершины произволен, он не влияет на способность алгоритма находить минимальное остовное дерево.

  По умолчанию: `*vertices(g).first`.

- `IN: weight_map(WeightMap w_map)`

  Вес или «длина» каждого ребра в графе. Тип `WeightMap` должен быть моделью ReadablePropertyMap. Типом ключа для этого отображения свойства должен быть дескриптор ребра графа. Тип значения для отображения весов такой же, как и тип значения отображения расстояний.

  По умолчанию: `get(edge_weight, g)`.

- `IN: vertex_index_map(VertexIndexMap i_map)`

Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим для эффективного обновления структуры данных на куче при релаксации ребра. Тип `VertexIndexMap` должен моделировать `ReadablePropertyMap`. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа должен быть задан в качестве типа ключа отображения.

По умолчанию: `get(vertex_index, g)`.

- `UTIL: distance_map(DistanceMap d_map)`

Вес кратчайшего пути из исходной вершины `s` в каждую вершину графа `g` записан в этом отображении свойства. Вес кратчайшего пути — это сумма весов ребер, из которых состоит путь. Тип `DistanceMap` должен моделировать `ReadWritePropertyMap`. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения расстояний. Тип значения — элемент типа `Monoid`, состоящий из операции сложения и объекта `zero` для нейтрального элемента (строимого по умолчанию). Также тип значения должен быть моделью `LessThanComparable`.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа, как тип значения `WeightMap`, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `UTIL/OUT: color_map(ColorMap c_map)`

Используется во время выполнения алгоритма для маркировки вершин. В начале вершины имеют белый цвет и становятся серыми, когда попадают в очередь. Они перекрашиваются в черный при удалении из очереди. В конце алгоритма вершины, достижимые из исходной, имеют черный цвет. Все остальные вершины будут белыми. Тип `ColorMap` должен быть моделью `ReadWritePropertyMap`. Дескриптор вершины должен быть задан в качестве типа ключа для отображения, и тип значения отображения должен моделировать `ColorValue`.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа `default_color_type`, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

Сложность

Временная сложность порядка $O(|E| \log |V|)$.

Пример

Исходный код примера из листинга 13.7 находится в файле example/prim-example.cpp.

Листинг 13.7. Пример использования алгоритма Прима

```
typedef adjacency_list < vecS, vecS, undirectedS,
  property<vertex_distance_t, int>, property < edge_weight_t, int > > Graph;
typedef std::pair < int, int >E;
const int num_nodes = 5;
E edges[] = { E(0, 2), E(1,1), E(1, 3), E(1, 4), E(2, 1), E(2, 3),
  E(3, 4), E(4, 0)
};
int weights[] = { 1, 2, 1, 2, 7, 3, 1, 1 };

Graph g(edges, edges + sizeof(edges) / sizeof(E), weights, num_nodes);
```

продолжение ⇩

Листинг 13.7 (*продолжение*)

```
property_map<Graph, edge_weight_t>::type weightmap = get(edge_weight, g);

std::vector < graph_traits < Graph >::vertex_descriptor >
  p(num_vertices(g));

prim_minimum_spanning_tree(g, &p[0]);

for (std::size_t i = 0; i != p.size(); ++i)
  if (p[i] != i)
    std::cout << "parent[" << i << "] = " << p[i] << std::endl;
  else
    std::cout << "parent[" << i << "] = no parent" << std::endl;
```

Вывод этой программы такой:

```
parent[0] = 0
parent[1] = 3
parent[2] = 0
parent[3] = 4
parent[4] = 0
```

13.5. Статические компоненты связности

13.5.1. connected_components

```
template <typename Graph, typename ComponentMap,
  typename P, typename T, typename R>
typename property_traits<ComponentMap>::value_type
connected_components(const Graph& g, ComponentMap c,
  const bgl_named_params<P, T, R>& params = all_defaults)
```

Функция `connected_components()` вычисляет компоненты связности неориентированного графа, используя подход, основанный на поиске в глубину. *Компонента связности* — это группа вершин неориентированного графа, в которой каждая вершина достижима из любой другой. Если компоненты связности нужно вычислять для растущего графа, то метод, основанный на непересекающихся множествах (функция `incremental_components()`), является более быстрым. Для статических графов поиск в глубину быстрее [10].

Результат алгоритма записывается в отображение свойства компоненты `c`, которое содержит номера компонент для каждой вершины. Полное число компонент — возвращаемое значение этой функции.

Где определен

Алгоритм связанных компонент находится в boost/graph/connected_components.hpp.

Параметры

Ниже приведены параметры функции `connected_components()`.

- `IN: const Graph& g`

  Неориентированный граф. Тип графа должен быть моделью VertexListGraph и IncidenceGraph.

- `OUT: ComponentMap c`

Алгоритм вычисляет, сколько компонент связности находятся в графе, и присваивает каждой компоненте целочисленную метку. Затем алгоритм записывает, какой компоненте принадлежит каждая вершина графа, в отображение свойства компонент. Тип `ComponentMap` должен быть моделью **WritablePropertyMap**. Тип значения должен быть `vertices_size_type` графа. Тип ключа — тип дескриптора вершины графа.

Именованные параметры

Ниже приведены именованные параметры функции `connected_components()`.

- `UTIL: color_map(ColorMap color)`

Используется алгоритмом для отслеживания продвижения по графу. Тип `ColorMap` должен быть моделью **ReadWritePropertyMap**, тип ключа — дескриптор вершины графа, тип значения `color_map` должен быть моделью **ColorValue**.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` с элементами типа `default_color_type`, размером `num_vertices()`. Он использует `i_map` для отображения индексов.

- `IN: vertex_index_map(VertexIndexMap i_map)`

Отображает каждую вершину в целое число из диапазона $[0, |V|)$. Этот параметр необходим, только когда используется отображение свойства цветовой окраски по умолчанию. Тип `VertexIndexMap` должен моделировать **ReadablePropertyMap**. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа должен быть задан в качестве типа ключа отображения.

По умолчанию: `get(vertex_index, g)`.

Сложность

Временная сложность для сильных компонент связности порядка $O(|V| + |E|)$. Временная сложность для компонент связности того же порядка.

Пример

Вычисление компонент связности неориентированного графа приведено в листинге 13.8.

Листинг 13.8. Вычисление компонент связности неориентированного графа

```
⟨ connected-components.cpp ⟩ ≡
#include <boost/config.hpp>
#include <iostream>
#include <vector>
#include <boost/graph/connected_components.hpp>
#include <boost/graph/adjacency_list.hpp>

int main() {
  using namespace boost;
  typedef adjacency_list < vecS, vecS, undirectedS > Graph;
  typedef graph_traits < Graph >::vertex_descriptor Vertex;

  const int N = 6;
  Graph G(N);
  add_edge(0, 1, G);
```

продолжение ⇩

Листинг 13.8 (*продолжение*)

```
  add_edge(1, 4, G);
  add_edge(4, 0, G);
  add_edge(2, 5, G);

  std::vector<int> c(num_vertices(G));
  int num = connected_components
    (G, make_iterator_property_map(c.begin(), get(vertex_index, G), c[0]));

  std::cout << std::endl;
  std::vector < int >::iterator i;
  std::cout << "Общее число компонент: " << num << std::endl;
  for (i = c.begin(); i != c.end(); ++i)
    std::cout << "Вершина " << i - c.begin()
      << " в компоненте " << *i << std::endl;
  std::cout << std::endl;
  return EXIT_SUCCESS;
}
```

Вывод будет следующим:

```
Общее число компонент: 3
Вершина 0 в компоненте 0
Вершина 1 в компоненте 0
Вершина 2 в компоненте 1
Вершина 3 в компоненте 2
Вершина 4 в компоненте 0
Вершина 5 в компоненте 1
```

13.5.2. strong_components

```
template <class Graph, class ComponentMap, class P, class T, class R>
typename property_traits<ComponentMap>::value_type
strong_components(Graph& g, ComponentMap comp,
  const bgl_named_params<P, T, R>& params = all defaults)
```

Функция `strong_components()` вычисляет сильные компоненты связности ориентированного графа с использованием алгоритма Тарьяна, который основан на поиске в глубину [43].

Результат алгоритма записывается в отображение свойства компонент `comp`, которое содержит номер компоненты для каждой вершины. Идентификационные номера изменяются от нуля до числа компонент в графе минус один. Полное число компонент — возвращаемое значение этой функции.

Где определен

Алгоритм Тарьяна находится в `boost/graph/strong_components.hpp`.

Определения

Сильная компонента связности ориентированного графа $G = (V, E)$ — максимальный набор вершин $U \subseteq V$, такой, что для каждой пары вершин u и v в U мы имеем как путь из u в v, так и путь из v в u, то есть u и v достижимы друг из друга.

Ниже дано неформальное описание алгоритма Тарьяна для вычисления сильных компонент связности. В основном это вариация поиска в глубину с дополнительными действиями в событийных точках «посещение вершины» и «окончание обработки вершины». Можно думать о действиях в точке «посещение вершины»

как о выполняемых «на пути вниз» по дереву поиска в глубину (от корня к листьям), а о действиях в точке «окончание обработки вершины» — как о выполняемых «на пути вверх».

Три вещи должно произойти по пути вниз. Для каждой посещенной вершины u мы записываем отметку посещения $d[u]$, помещаем u во вспомогательный стек и присваиваем $root[u] = u$. Поле *root* в итоге будет отображать каждую вершину на самую верхнюю вершину в той же сильной компоненте связности. Устанавливая $root[u] = u$, мы начинаем с ситуации, когда каждая вершина — компонента сама по себе.

Опишем теперь то, что происходит по пути вверх. Предположим, что мы только что закончили посещение всех вершин, смежных с некоторой вершиной u. Теперь можно рассмотреть каждую из смежных вершин снова, проверяя корень каждой из них. Корень, имеющий наименьшее значение метки посещения, назовем корнем a. Затем мы сравниваем a с вершиной u и рассматриваем следующие случаи:

1. Если $d[a] < d[u]$, то мы знаем, что a — предок u в дереве поиска в глубину, и значит, мы имеем цикл и u должна быть в одной сильной компоненте связности с a. Тогда мы присваиваем $root[u] = a$ и продолжаем путь назад, вверх по дереву поиска в глубину.
2. Если $a = u$, то мы знаем, что u должна быть самой верхней вершиной поддерева, определяющего сильные компоненты связности. Все вершины в этом поддереве расположены ниже в стеке по отношению к вершине u, так что мы выталкиваем вершины из стека, пока не достигнем u, и отмечаем каждую как принадлежащую той же самой компоненте.
3. Если $d[a] > d[u]$, то смежные вершины находятся в разных сильных компонентах связности. Мы продолжаем путь назад вверх по дереву поиска в глубину.

Параметры

Ниже приведены параметры функции `strong_components()`.

- `IN: const Graph& g`

  Ориентированный граф. Тип графа должен быть моделью VertexListGraph и IncidenceGraph.

- `OUT: ComponentMap comp`

  Алгоритм вычисляет количество компонент связности в графе и присваивает каждой компоненте целочисленную метку. Затем алгоритм записывает, какой компоненте связности принадлежит каждая вершина графа, записывая номер компоненты в отображение свойства компонент. Тип `ComponentMap` должен быть моделью WritablePropertyMap. Тип значения требуется задавать целым числом, лучше таким же, как `vertices_size_type` графа. Тип ключа должен быть типом дескриптора вершины.

Именованные параметры

Ниже приведены именованные параметры функции `strong_components()`.

- `UTIL: root_map(RootMap r_map)`

  Используется алгоритмом для записи кандидатов в корневые вершины для каждой вершины. В конце работы алгоритма у каждой сильной компоненты связности всего одна корневая вершина и `get(r_map, v)` возвращает корневую

вершину для той компоненты, в которую входит v. RootMap должен быть ReadWritePropertyMap, где тип ключа и тип значения должны совпадать с типом дескриптора вершины в графе.

По умолчанию: iterator_property_map, созданный из вектора std::vector описателей вершин, размером num_vertices(g). Использует i_map для отображения индексов.

- UTIL: discover_time(TimeMap t_map)

Используется алгоритмом для отслеживания упорядочения вершин поиска в глубину. TimeMap должен быть моделью ReadWritePropertyMap и его тип значения должен быть целым типом. Тип ключа должен быть дескриптором вершины графа.

По умолчанию: iterator_property_map, созданный из вектора std::vector целых чисел, размером num_vertices(g). Использует i_map для отображения индексов.

- UTIL: color_map(ColorMap c_map)

Используется алгоритмом для отслеживания продвижения по графу. Тип ColorMap должен быть моделью ReadWritePropertyMap, тип ключа — дескриптор вершины графа, тип значения color_map должен быть моделью ColorValue.

По умолчанию: iterator_property_map, созданный из вектора std::vector с элементами типа default_color_type, размером num_vertices(). Он использует i_map для отображения индексов.

- IN: vertex_index_map(VertexIndexMap i_map)

Отображает каждую вершину в целое число из диапазона $[0, N)$, где N — число вершин графа. Этот параметр необходим, только когда используется значение по умолчанию для одного из других именованных параметров. Тип VertexIndexMap должен моделировать ReadablePropertyMap. Тип значения отображения должен быть целочисленным типом. Тип дескриптора вершины графа требуется задавать в качестве типа ключа отображения.

По умолчанию: get(vertex_index, g).

Сложность

Временная сложность для алгоритма нахождения сильных компонент связности порядка $O(|V| + |E|)$.

Смотри также

Для дополнительной информации смотрите connected_components() и incremental_components().

Пример

Вычисление сильных компонент связности для ориентированного графа приведено в листинге 13.9.

Листинг 13.9. Вычисление сильных компонент связности для ориентированного графа

```
⟨ strong-components.cpp ⟩ ≡
#include <boost/config.hpp>
#include <vector>
#include <iostream>
```

```
#include <boost/graph/strong_components.hpp>
#include <boost/graph/adjacency_list.hpp>

int main() {
  using namespace boost;
  typedef adjacency_list < vecS, vecS, directedS > Graph;
  const int N = 6;
  Graph G(N);
  add_edge(0, 1, G);  add_edge(1, 1, G);  add_edge(1, 3, G);
  add_edge(1, 4, G);  add_edge(3, 4, G);  add_edge(3, 0, G);
  add_edge(4, 3, G);  add_edge(5, 2, G);

  std::vector<int> c(N);
  int num = strong_components
    (G, make_iterator_property_map(c.begin(), get(vertex_index, G), c[0]));

  std::cout << "Общее число компонент: " << num << std::endl;
  std::vector < int >::iterator i;
  for (i = c.begin(); i != c.end(); ++i)
    std::cout << "Вершина " << i - c.begin()
      << " в компоненте " << *i << std::endl;
  return EXIT_SUCCESS;
}
```

Программа выводит следующее:

```
Общее число компонент: 3
Вершина 0 в компоненте 0
Вершина 1 в компоненте 0
Вершина 2 в компоненте 1
Вершина 3 в компоненте 0
Вершина 4 в компоненте 0
Вершина 5 в компоненте 2
```

13.6. Растущие компоненты связности

Этот раздел описывает семейство функций и классов, которые вычисляют компоненты связности неориентированного графа. Алгоритм, который используется здесь, основан на структуре данных для представления непересекающихся множеств (disjoint-sets) [10, 44], которая является лучшим средством для ситуаций, когда граф растет (к нему добавляются ребра) и информация о компонентах связности нуждается в постоянном обновлении. Класс непересекающихся множеств (НМ) описан в разделе 16.6.

Следующие операции являются основными функциями для вычисления и поддержания компонент связности. Использованные здесь объекты — граф `g`, НМ-объект `ds` и вершины `u` и `v`.

- `initialize_incremental_components(g,ds)`

  Основная инициализация структуры непересекающихся множеств. Каждая вершина графа `g` в своем собственном множестве.

- `incremental_components(g,ds)`

  Компоненты связности вычисляются на основе ребер в графе `g` и информации, включенной в НМ-объект `ds`.

- `ds.find_set(v)`

  Выдает информацию о компоненте для вершины `v` из НМ-объекта.

- `ds.union_set(u,v)`

  Обновляет НМ-объект, когда ребро (u, v) добавляется к графу.

Сложность

Временная сложность для всего процесса порядка $O(|V| + |E|\alpha(|E|, |V|))$, где $|E|$ — полное число ребер в графе (в конце процесса) и $|V|$ — число вершин, α — функция, обратная функции Аккермана. Последняя имеет взрывной рекурсивно-экспоненциальный рост. Значит, обратная ей функция растет крайне медленно. На практике $\alpha(m, n) \leq 4$, что означает временную сложность немного больше, чем $O(|V| + |E|)$.

Пример

В примере из листинга 13.10 мы поддерживаем компоненты связности графа при добавлении ребер, используя структуру данных непересекающихся множеств. Полный исходный код этого примера находится в файле example/incremental-components-eg.cpp.

Листинг 13.10. Пример увеличивающихся компонент связности

```
// Создать граф
typedef adjacency_list < vecS, vecS, undirectedS > Graph;
typedef graph_traits < Graph >::vertex_descriptor Vertex;
const int N = 6;
Graph G(N);
add_edge(0, 1, G);
add_edge(1, 4, G);
// создать НМ-объект, для которого нужны свойства ранга и родителя вершины
std::vector < Vertex > rank(num_vertices(G));
std::vector < Vertex > parent(num_vertices(G));
typedef graph_traits<Graph>::vertices_size_type* Rank;
typedef Vertex* Parent;
disjoint_sets < Rank, Parent > ds(&rank[0], &parent[0]);

// определить компоненты связности, сохраняя результат в НМ-объекте
initialize_incremental_components(G, ds);
incremental_components(G, ds);

// Добавить еще пару вершин и обновить непересекающиеся множества
graph_traits < Graph >::edge_descriptor e;
bool flag;
tie(e, flag) = add_edge(4, 0, G);
ds.union_set(4, 0);
tie(e, flag) = add_edge(2, 5, G);
ds.union_set(2, 5);

graph_traits < Graph >::vertex_iterator iter, end;
for (tie(iter, end) = vertices(G); iter != end; ++iter)
  std::cout << "представитель[" << *iter << "] = " <<
    ds.find_set(*iter) << std::endl;
std::cout << std::endl;

typedef component_index < unsigned int >Components;
Components components(parent.begin(), parent.end());
for (Components::size_type i = 0; i < components.size(); ++i) {
  std::cout << "компонента " << i << " содержит: ";
```

```
  for (Components::value_type::iterator j = components[i].begin();
       j != components[i].end(); ++j)
    std::cout << *j << " ";
  std::cout << std::endl;
}
```

Вывод будет следующим:

```
представитель[0] = 1
представитель[1] = 1
представитель[2] = 5
представитель[3] = 3
представитель[4] = 1
представитель[5] = 5

компонента 0 содержит: 4 1 0
компонента 1 содержит: 3
компонента 2 содержит: 5 2
```

Где определен

Все функции этого раздела определены в boost/graph/incremental_components.hpp.

13.6.1. initialize_incremental_components

```
template <typename VertexListGraph, typename DisjointSets>
void initialize_incremental_components(VertexListGraph& G, DisjointSets& ds)
```

Функция initialize_incremental_components() инициализирует структуру непересекающихся множеств для алгоритма увеличивающихся компонент связности, делая каждую вершину неориентированного графа членом своей собственной компоненты.

Сложность

Временная сложность порядка $O(|V|)$.

13.6.2. incremental_components

```
template <typename EdgeListGraph, typename DisjointSets>
void incremental_components(EdgeListGraph& g, DisjointSets& ds)
```

Функция incremental_components() вычисляет компоненты связности неориентированного графа, включая результаты в НМ-структуру данных.

Сложность

Временная сложность порядка $O(|E|)$.

13.6.3. same_component

```
template <typename Vertex, typename DisjointSets>
bool same_component(Vertex u, Vertex v, DisjointSets& ds)
```

Функция same_component() определяет, находятся ли u и v в той же самой компоненте.

Сложность

Временная сложность — $O(\alpha(|E|, |V|))$.

13.6.4. component_index

```
component_index<Index>
```

Класс component_index обеспечивает представление данных для компонент графа, подобное контейнеру STL. Каждая компонента является контейнероподобным объектом, и объект component_index обеспечивает доступ к объектам компоненты через operator[]. Объект component_index инициализируется свойством родителей в непересекающихся множествах, вычисленных функцией incremental_components().

Параметры шаблона

- Index — целый беззнаковый тип, используемый для подсчета компонент.

Где определен

Класс component_index находится в boost/graph/incremental_components.hpp.

Ассоциированные типы

Ниже приведены ассоциированные типы класса component_index.

- component_index::value_type

Тип для объекта-компоненты. Тип компоненты имеет следующие члены.

- component_index::size_type

Тип, используемый для представления числа компонент.

Функции — методы класса

Ниже приведены функции — методы класса component_index.

- template <typename ComponentsContainer>
  component_index::component_index(const ComponentsContainer& c)

Создает component_index, используя информацию из контейнера компонент c, который был результатом выполнения incremental_components.

- template <typename ParentIterator>
  component_index::component_index(ParentIterator first, ParentIterator last)

Создает индекс компонент из «родителей», вычисленных функцией incremental_components().

- value_type component_index::operator[ ](size_type i) const

Возвращает i-ю компоненту графа.

- size_type component_index::size() const

Возвращает число компонент графа.

Ассоциированные типы для компоненты

Тип value_type для component_index — компонента, которая имеет следующие ассоциированные типы.

- value_type::value_type

Тип значения для объекта-компоненты — идентификатор вершины.

- value_type::iterator

- `value_type::const_iterator`

Этот итератор может быть применен для обхода всех вершин компоненты. Данный итератор разыменовывается в идентификатор вершины.

Функции — методы класса компоненты

Тип `value_type` для `component_index` является представлением компоненты и имеет следующие функции — методы класса.

- `iterator begin() const`

Возвращает итератор, указывающий на первую вершину компоненты.

- `iterator end() const`

Возвращает итератор, указывающий на последнюю вершину компоненты.

13.7. Алгоритмы максимального потока

13.7.1. edmunds_karp_max_flow

```
template <typename Graph, typename P, typename T, typename R>
typename detail::edge_capacity_value<Graph, P, T, R>::type
edmunds_karp_max_flow(Graph& g,
  typename graph_traits<Graph>::vertex_descriptor src,
  typename graph_traits<Graph>::vertex_descriptor sink,
  const bgl_named_params<P, T, R>& params = all_defaults)
```

Функция `edmunds_karp_max_flow()` вычисляет максимальный поток в сети (см. главу 8). Максимальный поток является возвращаемым значением этой функции. Функция также вычисляет значения потока $f(u, v)$ $\forall (u, v) \in E$, которые возвращаются в форме остаточной мощности $r(u, v) = c(u, v) - f(u, v)$.

Где определен

Алгоритм находится в boost/graph/edmunds_karp_max_flow.hpp.

Параметры

Ниже приведены параметры функции `edmunds_karp_max_flow()`.

- `IN: Graph& g`

Ориентированный граф. Тип графа должен быть моделью VertexListGraph и IncidenceGraph. Для каждого ребра (u, v) обратное ребро (v, u) также должно быть в графе.

- `IN: vertex_descriptor src`

Исходная вершина для графа потоковой сети.

- `IN: vertex_descriptor sink`

Сток для графа потоковой сети.

Именованные параметры

Ниже приведены именованные параметры функции `edmunds_karp_max_flow()`.

- `IN: capacity_map(CapacityEdgeMap cap)`

Отображение свойства мощности ребра. Тип должен быть моделью константной LvaluePropertyMap. Тип ключа отображения должен быть типом дескриптора ребра графа.

По умолчанию: `get(edge_capacity, g)`.

- `OUT: residual_capacity_map(ResidualCapacityEdgeMap res)`

Отображение свойства остаточной мощности ребра. Тип должен быть моделью изменяемой LvaluePropertyMap. Тип ключа отображения должен быть типом дескриптора ребра графа.

По умолчанию: `get(edge_residual_capacity, g)`.

- `IN: reverse_edge_map(ReverseEdgeMap rev)`

Отображение свойства ребра, которое отображает каждое ребро графа (u, v) в обратное ребро (v, u). Это отображение должно быть моделью константной LvaluePropertyMap. Тип ключа отображения должен быть типом дескриптора ребра графа.

По умолчанию: `get(edge_reverse, g)`.

- `UTIL: predecessor_map(PredecessorMap p_map)`

Это отображение предшественников отличается от обычного отображения предшественников тем, что тип значения является типом дескриптора ребра, а не вершины. Тип ключа для этого отображения — дескриптор вершины.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` описателей ребер, размером `num_vertices(g)`. В качестве индекса отображения используется `i_map`.

- `UTIL: color_map(ColorMap c_map)`

Это отображение используется во внутренних вычислениях. Тип `ColorMap` должен быть моделью ReadWritePropertyMap. Дескриптор вершины требуется задавать в качестве типа ключа отображения, и тип значения отображения должен быть моделью ColorValue.

По умолчанию: `iterator_property_map`, созданный из вектора `std::vector` элементов типа `default_color_type`, размером `num_vertices(g)`. В качестве индекса отображения используется `i_map`.

- `IN: vertex_index_map(VertexIndexMap index_map)`

Это отображение необходимо, если применялось отображение цветов по умолчанию или отображение предшественников по умолчанию. Каждая вершина графа отображается в целое число из диапазона $[0, |V|)$. Отображение должно быть моделью константной LvaluePropertyMap. Тип ключа отображения должен совпадать с типом дескриптора вершины графа.

По умолчанию: `get(vertex_index, g)`.

Пример

Программа в листинге 13.11 читает пример задачи максимального потока (граф с мощностями ребер) из файла в формате DIMACS [1].

Листинг 13.11. Решение задачи максимального потока

```
⟨ edmunds-karp-eg.cpp ⟩ ≡
#include <boost/config.hpp>
```

```
#include <iostream>
#include <string>
#include <boost/graph/edmunds_karp_max_flow.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/read_dimacs.hpp>
#include <boost/graph/graph_utility.hpp>

int main() {
  using namespace boost;

  typedef adjacency_list_traits < vecS, vecS, directedS > Traits;
  typedef adjacency_list < listS, vecS, directedS,
    property < vertex_name_t, std::string >,
    property < edge_capacity_t, long,
    property < edge_residual_capacity_t, long,
    property < edge_reverse_t, Traits::edge_descriptor > > > > Graph;

  Graph g;

  property_map < Graph, edge_capacity_t >::type
    capacity = get(edge_capacity, g);
  property_map < Graph, edge_reverse_t >::type rev = get(edge_reverse, g);
  property_map < Graph, edge_residual_capacity_t >::type
    residual_capacity = get(edge_residual_capacity, g);

  Traits::vertex_descriptor s, t;
  read_dimacs_max_flow(g, capacity, rev, s, t);

  long flow = edmunds_karp_max_flow(g, s, t);

  std::cout << "c  Полный поток:" << std::endl;
  std::cout << "s " << flow << std::endl << std::endl;

  std::cout << "c значения потока:" << std::endl;
  graph_traits < Graph >::vertex_iterator u_iter, u_end;
  graph_traits < Graph >::out_edge_iterator ei, e_end;
  for (tie(u_iter, u_end) = vertices(g); u_iter != u_end; ++u_iter)
    for (tie(ei, e_end) = out_edges(*u_iter, g); ei != e_end; ++ei)
      if (capacity[*ei] > 0)
        std::cout << "f " << *u_iter << " " << target(*ei, g) << " "
          << (capacity[*ei] - residual_capacity[*ei]) << std::endl;

  return EXIT_SUCCESS;
}
```

Программа выводит следующее:

```
c  Полный поток:
s 13

c значения потока:
f 0 6 3
f 0 1 6
f 0 2 4
f 1 5 1
f 1 0 0
f 1 3 5
f 2 4 4
f 2 3 0
f 2 0 0
f 3 7 5
f 3 2 0
```

```
f 3 1 0
f 4 5 4
f 4 6 0
f 5 4 0
f 5 7 5
f 6 7 3
f 6 4 0
f 7 6 0
f 7 5 0
```

13.7.2. push_relabel_max_flow

```
template <typename Graph, typename P, typename T, typename R>
typename detail::edge_capacity_value<Graph, P, T, R>::type
push_relabel_max_flow(Graph& g,
  typename graph_traits<Graph>::vertex_descriptor src,
  typename graph_traits<Graph>::vertex_descriptor sink,
  const bgl_named_params<P, T, R>& params)
```

Функция `push_relabel_max_flow()` вычисляет максимальный поток сети (см. главу 8). Максимальный поток является возвращаемым значением этой функции. Функция также вычисляет значения потока $f(u, v)\ \forall(u, v) \in E$, которые возвращаются в форме остаточной мощности $r(u, v) = c(u, v) - f(u, v)$. Сеть с ребрами, отмеченными значениями потока и мощности, изображена на рис. 13.7.

Имеется несколько особых требований к входному графу и параметрам отображений свойств для этого алгоритма. Во-первых, ориентированный граф $G = (V, E)$, представляющий сеть, должен быть расширен так, чтобы для любого ребра из E в него вошло также и обратное ребро. То есть входной граф должен быть $G_{in}(V,\{E \cup E^T\})$. Аргумент `rev` для `ReverseEdgeMap` должен отображать каждое ребро в исходном графе на обратное ребро, то есть $(u, v) \rightarrow (v, u)\ \forall(u, v) \in E$. Аргумент `cap` от `CapacityEdgeMap` должен отображать каждое ребро в E на положительное число, а каждое ребро в E^T в 0. Другими словами, отображение мощности должно удовлетворять этим ограничениям: $c(u, v) > 0$ и $c(v, u) = 0$ для каждого $(u, v) \in E$.

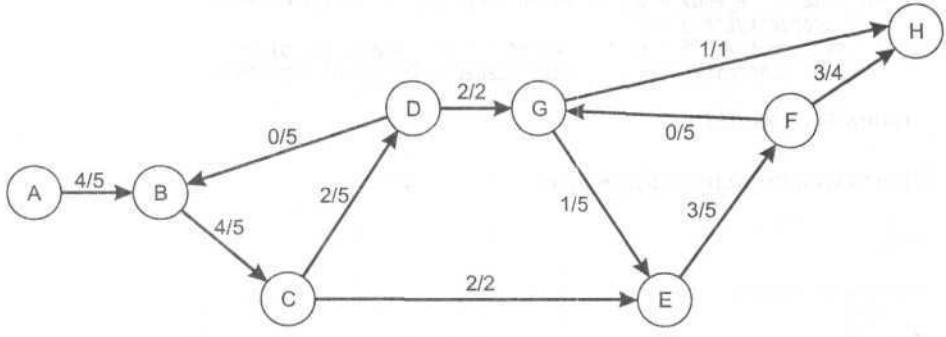


Рис. 13.7. Потоковая сеть, в которой ребра помечены значениями потока и мощности

Где определен

Алгоритм находится в boost/graph/push_relabel_max_flow.hpp.

Параметры

Ниже приведены параметры функции `push_relabel_max_flow()`.

- `IN: Graph& g`

Ориентированный граф. Графовый тип должен быть моделью VertexListGraph и IncidenceGraph. Для каждого ребра (u, v) обратное ребро (v, u) также должно быть в графе.

- `IN: vertex_descriptor src`

Исходная вершина для графа потоковой сети.

- `IN: vertex_descriptor sink`

Сток для графа потоковой сети.

Именованные параметры

Ниже приведены именованные параметры функции `push_relabel_max_flow()`.

- `IN: capacity_map(CapacityEdgeMap cap)`

Отображение свойства реберной мощности. Тип должен быть моделью константной LvaluePropertyMap. Тип ключа отображения должен быть дескриптором ребра графа.

По умолчанию: `get(edge_capacity, g)`.

- `OUT: residual_capacity_map(ResidualCapacityEdgeMap res)`

Отображение свойства остаточной реберной мощности. Тип должен быть моделью неконстантной LvaluePropertyMap. Тип ключа отображения должен быть дескриптором ребра графа.

По умолчанию: `get(edge_residual_capacity, g)`.

- `IN: reverse_edge_map(ReverseEdgeMap rev)`

Отображение свойства ребра, которое отображает каждое ребро (u, v) в графе на обратное ребро (v, u). Отображение должно быть моделью константной LvaluePropertyMap. Тип ключа отображения должен быть дескриптором ребра графа.

По умолчанию: `get(edge_reverse, g)`.

- `IN: vertex_index_map(VertexIndexMap index_map)`

Это отображает каждую вершину на целое в диапазоне $[0, N)$, где N — количество вершин в графе. Отображение должно быть моделью константной LvaluePropertyMap. Тип ключа отображения должен быть дескриптором ребра графа.

По умолчанию: `get(vertex_index, g)`.

Пример

Пример в листинге 13.12 читает пример задачи максимального потока (граф с мощностями ребер) из файла в формате DIMACS [1].

Листинг 13.12. Решение задачи максимального потока методом проталкивания предпотока

```
⟨ push-relabel-eg.cpp ⟩ ≡
#include <boost/config.hpp>
#include <iostream>
#include <string>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/read_dimacs.hpp>
```

Листинг 13.12 (*продолжение*)

```
  using namespace boost;
  typedef adjacency_list_traits < vecS, vecS, directedS > Traits;
  typedef adjacency_list < vecS, vecS, directedS,
    property < vertex_name_t, std::string >,
    property < edge_capacity_t, long,
    property < edge_residual_capacity_t, long,
    property < edge_reverse_t, Traits::edge_descriptor > > > > Graph;
  Graph g;

  property_map < Graph, edge_capacity_t >::type
    capacity = get(edge_capacity, g);
  property_map < Graph, edge_residual_capacity_t >::type
    residual_capacity = get(edge_residual_capacity, g);
  property_map < Graph, edge_reverse_t >::type rev = get(edge_reverse, g);
  Traits::vertex_descriptor s, t;
  read_dimacs_max_flow(g, capacity, rev, s, t);

  long flow = push_relabel_max_flow(g, s, t);

  std::cout << "c  The total flow:" << std::endl;
  std::cout << "s " << flow << std::endl << std::endl;
  std::cout << "c flow values:" << std::endl;
  graph_traits < Graph >::vertex_iterator u_iter, u_end;
  graph_traits < Graph >::out_edge_iterator ei, e_end;
  for (tie(u_iter, u_end) = vertices(g); u_iter != u_end; ++u_iter)
    for (tie(ei, e_end) = out_edges(*u_iter, g); ei != e_end; ++ei)
      if (capacity[*ei] > 0)
        std::cout << "f " << *u_iter << " " << target(*ei, g) << " "
          << (capacity[*ei] - residual_capacity[*ei]) << std::endl;
  return EXIT_SUCCESS;
}
```

Программа выводит следующее:

```
c  Полный поток:
s 13

c значения потока:
f 0 6 3
f 0 1 0
f 0 2 10
f 1 5 1
f 1 0 0
f 1 3 0
f 2 4 4
f 2 3 6
f 2 0 0
f 3 7 5
f 3 2 0
f 3 1 1
f 4 5 4
f 4 6 0
f 5 4 0
f 5 7 5
f 6 7 3
f 6 4 0
f 7 6 0
f 7 5 0
```

Классы BGL 14

14.1. Классы графов

14.1.1. adjacency_list

```
adjacency_list<EdgeList, VertexList, Directed,
  VertexProperties, EdgeProperties, GraphProperties>
```

Класс `adjacency_list` реализует интерфейс BGL-графа, используя несколько различных вариантов традиционного представления графа в виде списка смежности.

Представление графа в виде списка смежности содержит последовательность исходящих ребер для каждой вершины. Для разреженных графов это экономит место по сравнению с матрицей смежности, поскольку памяти требуется только порядка $O(|V| + |E|)$, а не $O(|V|^2)$. Кроме того, доступ к исходящим ребрам для каждой вершины может быть сделан эффективным. Представление ориентированного графа в виде списка смежности показано на рис. 14.1.

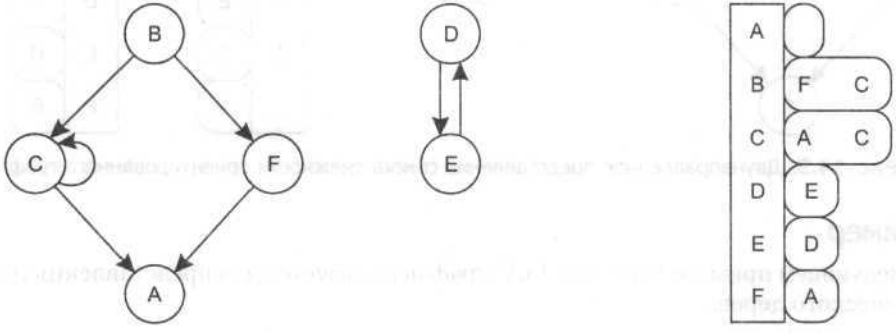


Рис. 14.1. Представление ориентированного графа в виде списка смежности

Параметры шаблона класса `adjacency_list` предоставляют много возможностей, это позволяет вам выбрать наиболее подходящий вариант класса. Параметр

шаблона VertexList класса adjacency_list задает вид контейнера для представления последовательности вершин (прямоугольник на рис. 14.1). Параметр шаблона EdgeList задает вид контейнера для представления последовательности исходящих ребер для каждой вершины (овалы на рис. 14.1). Выбор EdgeList и VertexList определяет затраты памяти и эффективность различных операций на графе. Возможные варианты и компромиссы рассмотрены далее в этом разделе.

Параметр шаблона Directed указывает, является ли граф ориентированным, неориентированным или ориентированным с доступом к входящим и исходящим ребрам (который мы называем *двунаправленным*). Двунаправленный граф требует в 2 раза больше места (на каждое ребро), чем ориентированный, так как каждое ребро появляется как в списке исходящих, так и в списке входящих ребер. Представление в виде списка смежности неориентированного графа показано на рис. 14.2. Двунаправленное представление ориентированного графа показано на рис. 14.3.

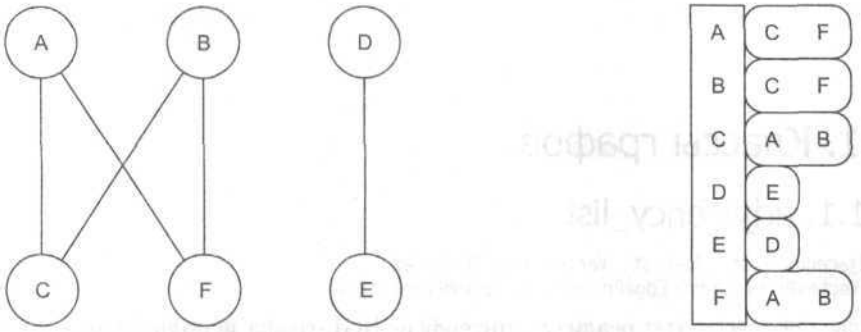


Рис. 14.2. Представление неориентированного графа в виде списка смежности

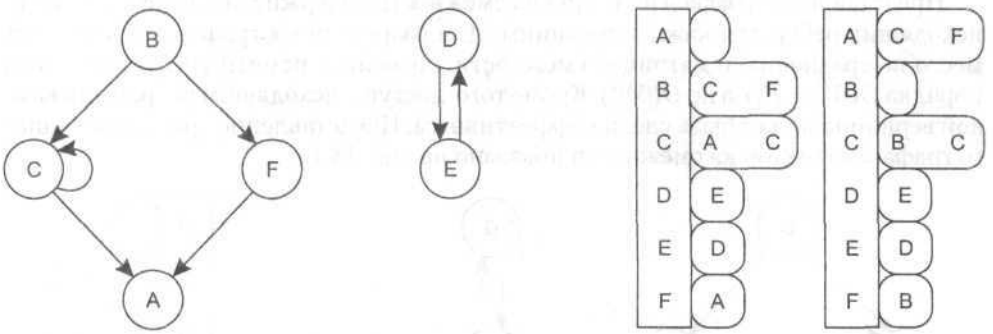


Рис. 14.3. Двунаправленное представление списка смежности ориентированного графа

Пример

В следующем примере (листинг 14.1) граф используется для представления генеалогического дерева.

Листинг 14.1. Работа с фамильным деревом

```
⟨ family-tree-eg.cpp ⟩ ≡
#include <iostream>
#include <vector>
```

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include <boost/tuple/tuple.hpp>

enum family { Jeanie, Debbie, Rick, John, Amanda, Margaret, Benjamin, N };

int main() {
  using namespace boost;
  const char *name[] = { "Jeanie", "Debbie", "Rick",
       "John", "Amanda", "Margaret", "Benjamin"
  };

  adjacency_list <> g(N);
  add_edge(Jeanie, Debbie, g);
  add_edge(Jeanie, Rick, g);
  add_edge(Jeanie, John, g);
  add_edge(Debbie, Amanda, g);
  add_edge(Rick, Margaret, g);
  add_edge(John, Benjamin, g);

  graph_traits < adjacency_list <> >::vertex_iterator i, end;
  graph_traits < adjacency_list <> >::adjacency_iterator ai, a_end;
  property_map < adjacency_list <>, vertex_index_t >::type
    index_map = get(vertex_index, g);

  for (tie(i, end) = vertices(g); i != end; ++i) {
    std::cout << name[get(index_map, *i)];
    tie(ai, a_end) = adjacent_vertices(*i, g);
    if (ai == a_end)
      std::cout << " не имеет детей";
    else
      std::cout << " является родителем для ";
    for (; ai != a_end; ++ai) {
      std::cout << name[get(index_map, *ai)];
      if (boost::next(ai) != a_end)
        std::cout << ", ";
    }
    std::cout << std::endl;
  }
  return EXIT_SUCCESS;
}
```

Эта программа выводит следующее:

```
Jeanie является родителем для Debbie, Rick, John
Debbie является родителем для Amanda
Rick является родителем для Margaret
John является родителем для Benjamin
Amanda не имеет детей
Margaret не имеет детей
Benjamin не имеет детей
```

Параметры шаблона

Ниже приведены параметры шаблона класса `adjacency_list`.

`EdgeList`	Контейнер для представления списка ребер каждой вершины. По умолчанию: `vecS`.
`VertexList`	Контейнер для представления множества вершин графа. По умолчанию: `vecS`.
`Directed`	Определяет, является ли граф ориентированным, неориентированным или ориентированным с двунаправленным доступом к ребрам (доступ к исходящим и входящим ребрам). Соответствующие значения: `directedS`, `undirectedS` и `bidirectionalS`. По умолчанию: `directedS`.
`VertexProperties`	Задает внутреннее хранилище для свойств вершин. По умолчанию: `no_property`.
`EdgeProperties`	Задает внутреннее хранилище для свойств ребер. По умолчанию: `no_property`.
`GraphProperties`	Задает хранилище для свойств графа. По умолчанию: `no_property`.

Модель для

DefaultConstructible, **CopyConstructible**, **Assignable**, **VertexListGraph**, **EdgeListGraph**, **IncidenceGraph**, **AdjacencyGraph**, **VertexMutableGraph** и **EdgeMutableGraph**.

Также `adjacency_list` моделирует концепцию **BidirectionalGraph**, когда `Directed=bidirectionalS` или `Directed=undirectedS`, и моделирует **VertexMutablePropertyGraph** и **EdgeMutablePropertyGraph**, когда добавляются соответствующие внутренние свойства.

Где определен

Класс `adjacency_list` находится в файле **boost/graph/adjacency_list.hpp**.

Ассоциированные типы

Ниже приведены ассоциированные типы класса `adjacency_list`.

- `graph_traits<adjacency_list>::vertex_descriptor`

Тип дескрипторов вершин, ассоциированный с `adjacency_list`.
(Требуется для **Graph**.)

- `graph_traits<adjacency_list>::edge_descriptor`

Тип дескрипторов ребер, ассоциированный с `adjacency_list`.
(Требуется для **Graph**.)

- `graph_traits<adjacency_list>::vertex_iterator`

Тип итераторов, возвращаемый `vertices()`.
(Требуется для **VertexListGraph**.)

- `graph_traits<adjacency_list>::edge_iterator`

Тип итераторов, возвращаемый `edges()`.
(Требуется для EdgeListGraph.)

- `graph_traits<adjacency_list>::out_edge_iterator`

Тип итераторов, возвращаемый `out_edges()`.
(Требуется для IncidenceGraph.)

- `graph_traits<adjacency_list>::in_edge_iterator`

Этот тип доступен для неориентированных или двунаправленных списков смежности, но не для ориентированных. Итератор `in_edge_iterator` — тип итератора, возвращаемый функцией `in_edges()`.
(Требуется для BidirectionalGraph.)

- `graph_traits<adjacency_list>::adjacency_iterator`

Тип итераторов, возвращаемый `adjacent_vertices()`.
(Требуется для AdjacencyGraph.)

- `graph_traits<adjacency_list>::directed_category`

Предоставляет информацию о том, является ли граф ориентированным (`directed_tag`) или неориентированным (`undirected_tag`).
(Требуется для Graph.)

- `graph_traits<adjacency_list>::edge_parallel_category`

Дает информацию о том, позволяет ли граф вставку параллельных ребер (ребер, у которых одна и та же начальная и конечная вершины). Два возможных тега: `allow_parallel_edge` и `disallow_parallel_edge_tag`. Варианты с `setS` и `hash_setS` не позволяют задавать параллельные ребра, тогда как другие варианты позволяют.
(Требуется для Graph.)

- `graph_traits<adjacency_list>::traversal_category`

Категория обхода (traversal category) отражает поддерживаемые графовым классом виды итераторов. Для списка смежности это включает итераторы вершин, ребер, исходящих ребер и итераторы смежности. Итератор входящих ребер доступен для неориентированных и двунаправленных, но не для ориентированных списков смежности.

- `graph_traits<adjacency_list>::vertices_size_type`

Тип используется для представления числа вершин в графе.
(Требуется для VertexListGraph.)

- `graph_traits<adjacency_list>::edges_size_type`

Тип используется для представления числа ребер в графе.
(Требуется для EdgeListGraph.)

- `graph_traits<adjacency_list>::degree_size_type`

Тип используется для представления числа исходящих ребер в графе.
(Требуется для IncidenceGraph.)

- `property_map<adjacency_list. PropertyTag>::type`
  `property_map<adjacency_list. PropertyTag>::const_type`

Тип *отображения* для свойств вершины или ребра графа. Свойство задается аргументом `PropertyTag` шаблона и должно совпадать с одним из свойств из `VertexProperties` или `EdgeProperties` для графа.
(Требуется для PropertyGraph.)

Функции — методы

Ниже приведены функции — методы класса `adjacency_list`.

- `adjacency_list(const GraphProperties& p = GraphProperties())`

  Конструктор по умолчанию. Создает пустой объект-граф с нулевым числом вершин и ребер.

  (Требуется для DefaultConstructible.)

- `adjacency_list(vertices_size_type n, const GraphProperties& p = GraphProperties())`

  Создает объект-граф с *n* вершинами и без ребер.

- ```
  template <typename EdgeIterator>
  adjacency_list(EdgeIterator first, EdgeIterator last,
    vertices_size_type n, edges_size_type m = 0,
    const GraphProperties& p = GraphProperties())
  ```

  Создает граф с *n* вершин и *m* ребер. Ребра заданы в списке ребер из диапазона [*first*, *last*). Если *n* или *m* равно нулю, число вершин или ребер вычисляется по списку ребер. Тип значения для `EdgeIterator` должен быть `std::pair`, где в паре используется целый тип. Целые числа соответствуют вершинам, и все они должны относиться к диапазону [0, *n*).

- ```
  template <typename EdgeIterator,
            typename EdgePropertiesIterator>
  adjacency_list(EdgeIterator first,
    EdgeIterator last, EdgePropertiesIterator ep_iter,
    vertices_size_type n, edges_size_type m = 0,
    const GraphProperties& p = GraphProperties())
  ```

  Создает графовый объект с *n* вершин и *m* ребер. Ребра заданы в списке ребер из диапазона [*first*, *last*). Если *n* или *m* равно нулю, число вершин или ребер вычисляется по списку ребер. Тип значения для `EdgeIterator` должен быть `std::pair`, где в паре используется целый тип. Целые числа соответствуют вершинам, все они должны относиться к диапазону [0, *n*). Тип `value_type` итератора `ep_iter` должен совпадать с параметром шаблона `EdgeProperties`.

Функции — не методы

Ниже приведены функции — не методы класса `adjacency_list`.

- ```
  std::pair<vertex_iterator, vertex_iterator>
  vertices(const adjacency_list& g)
  ```

  Возвращает пару итераторов, обеспечивающую доступ к множеству вершин графа *g*.

  (Требуется для VertexListGraph.)

- ```
  std::pair<edge_iterator, edge_iterator>
  edges(const adjacency_list& g)
  ```

  Возвращает пару итераторов, обеспечивающую доступ к набору ребер графа *g*.

  (Требуется для EdgeListGraph.)

- `std::pair<adjacency_iterator, adjacency_iterator>`
  `adjacent_vertices(vertex_descriptor v,`
  `const adjacency_list& g)`

Возвращает пару итераторов, обеспечивающую доступ к вершинам, смежным с вершиной v графа g.

(Требуется для AdjacencyGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `out_edges(vertex_descriptor v, const adjacency_list& g)`

Возвращает пару итераторов, обеспечивающую доступ к исходящим ребрам вершины v графа g. Если граф является неориентированным, этот итератор обеспечивает доступ ко всем ребрам, инцидентным вершине v.

(Требуется для IncidenceGraph.)

- `std::pair<in_edge_iterator, in_edge_iterator>`
  `in_edges(vertex_descriptor v, const adjacency_list& g)`

Возвращает пару итераторов, обеспечивающую доступ к входящим ребрам вершины v графа g. Эта операция недоступна, если для параметра шаблона `Directed` указано `directedS`, и доступна, если используются `undirectedS` и `bidirectionalS`.

(Требуется для BidirectionalGraph.)

- `vertex_descriptor source(edge_descriptor e,`
  `const adjacency_list& g)`

Возвращает начальную вершину ребра e.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,`
  `const adjacency_list& g)`

Возвращает конечную вершину ребра e.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,`
  `const adjacency_list& g)`

Возвращает число ребер, исходящих из вершины u.

(Требуется для IncidenceGraph.)

- `degree_size_type in_degree(vertex_descriptor u,`
  `const adjacency_list& g)`

Возвращает число ребер, входящих в вершину u. Операция доступна, только если параметр шаблона `Directed` был задан как `bidirectionalS`.

(Требуется для BidirectionalGraph.)

- `vertices_size_type num_vertices(const adjacency_list& g)`

Возвращает число вершин в графе g.

(Требуется для VertexListGraph.)

- `edges_size_type num_edges(const adjacency_list& g)`

Возвращает число ребер в графе g.

(Требуется для EdgeListGraph.)

- `vertex_descriptor vertex(vertices_size_type n,`
  `const adjacency_list& g)`

Возвращает *n*-ю вершину в списке вершин графа.

- `std::pair<edge_descriptor, bool>`
  `edge(vertex_descriptor u, vertex_descriptor v,`
  `const adjacency_list& g)`

Возвращает ребро, соединяющее вершину *u* с вершиной *v* в графе *g*.

(Требуется для AdjacencyMatrix.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `edge_range(vertex_descriptor u, vertex_descriptor v,`
  `const adjacency_list& g)`

Возвращает пару итераторов исходящих ребер, дающих доступ ко всем параллельным ребрам из `u` в `v`. Эта функция работает, только когда `EdgeList` для `adjacency_list` является контейнером, сортирующим исходящие ребра по конечным вершинам, а также когда параллельные ребра разрешены. `multisetS` в качестве `EdgeList` является таким контейнером.

- `std::pair<edge_descriptor, bool>`
  `add_edge(vertex_descriptor u, vertex_descriptor v,`
  `adjacency_list& g)`

Добавляет ребро (*u*, *v*) к графу и возвращает дескриптор ребра для нового ребра. Для тех графов, которые не разрешают иметь параллельные ребра, в случае если ребро уже присутствует в графе, дубликат добавлен не будет, а флаг `bool` будет иметь значение «ложь». Также, если *u* и *v* являются дескрипторами одной и той же вершины, а граф является неориентированным, создающее петлю ребро добавлено не будет и флаг `bool` будет иметь значение «ложь». Когда флаг ложен, дескриптор ребра не является правильным и его нельзя использовать. Место нового ребра в списке исходящих ребер в общем случае не определено, хотя задать порядок в списке исходящих ребер можно при выборе `EdgeList`. Если `VertexList=vecS` и если один из дескрипторов вершин `u` или `v` (целые числа) имеет значение, большее, чем текущее число вершин графа, граф увеличивается таким образом, что число вершин становится равным `std::max(u,v) + 1`. Если `EdgeList=vecS`, добавление ребра делает недействительным любой итератор исходящих ребер (`out_edge_iterator`) для вершины *u*. То же самое происходит, если `EdgeList` — определенный пользователем контейнер, итераторы которого «портятся» при вызове `push(container, x)`. Если граф является двунаправленным, то итераторы входящих ребер для *v* (`in_edge_iterator`) тоже «портятся». Если граф является неориентированным, то любой итератор `out_edge_iterator` для *v* также «портится». Если граф является ориентированным, `add_edge()` «портит» любой итератор ребер (`out_edge`).

(Требуется для EdgeMutableGraph.)

- `std::pair<edge_descriptor, bool>`
  `add_edge(vertex_descriptor u, vertex_descriptor v,`
  `const EdgeProperties& p, adjacency_list& g)`

Добавляет ребро (*u*, *v*) к графу и присоединяет `p` как значение внутреннего свойства ребра. См. также описание предыдущей функции — не метода класса.

(Требуется для EdgeMutablePropertyGraph.)

- `void remove_edge(vertex_descriptor u, vertex_descriptor v, adjacency_list& g)`

Удаляет ребро (u, v) из графа. Эта операция вызывает сбой во всех еще не обработанных дескрипторах ребер и итераторах, которые указывают на ребро (u, v). Кроме того, если в качестве `EdgeList` выбран `vecS`, тогда эта операция вызывает сбой во всех итераторах, указывающих на элемент списка ребер вершины u. То же самое происходит для вершины v в случае неориентированного или двунаправленного графа. Аналогично, для ориентированных графов эта операция вызывает сбой любого итератора ребер (`edge_iterator`).

(Требуется для **EdgeMutableGraph**.)

- `void remove_edge(edge_descriptor e, adjacency_list& g)`

Удаляет ребро e из графа. Отличается от функции `remove_edge(u, v, g)` в случае мультиграфа. Эта функция удаляет единственное ребро графа, тогда как функция `edge(u, v, g)` удаляет все ребра (u, v). Данная операция делает недействительными любые еще не обработанные дескрипторы ребер и итераторы для ребра e. Кроме того, эта операция «портит» все итераторы, которые указывают на список ребер для `target(e, g)`. Аналогично, для ориентированных графов эта операция вызывает сбой любого итератора ребер.

(Требуется для **EdgeMutableGraph**.)

- `void remove_edge(out_edge_iterator iter, adjacency_list& g)`

Имеет тот же эффект, что и `remove_edge(*iter, g)`. Разница состоит в том, что эта функция выполняется за постоянное время в случае ориентированных графов, тогда как `remove_edge(e, g)` имеет временную сложность порядка $O(|E|/|V|)$.

(Требуется для **MutableIncidenceGraph**.)

- ```
  template <typename Predicate>
  void remove_out_edge_if (vertex_descriptor u,
    Predicate predicate, adjacency_list& g)
  ```

Удаляет все исходящие ребра вершины u из графа, которые удовлетворяют предикату, то есть если предикат возвращает истину при применении к дескриптору ребра, ребро удаляется. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для каждого из подлежащих удалению ребер.

(Требуется для **MutableIncidenceGraph**.)

- ```
  template <typename Predicate>
  void remove_in_edge_if (vertex_descriptor v,
    Predicate predicate, adjacency_list& g)
  ```

Удаляет все входящие ребра вершины v из графа, которые удовлетворяют предикату, то есть, если предикат возвращает истину при применении к дескриптору ребра, ребро удаляется. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для каждого из подлежащих удалению ребер.

(Требуется для **MutableBidirectionalGraph**.)

- ```
  template <typename Predicate>
  void remove_edge_if (Predicate predicate,
    adjacency_list& g)
  ```
  ```

Удаляет все ребра из графа, которые удовлетворяют предикату, то есть если предикат возвращает истину при применении к дескриптору ребра, ребро удаляется. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для каждого из подлежащих удалению ребер.

(Требуется для MutableEdgeListGraph.)

- `vertex_descriptor add_vertex(adjacency_list& g)`

Добавляет вершину к графу и возвращает дескриптор вершины для вновь созданной вершины.

(Требуется для VertexMutableGraph.)

- `vertex_discriptor add_vertex(const VertexProperties& p, adjacency_list& g)`

Добавляет вершину к графу и возвращает дескриптор вершины для вновь созданной вершины.

(Требуется для VertexMutablePropertyGraph.)

- `void clear_vertex(vertex_descriptor u, adjacency_list& g)`

Удаляет все ребра, входящие и исходящие из вершины *u*. Вершина остается во множестве вершин графа. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для всех ребер, у которых `u` — начальная или конечная вершина.

(Требуется для EdgeMutableGraph.)

- `void clear_out_edges(vertex_descriptor u, adjacency_list& g)`

Удаляет все ребра, исходящие из вершины *u*. Вершина остается во множестве вершин графа. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для всех ребер, у которых `u` — начальная вершина. Эта операция не применяется к неориентированным графам (вместо нее используйте `clear_vertex()`).

- `void clear_in_edges(vertex_descriptor u, adjacency_list& g)`

Удаляет все ребра, входящие в вершину *u*. Вершина остается во множестве вершин графа. Эффект для дескриптора и итератора такой же, что и при вызове `remove_edge()` для всех ребер, у которых `u` — конечная вершина. Эта операция применима только к двунаправленным графам.

- `void remove_vertex(vertex_descriptor u, adjacency_list& g)`

Удаляет вершину *u* из множества вершин графа. Предполагается, что на момент удаления у этой вершины нет входящих или исходящих ребер. Чтобы гарантировать такое состояние, можно заранее применить `clear_vertex()`. Если параметр шаблона `VertexList` списка `adjacency_list` — `vecS`, то все дескрипторы вершин, дескрипторы ребер и итераторы для графа становятся недействительными. Встроенные свойства `vertex_index_t` для каждой вершины после этой операции будут перенумерованы таким образом, что индексы вершин по-прежнему образуют непрерывный диапазон $[0, |V|)$. Если вы используете внешнее хранилище свойств, основанное на встроенных индексах вершин, то внешнее хранилище должно быть соответствующим образом перестроено. Другой воз-

можностью является отказ от использования встроенного индекса вершин в пользу использования свойства для добавления собственного индекса вершин. Если вам необходимо часто использовать функцию remove_vertex(), listS является гораздо лучшим выбором для шаблонного параметра VertexList.

(Требуется для VertexMutableGraph.)

- ```
  template <typename PropertyTag>
  property_map<adjacency_list, PropertyTag>::type
  get(PropertyTag, adjacency_list& g)
  ```

Возвращает изменяемый объект-отображение свойств для свойства вершины, заданной PropertyTag. PropertyTag должен совпадать с одним из свойств, заданных в шаблонном параметре VertexProperties графа.

(Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag>
  property_map<adjacency_list, PropertyTag>::const_type
  get(PropertyTag, const adjacency_list& g)
  ```

Возвращает константный объект-отображение свойств для свойства вершины, заданной PropertyTag. PropertyTag должен совпадать с одним из свойств, заданных в шаблонном параметре VertexProperties графа.

(Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X>
  typename property_traits<
    typename property_map<adjacency_list,
    PropertyTag>::const_type>::value_type
  get(PropertyTag, const adjacency_list& g, X x)
  ```

Возвращает значение свойства для x, где x — дескриптор вершины или ребра.

(Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X,
    typename Value>
  void put(PropertyTag, const adjacency_list& g, X x,
    const Value& value)
  ```

Присваивает значение value свойству x, где x — дескриптор вершины или ребра. Значение должно быть преобразуемым к typename property_traits<property_map<adjacency_list, PropertyTag>::type>::value_type.

(Требуется для PropertyGraph.)

- ```
  template <typename GraphProperties,
    typename GraphProperties>
  typename property_value<GraphProperties,
    GraphProperties>::type&
  get_property(adjacency_list& g, GraphProperties);
  ```

Возвращает свойство, заданное по GraphProperties и присоединенное к графовому объекту g. Класс свойств property_value определен в boost/pending/property.hpp.

- ```
  template <typename GraphProperties,
    typename GraphProperties>
  const typename property_value<GraphProperties,
    GraphProperties>::type&
  get_property(const adjacency_list& g, GraphProperties);
  ```

Возвращает свойство, заданное по GraphProperties, присоединенное к графовому объекту g. Класс свойств property_value определен в boost/pending/property.hpp.

Выбираем EdgeList и VertexList

В этом разделе уделено внимание тому, какую версию класса adjacency_list использовать в той или иной ситуации. Список смежности имеет очень много возможностей для конфигурирования. Интересующие нас параметры EdgeList и VertexList задают структуры данных, используемые для представления графа. Выбор EdgeList и VertexList влияет на временную сложность многих графовых операций и пространственную сложность графового объекта.

BGL использует контейнеры из STL, такие как вектор std::vector, список std::list и множество std::set для представления множества вершин и структуры смежности (входящие и исходящие ребра) графа. В качестве контейнера для EdgeList и VertexList могут быть выбраны разные типы:

- vecS задает std::vector.
- listS задает std::list.
- slistS задает std::slist[1].
- setS задает std::set.
- hash_setS задает std::hash_set[2].

Выбираем тип VertexList

Параметр VertexList определяет вид контейнера, который будет использован для представления множества вершин или двумерной структуры графа. Контейнер должен быть моделью Sequence или RandomAccessContainer. В общем случае listS является хорошим выбором, если вам нужно быстро добавить и удалить вершины. Но тогда появляются дополнительные накладные расходы по сравнению с vecS.

Пространственная сложность: std::list требует хранить больше информации для каждой вершины, чем std::vector, так как дополнительно хранит два указателя.

Временная сложность: выбор VertexList влияет на временную сложность следующих операций.

- add_vertex()

Эта операция выполняется за амортизированное постоянное время как для vecS, так и для listS (реализована с push_back()). Однако, когда тип VertexList=vecS является типом vecS, время выполнения этой операции иногда больше из-за того, что вектор приходится размещать в памяти заново, а весь граф копировать.

- remove_vertex()

Эта операция выполняется за постоянное время для listS и за время $O(|V|+|E|)$ для vecS. Большая временная сложность vecS объясняется тем, что дескрипторы вершин (которые в этом случае являются индексами, соответствующими

[1] Если реализация STL, которую вы применяете, использует std::slist.

[2] Если реализация STL, которую вы применяете, имеет std::hash_set. Например, SGI STL является такой реализацией.

месту в списке вершин) должны быть скорректированы в исходящих ребрах всего графа.

- `vertex()`

Эта операция выполняется за постоянное время для `vecS` и $O(|V|)$ для `listS`.

Выбираем тип EdgeList

Параметр `EdgeList` определяет, какой вид контейнера используется для хранения исходящих ребер (и, возможно, входящих тоже) для каждой вершины в графе. Контейнеры, используемые для списков ребер, должны удовлетворять требованиям либо Sequence, либо AssociativeContainer.

Одним из первых вопросов, которые необходимо рассмотреть при выборе `EdgeList`, является то, хотите ли вы гарантировать отсутствие параллельных ребер в графе. Если нужна гарантия того, что граф не станет мультиграфом, можно использовать `setS` или `hash_setS`. Если вы хотите иметь мультиграф или знаете, что параллельные ребра вставляться не будут, тогда можно выбрать один из типов-последовательностей: `vecS`, `listS` или `slistS`. Помимо этого, необходимо принять во внимание разницу во временной и пространственной сложности для различных графовых операций. Мы используем $|V|$ для обозначения общего числа вершин графа и $|E|$ для числа ребер. Операции, которые здесь не рассмотрены, выполняются за постоянное время.

Пространственная сложность: выбор `EdgeList` влияет на объем памяти, выделяемой на одно ребро в графовом объекте. В порядке возрастания требуемого места следуют: `vecS`, `slistS`, `listS`, `hash_setS` и `setS`.

Временная сложность: в следующем описании временной сложности различных операций мы используем $|E|/|V|$ внутри обозначений с «большим O» для выражения длины списка исходящих ребер. Строго говоря, это не совсем точно, так как $|E|/|V|$ дает только среднее число ребер на вершину в графе. В худшем случае число исходящих ребер для вершины равно $|V|$ (если граф — не мультиграф). В разреженных графах $|E|$ обычно намного меньше, чем $|V|$, и может рассматриваться как константа.

- `add_edge()`

Когда `EdgeList` является UniqueAssociativeContainer (ассоциативный контейнер с уникальными элементами) как `std::set`, отсутствие параллельных ребер после добавления ребра гарантировано. Дополнительное время поиска имеет временную сложность $O(\log(|E|/|V|))$. Типы `EdgeList`, которые моделируют Sequence (последовательность), не осуществляют такую проверку и поэтому `add_edge()` выполняется за амортизированное постоянное время. Это означает, что если вам безразлично, имеет ли граф параллельные ребра, или вы уверены, что параллельные ребра не будут добавляться к графу, то лучше использовать основанный на последовательностях `EdgeList`. Функция `add_edge()` для последовательного `EdgeList` реализована как `push_front()` или `push_back()`. Однако для `std::list` и `std::slist` эта операция обычно выполняется быстрее, чем для `std::vector`, который иногда перемещается в памяти и копирует свои элементы.

- `remove_edge()`

Для последовательных типов `EdgeList` эта операция реализуется с использованием `std::remove_if()`. Это означает, что среднее время равно $|E|/|V|$. Для

основанных на множествах типов EdgeList используется функция — метод класса erase(), которая имеет временную сложность $\log(|E| / |V|)$.

- edge()

Временная сложность этой операции равна $O(|E| / |V|)$, когда тип EdgeList является Sequence, и $O(\log(|E| / |V|))$, когда тип EdgeList является AssociativeContainer.

- clear_vertex()

Для ориентированных графов с последовательным типом EdgeList временная сложность порядка $O(|E| + |V|)$, тогда как для EdgeList на основе ассоциативного контейнера операция выполняется быстрее, всего за $O(|V|\log(|E|/|V|))$. Для неориентированных графов данная операция имеет временную сложность порядка $O((|E| / |V|)^2)$ и $O(|E|\log(|E| / |V|) / |V|)$.

- remove_vertex()

Временная сложность порядка $O(|E| + |V|)$ вне зависимости от типа EdgeList.

- out_edge_iterator::operator++()

Эта операция выполняется за постоянное время для всех типов EdgeList. Однако имеется значительная разница (постоянный множитель) по времени между различными типами. И это важно, поскольку операция является «рабочей лошадкой» многих алгоритмов на графах. Скорость этой операции в порядке ее уменьшения: vecS, slistS, listS, setS, hash_setS.

- in_edge_iterator::operator++()

См. выше.

- vertex_iterator::operator++()

Эта операция выполняется за постоянное время и достаточно быстро (выполняется со скоростью инкремента указателя). Выбор OneD не влияет на скорость этой операции.

- edge_iterator::operator++()

Данная операция выполняется за постоянное время и показывает похожее упорядочение по скоростям, что и out_edge_iterator в отношении выбора EdgeList. Обход всех ребер имеет временную сложность порядка $O(|E| + |V|)$.

- adjacency_iterator::operator++()

Данная операция выполняется за постоянное время и показывает похожее упорядочение по скоростям, что и out_edge_iterator в отношении выбора EdgeList.

Стабильность и сбои итераторов и дескрипторов

При изменении структуры графа (путем добавления или удаления ребер) необходимо действовать осторожно. В зависимости от типа adjacency_list и от операции некоторые объекты-итераторы и объекты-дескрипторы, указывающие на граф, могут стать некорректными. Например, результаты выполнения кода в листинге 14.2 неопределенны и могут оказаться разрушительными.

Листинг 14.2. Пример сбоя итераторов

```
// VertexList=vecS
typedef adjacency_list<listS, vecS> Graph;
```

```
Graph G(N);

// Наполнить граф...

// Попытка удалить все вершины. Неверно!
graph_traits<Graph>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(G); vi != vi_end; ++vi)
  remove_vertex(*vi, G);

// Другая попытка удалить все вершины. Все равно неверно!
graph_traits<Graph>::vertex_iterator vi, vi_end, next;
tie(vi, vi_end) = vertices(G);
for (next = vi; vi != vi_end; vi = next) {
  ++next;
  remove_vertex(*vi, G);
}
```

Причина этой проблемы в том, что мы вызываем `remove_vertex()`, который при использовании `adjacency_list` с `VertexList=vecS` вызывает сбой всех итераторов и дескрипторов графа (в нашем случае — `vi` и `vi_end`), таким образом, ошибка появляется в последующих итерациях цикла.

При использовании другого вида `adjacency_list`, где `VertexList=listS`, итераторы не «портятся» при удалении вершин, если, конечно, итератор не указывал на удаляемую вершину. Это демонстрирует код из листинга 14.3.

Листинг 14.3. Пример сбоя итераторов (2)

```
// VertexList=listS
typedef adjacency_list<listS, listS> Graph;
Graph G(N);
// Наполнить граф...

// Попытка удалить все вершины. Неверно!
graph_traits<Graph>::vertex_iterator vi, vi_end;
for (tie(vi, vi_end) = vertices(G); vi != vi_end; ++vi)
  remove_vertex(*vi, G);

// Удалить все вершины. Правильно.
graph_traits<Graph>::vertex_iterator vi, vi_end, next;
tie(vi, vi_end) = vertices(G);
for (next = vi; vi != vi_end; vi = next) {
  ++next;
  remove_vertex(*vi, G);
}
```

Наиболее безопасным и эффективным способом массового удаления ребер из `adjacency_list` является применение функции `remove_edge_if()`.

Вопрос корректности касается также дескрипторов вершин и ребер. Например, предположим, что вы используете вектор дескрипторов вершин для отслеживания родителей (или предшественников) вершин в дереве кратчайших путей (см. файл **example/dijkstra-example.cpp**). Вы создаете вектор родителей вызовом `dijkstra_shortest_paths()`, а затем удаляете из графа вершину. После этого вы пытаетесь использовать вектор родителей, но поскольку все дескрипторы вершин стали некорректны, результат также неверен. Это можно проследить в листинге 14.4.

Листинг 14.4. Пример сбоя дескрипторов вершин

```
std::vector<Vertex> parent(num_vertices(G));
std::vector<Vertex> distance(num_vertices(G));
dijkstra_shortest_paths(G, s, distance_map(&distance[0]).
  predecessor_map(&parent[0]));

// Плохая идея! Дескрипторы вершин становятся некорректными
// в векторе родителей
remove_vertex(s, G);

// Получаем неверные результаты
for(tie(vi, vend) = vertices(G); vi != vend; ++vi)
  std::cout << p[*vi] << " является родителем для " << *vi << std::endl;
```

При поиске причины сбоев итераторов и дескрипторов следует обратить внимание, что затрагиваются дескрипторы и итераторы, *не участвующие* в операции непосредственно. Например, выполнение `remove_edge(u, v, g)` всегда делает недействительными дескриптор ребра (u, v) или итератор, указывающий на (u, v), независимо от вида `adjacency_list`. То есть в этом разделе нас волнуют эффекты, которые вызывает `remove_edge(u, v, g)` для дескрипторов и итераторов, указывающих на отличные от (u, v) ребра.

В общем случае, если вы хотите, чтобы дескрипторы вершин и ребер были стабильными (никогда не «портились»), следует использовать `listS` или `setS` для шаблонных параметров `VertexList` и `EdgeList` класса `adjacency_list`. Если для вас важнее затраты памяти и скорость обхода графа, используйте `vecS` для шаблонных параметров `VertexList` и/или `EdgeList`.

Ориентированные и неориентированные списки смежности

Класс `adjacency_list` может быть использован для представления как ориентированных, так и неориентированных графов, в зависимости от аргумента, присвоенного шаблонному параметру `Directed`. Указанием `directedS` или `bidirectionalS` выбирается ориентированный граф, тогда как `undirectedS` выбирает неориентированный. См. раздел 12.1.1, где дано описание различий между ориентированным и неориентированным графами в BGL. Выбор `bidirectionalS` указывает, что граф предоставит функцию `in_edges()` в дополнение к функции `out_edges()`. Это требует двойных расходов памяти на одно ребро (и служит причиной того, почему `in_edges()` необязательна).

Внутренние свойства

Свойства могут быть закреплены за вершинами и ребрами графа, заданного списком смежности, через *интерфейс свойств* (property interface). Шаблонные параметры `VertexProperties` и `EdgeProperties` класса `adjacency_list` подразумевают заполнение классом свойств, который определен следующим образом.

```
template <typename PropertyTag, typename T,
          typename NextProperty = no_property>
struct property;
```

`PropertyTag` — это тип, который просто идентифицирует или дает уникальное имя свойству. Имеются несколько предопределенных тегов (см. раздел 15.2.3), и очень легко добавить новые. Для удобства BGL также предоставляет предопределенные объекты теговых типов (в данном случае значения перечисления `enum`) для использования в качестве аргументов к функциям, которые принимают объекты-теги свойств (к ним относится, например, функция `get()` из `adjacency_list`).

Параметр T свойства property обозначает тип значений свойств. Параметр Next-Property позволяет указать следующее свойство, так что произвольное число свойств может быть связано с одним и тем же графом.

Следующий код показывает, как свойства вершин и ребер могут быть использованы при создании графа. Мы связали свойство «расстояние» со значениями типа float и свойство «имя» со значениями типа std::string с вершинами графа. С ребрами графа связано свойство «вес» со значениями типа float.

```
typedef property<distance_t, float,
  property<name_t, std::string> > VertexProperties;
typedef property<weight_t, float> EdgeProperties;
typedef adjacency_list<mapS, vecS, undirectedS,
  VertexProperties, EdgeProperties> Graph;
Graph g(num_vertices);   // построить графовый объект
```

Значения свойств могут быть прочитаны и записаны с использованием отображений свойств. Описание того, как извлечь отображения свойств из графа, см. в разделе 3.6. Глава 15 целиком посвящена использованию отображений свойств.

Свойство индекса вершины

Если VertexList графа есть vecS, то граф имеет встроенное свойство «индекс», которое может быть получено через свойство vertex_index_t. Индексы находятся в диапазоне $[0, |V|)$, без пропусков. Когда вершина удаляется, индексы перестраиваются так, что опять находятся в соответствующем диапазоне и без пропусков. Необходима некоторая осторожность при пользовании этими индексами для доступа к свойствам, хранящимся вне графового объекта, поскольку пользователь должен обновить внешнее хранилище в соответствии с новыми индексами.

Свойства ребер, созданные пользователем

Создание пользовательских типов свойств — достаточно простое дело. Нужно только определить теговый класс для нового свойства. В следующем коде определяются теговый класс для свойств «мощность» и «поток», которые мы закрепляем за ребрами графа.

```
enum edge_capacity_t { edge_capacity };
enum edge_flow_t { edge_flow };

namespace boost {
  BOOST_INSTALL_PROPERTY(edge, flow);
  BOOST_INSTALL_PROPERTY(edge, capacity);
}
```

Теперь вы можете использовать тег нового свойства в определении свойств так же, как и один из встроенных тегов.

```
typedef property<capacity_t, int> Cap;
typedef property<flow_t, int, Cap> EdgeProperties;
typedef adjacency_list<vecS, vecS, no_property, EdgeProperties> Graph;
```

Как обычно, отображения свойств для этих свойств могут быть получены из графа через функцию get().

```
property_map<Graph, edge_capacity_t>::type
  capacity = get(edge_capacity, G);
property_map<Graph, edge_flow_t>::type
  flow = get(edge_flow, G);
```

В файле edge_property.cpp приведен полный исходный код для рассмотренного примера.

Свойства вершин, созданные пользователем

Закреплять за вершинами некоторые свойства так же легко, как закреплять их за ребрами. Здесь мы хотим закрепить за вершинами графа имена людей.

```
enum vertex_first_name_t { vertex_first_name };
namespace boost {
  BOOST_INSTALL_PROPERTY(vertex, first_name);
}
```

Теперь мы можем использовать новый тег в классе свойств property при сборке графового типа. Следующий код (листинг 14.5) показывает создание графового типа и затем создание объекта-графа. Мы заполняем ребра и так же назначаем имена вершинам. Ребра представляют информацию о том, «кто кому должен».

Листинг 14.5. Создание графового типа и объекта

```
typedef adjacency_list<vecS, vecS, directedS,
  property<vertex_first_name_t, std::string> > MyGraphType;

typedef pair<int,int> Pair;
  Pair edge_array[11] = { Pair(0,1), Pair(0,2), Pair(0,3), Pair(0,4),
                          Pair(2,0), Pair(3,0), Pair(2,4), Pair(3,1),
                          Pair(3,4), Pair(4,0), Pair(4,1) };

MyGraphType G(5);
for (int i=0; i<11; ++i)
  add_edge(edge_array[i].first, edge_array[i].second, G);

property_map<MyGraphType, vertex_first_name_t>::type
  name = get(vertex_first_name, G);

boost::put(name, 0, "Jeremy");
boost::put(name, 1, "Rich");
boost::put(name, 2, "Andrew");
boost::put(name, 3, "Jeff");
name[4] = "Kinis";  // можно и так

who_owes_who(edges(G).first, edges(G).second, G);
```

Функция who_owes_who(), написанная для этого примера, была реализована в обобщенном стиле. Ввод задан классом-шаблоном, так что мы не знаем реального типа графа. Для нахождения типа отображения свойства для свойства first_name нам необходимо использовать класс свойств vertex_property_map. Тип const_type использован из-за того, что параметр-граф также является константой. Как только мы получили тип отображения свойства, мы можем сделать заключение о типе значений свойства, используя класс property_traits. В нашем примере известно, что тип значения свойства — std::string, но написанная в таком обобщенном стиле функция who_owes_who() может работать и с другими типами значений свойства. Код функции приведен в листинге 14.6.

Листинг 14.6. Функция who_owes_who()

```
template <class EdgeIter, class Graph>
void who_owes_who(EdgeIter first, EdgeIter last, const Graph& G)
```

```
{
  // Доступ к типу средства доступа к свойству для этого графа
  typedef typename property_map<Graph, vertex_first_name_t>
    ::const_type NamePA;
  NamePA name = get(vertex_first_name, G);

  typedef typename boost::property_traits<NamePA>::value_type NameType;
  NameType src_name, targ_name;

  while (first != last) {
    src_name = boost::get(name, source(*first,G));
    targ_name = boost::get(name, target(*first,G));
    cout << src_name << " должен "
         << targ_name << " деньги" << endl;
    ++first;
  }
}
```

Эта программа выводит следующее:

```
Jeremy должен Rich деньги
Jeremy должен Andrew деньги
Jeremy должен Jeff деньги
Jeremy должен Kinis деньги
Andrew должен Jeremy деньги
Andrew должен Kinis деньги
Jeff должен Jeremy деньги
Jeff должен Rich деньги
Jeff должен Kinis деньги
Kinis должен Jeremy деньги
Kinis должен Rich деньги
```

Полный исходный код этого примера можно найти в файле interior_property_map.cpp.

Настройка хранилища для списка смежности

Класс adjacency_list реализован с использованием двух видов контейнеров. Один из типов контейнеров содержит все вершины графа, а другой — список исходящих ребер (и, возможно, входящих) для каждой вершины. BGL предоставляет классы-селекторы для того, чтобы пользователь мог выбрать среди нескольких контейнеров из STL. Также есть возможность для использования своего собственного контейнерного типа. При настройке VertexList вам необходимо определить генератор контейнера. При настройке EdgeList нужно определить генератор контейнера и свойства параллельных ребер. Файл container_gen.cpp является примером того, как использовать настройку хранилища.

Генератор контейнера

Класс adjacency_list использует класс свойств, называемый container_gen, для отображения селекторов EdgeList и VertexList на реальные контейнерные типы, применяемые для хранения графа. Версия класса свойств по умолчанию приведена в листинге 14.7 вместе с примером того, как класс специализируется для селектора listS.

Листинг 14.7. Версия класса свойств по умолчанию

```
namespace boost {
  template <typename Selector, typename ValueType>
```

продолжение ⇩

Листинг 14.7 (*продолжение*)

```
truct container_gen { };
template <typename ValueType>
truct container_gen<listS, ValueType> {
  typedef std::list<ValueType> type;
};
}
```

Для использования другого контейнера на ваш выбор определите класс-селектор и затем специализируйте container_gen для вашего селектора (листинг 14.8).

Листинг 14.8. Версия класса свойств со специализацией для селектора

```
struct custom_containerS { };  // ваш селектор
namespace boost {
  //  специализация для вашего селектора
  template <typename ValueType>
  struct container_gen<custom containerS, ValueType> {
  typedef custom_container<ValueType> type;
  };
}
```

Могут возникнуть ситуации, когда вы хотите использовать контейнер, который имеет больше шаблонных параметров, чем просто ValueType. Например, вы можете захотеть представить тип распределителя памяти (allocator type). Один из способов сделать это — четко прописать в дополнительных параметрах в специализации container_gen. Однако если вы хотите большей гибкости, то можно добавить шаблонный параметр к классу-селектору. В следующем коде в листинге 14.9 показано, как создать селектор, который позволяет задать распределитель памяти в std::list.

Листинг 14.9. Версия класса свойств со специализацией для селектора и типа размещения

```
template <typename Allocator> struct list_with_allocatorS {};
namespace boost {
  template <typename Alloc, typename ValueType>
  struct container_gen<list_with_allocatorS<Alloc>, ValueType>
  {
    typedef typename Alloc::template rebind<ValueType>::other Allocator;
    typedef std::list<ValueType, Allocator> type;
  };
}
//  теперь вы можете определить граф, используя std::list
//  и специальное размещение
typedef adjacency_list< list_with_allocatorS< std::allocator<int> >,
  vecS, directedS> MyGraph;
```

Свойства параллельных ребер

В дополнение к специализации класса container_gen можно также специализировать класс parallel_edge_traits для задания того, позволяет ли контейнерный тип иметь параллельные ребра (является Sequence) или не позволяет (является AssociativeContainer).

```
template <typename StorageSelector>
struct parallel_edge_traits { };
template <> struct parallel_edge_traits<vecS> {
  typedef allow_parallel_edge_tag type;
};
```

```
template <> struct parallel_edge_traits<setS> {
  typedef disallow_parallel_edge_tag type;
};
//...
```

Контейнер списка ребер: функции push() и erase()

Необходимо указать adjacency_list как ребра могут быть эффективно добавлены и удалены из контейнера списка ребер. Это выполняется перегрузкой функций push() и erase() для собственного контейнерного типа. Функция push() должна возвращать итератор, указывающий на только что вставленное ребро, и логический флаг, говорящий о том, было ли ребро вставлено. Если было задано allow_parallel_edge_tag для parallel_edge_traits, то push() должна всегда вставлять ребро и возвращать истину. Если было задано disallow_parallel_edge_tag, функция push() должна возвратить ложь и не вставлять ребро, если такое ребро уже есть в контейнере, и итератор должен указывать на уже существующее ребро.

Следующие функции — push() и erase() (листинг 14.10) по умолчанию уже написаны для контейнеров из STL. Семейство перегруженных функций push_dispatch() и erase_dispatch() обеспечивает различные способы вставки и удаления, которые могут быть выполнены для стандартных контейнеров.

Листинг 14.10. Функции push() и erase()

```
template <typename Container, typename T>
std::pair<typename Container::iterator, bool>
push(Container& c, const T& v)
{
  return push_dispatch(c, v, container_category(c));
}

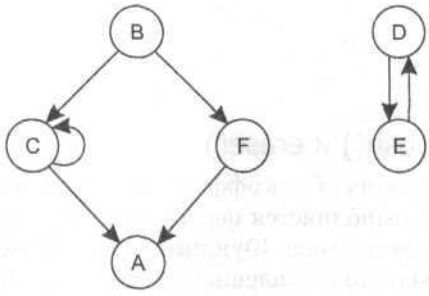
template <typename Container, typename T>
void erase(Container& c, const T& x)
{
  erase_dispatch(c, x, container_category(c));
}
```

14.1.2. adjacency_matrix

```
adjacency_matrix<Directed, VertexProperty, EdgeProperty, GraphProperty>
```

Класс adjacency_matrix реализует интерфейс BGL-графа, используя несколько различных вариантов традиционной графовой структуры матрицы смежности. Для графа с $|V|$ вершин используется матрица $|V| \times |V|$, где каждый элемент a_{ij} является логическим флагом, свидетельствующим о том, имеется ли ребро из вершины i в вершину j. Представление графа в виде матрицы смежности показано на рис. 14.4.

Преимуществом такого матричного формата над списком смежности является то, что ребра вставляются и убираются за постоянное время. Есть и несколько недостатков. Во-первых, объем используемой памяти имеет порядок $O(|V|^2)$ вместо $O(|V| + |E|)$ (где $|E|$ — число вершин графа). Во-вторых, операции по всем исходящим ребрам каждой вершины (как поиск в ширину) имеют временную сложность $O(|V|^2)$ в отличие от $O(|V| + |E|)$ для списка смежности. Матрицу смежности лучше использовать с плотными графами (где $|E| \approx |V|^2$), а список смежности — с разреженными (где $|E|$ намного меньше $|V|^2$).

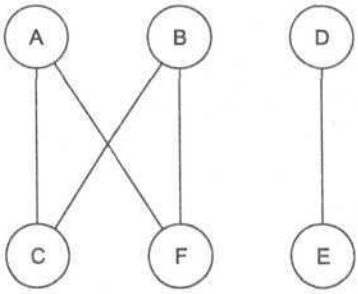


	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	1	0	0	1
C	1	0	1	0	0	0
D	0	0	0	0	1	0
E	0	0	0	1	0	0
F	1	0	0	0	0	0

Рис. 14.4. Представление графа в виде матрицы смежности

Класс `adjacency_matrix` расширяет традиционную структуру данных, позволяя прикреплять объекты к вершинам и ребрам посредством параметров шаблонов свойств. Информацию по использованию внутренних свойств см. в разделе 3.6.

В случае неориентированного графа класс `adjacency_matrix` использует не всю матрицу $|V| \times |V|$, а только ее нижний треугольник (диагональ и ниже), поскольку матрица для неориентированного графа симметрична. Это сокращает расходы на хранение до $(|V| \times |V|)/2$. Представление неориентированного графа в виде матрицы смежности показано на рис. 14.5.



	A	B	C	D	E	F
A	0					
B	0	0				
C	1	1	0			
D	0	0	0	0		
E	0	0	0	1	0	
F	1	1	0	0	0	0

Рис. 14.5. Представление неориентированного графа в виде матрицы смежности

Пример

В листинге 14.11 приведен пример построения графа, изображенного на рис. 14.4, а в листинге 14.12 — изображенного на рис. 14.5.

Листинг 14.11. Создание графа (на рис. 14.4)

```
enum { A, B, C, D, E, F, N };
const char* name = "ABCDEF";

typedef adjacency_matrix<directedS> Graph;
Graph g(N);
add_edge(B, C, g);  add_edge(B, F, g);
add_edge(C, A, g);  add_edge(C, C, g);
add_edge(D, E, g);  add_edge(E, D, g);
add_edge(F, A, g);

std::cout << "набор вершин: ";
print_vertices(g, name);
std::cout << std::endl;
```

```
std::cout << "набор ребер: ";
print_edges(g, name);
std::cout << std::endl;

std::cout << "исходящие ребра: " << std::endl;
print_graph(g, name);
std::cout << std::endl;
```

Эта программа выводит следующее:

```
набор вершин: A B C D E F

набор ребер: (B,C) (B,F) (C,A) (C,C) (D,E) (E,D) (F,A)

исходящие ребра:
A -->
B --> C F
C --> A C
D --> E
E --> D
F --> A
```

Листинг 14.12. Создание графа (на рис. 14.5)

```
enum { A, B, C, D, E, F, N };
const char* name = "ABCDEF";

typedef adjacency_matrix<undirectedS> UGraph;
UGraph ug(N);
add_edge(B, C, ug);
add_edge(B, F, ug);
add_edge(C, A, ug);
add_edge(D, E, ug);
add_edge(F, A, ug);

std::cout << "набор вершин: ";
print_vertices(ug, name);
std::cout << std::endl;

std::cout << "набор ребер: ";
print_edges(ug, name);
std::cout << std::endl;

std::cout << "инцидентные ребра: " << std::endl;
print_graph(ug, name);
std::cout << std::endl;
```

Эта программа выводит следующее:

```
набор вершин: A B C D E F

набор ребер: (C,A) (C,B) (E,D) (F,A) (F,B)

инцидентные ребра:

A <--> C F
B <--> C F
C <--> A B
D <--> E
E <--> D
F <--> A B
```

Где определен

Класс adjacency_matrix находится в boost/graph/adjacency_matrix.hpp.

Параметры шаблона

Ниже приведены параметры шаблона класса adjacency_matrix.

Directed	Селектор для выбора графа: ориентированный или неориентированный. Соответствующие опции: directedS и undirectedS. По умолчанию: directedS.
VertexProperty	Задает внутреннее хранилище свойств вершин. По умолчанию: no_property.
EdgeProperty	Задает внутреннее хранилище свойств ребер. По умолчанию: no_property.
GraphProperty	Задает внутреннее хранилище свойств графа. По умолчанию: no_property.

Модель для

VertexListGraph, EdgeListGraph, IncidenceGraph, AdjacencyGraph, AdjacencyMatrix, VertexMutablePropertyGraph и EdgeMutablePropertyGraph.

Требования к типам

Значение свойства должно быть DefaultConstructible и CopyConstructible.

Ассоциированные типы

Ниже приведены ассоциированные типы класса adjacency_matrix.

- graph_traits<adjacency_matrix>::vertex_descriptor

Тип дескрипторов вершин, ассоциированных с матрицей смежности. (Требуется для Graph.)

- graph_traits<adjacency_matrix>::edge_descriptor

Тип дескрипторов ребер, ассоциированных с матрицей смежности. (Требуется для Graph.)

- graph_traits<adjacency_matrix>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices(). (Требуется для VertexListGraph.)

- graph_traits<adjacency_matrix>::edge_iterator

Тип итераторов, возвращаемых функцией edges(). (Требуется для EdgeListGraph.)

- `graph_traits<adjacency_matrix>::out_edge_iterator`

Тип итераторов, возвращаемых функцией `out_edges()`.

(Требуется для IncidenceGraph.)

- `graph_traits<adjacency_matrix>::adjacency_iterator`

Тип итераторов, возвращаемых функцией `adjacent_vertices()`.

(Требуется для AdjacencyGraph.)

- `graph_traits<adjacency_matrix>::directed_category`

Предоставляет информацию о том, является граф ориентированным (`directed_tag`) или неориентированным (`undirected_tag`).

(Требуется для Graph.)

- `graph_traits<adjacency_matrix>::edge_parallel_category`

Матрица смежности не позволяет вставлять параллельные ребра, так что тип всегда `disallow_parallel_edge_tag`.

(Требуется для Graph.)

- `graph_traits<adjacency_matrix>::vertices_size_type`

Тип для работы с числом вершин в графе.

(Требуется для VertexListGraph.)

- `graph_traits<adjacency_matrix>::edges_size_type`

Тип для работы с числом ребер в графе.

(Требуется для EdgeListGraph.)

- `graph_traits<adjacency_matrix>::degree_size_type`

Тип для работы с числом исходящих ребер в графе.

(Требуется для IncidenceGraph.)

- `property_map<adjacency_matrix, PropertyTag>::type`
  `property_map<adjacency_matrix, PropertyTag>::const_type`

Тип отображения для свойств вершины и ребра графа. Свойство задается аргументом шаблона `PropertyTag` и должно совпадать с одним из свойств, указанных в `VertexProperty` или `EdgeProperty` графа.

(Требуется для PropertyGraph.)

Функции — методы

Ниже приведены функции — методы класса `adjacency_matrix`.

- ```
  adjacency_matrix(vertices_size_type n,
    const GraphProperty& p = GraphProperty());
  ```

Создает графовый объект с `n` вершин и без ребер.

- ```
  template <typename EdgeIterator>
  adjacency_matrix(EdgeIterator first, EdgeIterator last,
    vertices_size_type n,
    const GraphProperty& p = GraphProperty())
  ```

Создает графовый объект с n вершин и ребрами, заданными списком ребер в виде диапазона [*first*, *last*). Тип значения итератора EdgeIterator должен быть std::pair, где тип внутри пары является целым типом. Целые числа соответствуют вершинам и должны находиться в диапазоне [0, *n*).

- ```
  template <typename EdgeIterator, typename
    EdgePropertyIterator>
  adjacency_matrix(EdgeIterator first, EdgeIterator last,
    EdgePropertyIterator ep_iter, vertices_size_type n,
    const GraphProperty& p = GraphProperty())
  ```

Создает графовый объект с n вершин и ребрами, заданными списком ребер в виде диапазона [*first*, *last*), и свойствами, заданными ep_iter в списке свойств ребер. Тип значения итератора EdgeIterator должен быть std::pair, где тип внутри пары является целым типом. Целые числа соответствуют вершинам и должны находиться в диапазоне [0, *n*). Тип значения ep_iter должен быть EdgeProperty.

Функции — не методы

Ниже приведены функции — не методы класса adjacency_matrix.

- ```
  std::pair<vertex_iterator, vertex_iterator>
    vertices(const adjacency_matrix& g)
  ```

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин графа g.

(Требуется для VertexListGraph.)

- ```
  std::pair<edge_iterator, edge_iterator>
    edges(const adjacency_matrix& g)
  ```

Возвращает пару итераторов, обеспечивающих доступ к набору ребер графа g.

(Требуется для EdgeListGraph.)

- ```
  std::pair<adjacency_iterator, adjacency_iterator>
  adjacent_vertices(vertex_descriptor v,
    const adjacency_matrix& g)
  ```

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин, смежных с данной вершиной *v* графа g.

(Требуется для AdjacencyGraph.)

- ```
  std::pair<out_edge_iterator, out_edge_iterator>
  out_edges(vertex_descriptor v, const adjacency_matrix& g)
  ```

Возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам вершины *v* графа g. Если граф неориентированный, эти итераторы обеспечивают доступ ко всем ребрам, инцидентным вершине *v*.

(Требуется для IncidenceGraph.)

- ```
  vertex_descriptor source(edge_descriptor e,
    const adjacency_matrix& g)
  ```

Возвращает начальную вершину ребра *e*.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e, const adjacency_matrix& g)`

Возвращает конечную вершину ребра *e*.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u, const adjacency_matrix& g)`

Возвращает число ребер, исходящих из вершины *u*.

(Требуется для IncidenceGraph.)

- `vertices_size_type num_vertices(const adjacency_matrix& g)`

Возвращает число вершин в графе `g`.

(Требуется для VertexListGraph.)

- `edges_size_type num_edges(const adjacency_matrix& g)`

Возвращает число ребер в графе `g`.

(Требуется для EdgeListGraph.)

- `vertex descriptor_vertex(vertices_size_type n, const adjacency_matrix& g)`

Возвращает *n*-ю вершину в списке вершин графа.

- `std::pair<edge_descriptor, bool> edge(vertex_descriptor u, vertex_descriptor v, const adjacency_matrix& g)`

Возвращает ребро, соединяющее вершину *u* с вершиной *v* в графе `g`.

(Требуется для AdjacencyMatrix.)

- `std::pair<edge_descriptor, bool> add_edge(vertex_descriptor u, vertex_descriptor v, adjacency_matrix& g)`

Добавляет ребро (u, v) к графу и возвращает дескриптор ребра для нового ребра. Если ребро уже присутствует в графе, то повторно оно добавлено не будет и логический флаг будет ложью. Эта операция не «портит» итераторы и дескрипторы графа.

(Требуется для EdgeMutableGraph.)

- `std::pair<edge_descriptor, bool> add_edge(vertex_descriptor u, vertex_descriptor v, const EdgeProperty& p, adjacency_matrix& g)`

Добавляет ребро (u, v) к графу и присоединяет `p` как значение свойства ребра для внутреннего хранения (см. также предыдущую функцию-метод класса `add_edge()`).

- `void remove_edge(vertex_descriptor u, vertex_descriptor v, adjacency_matrix& g)`

Удаляет ребро (u, v) из графа.

(Требуется для EdgeMutableGraph.)

- `void remove_edge(edge_descriptor e, adjacency_matrix& g)`

Удаляет ребро `e` из графа.

(Требуется для EdgeMutableGraph.)

- `void clear_vertex(vertex_descriptor u,`
  `adjacency_matrix& g)`

Удаляет все ребра, исходящие и входящие, для вершины *u*. Вершина остается во множестве вершин графа. Воздействие на корректность дескрипторов и итераторов такое же, как если бы для всех этих ребер отдельно вызывалась `remove_edge()`.

(Требуется для EdgeMutableGraph.)

- ```
  template <typename Property>
  property_map<adjacency_matrix, Property>::type
  get(Property, adjacency_matrix& g)
  ```

Возвращает объект-отображение свойств, указанный с помощью `Property` (свойство). Свойство должно совпадать с одним из свойств, указанных в шаблонном аргументе `VertexProperty` графа.

(Требуется для PropertyGraph.)

- ```
  template <typename Property>
  property_map<adjacency_matrix, Property>::const_type
  get(Property, const adjacency_matrix& g)
  ```

Возвращает объект-отображение свойств, указанный с помощью `Property` (свойство). Свойство должно совпадать с одним из свойств, указанных в шаблонном аргументе `VertexProperty` графа.

(Требуется для PropertyGraph.)

- ```
  template <typename Property, typename X>
  typename property_traits< typename
  property_map<adjacency_matrix, Property>::const_type
    >::value_type
  get(Property, const adjacency_matrix& g, X x)
  ```

Возвращает значение свойства для `x`, где `x` — дескриптор вершины или ребра.

(Требуется для PropertyGraph.)

- ```
  template <typename Property, typename X, typename Value>
  void put(Property, const adjacency_matrix& g, X x,
    const Value& value)
  ```

Устанавливает значение свойства для `x` в `value`, где `x` — дескриптор вершины или ребра. Значение `value` должно быть преобразуемым к типу значения отображения свойства, указанного тегом `Property`.

(Требуется для PropertyGraph.)

- ```
  template <typename GraphProperties,
    typename GraphProperty>
  typename property_value<GraphProperties,
    GraphProperty>::type&
  get_property(adjacency_matrix& g, GraphProperty);
  ```

Возвращает свойство, указанное `GraphProperty`, которое было закреплено за графовым объектом `g`. Класс свойств `property_value` определен в файле **boost/pending/property.hpp**.

- ```
  template <typename GraphProperties,
    typename GraphProperty>
  ```

```
const typename property_value<GraphProperties,
  GraphProperty>::type&
get_property(const adjacency_matrix& g, GraphProperty);
```

Возвращает свойство, указанное GraphProperty, которое закреплено за графовым объектом g. Класс свойств property_value определен в файле boost/pending/property.hpp.

14.2. Вспомогательные классы

14.2.1. graph_traits

```
graph_traits<Graph>
```

Класс graph_traits обеспечивает механизм для доступа к ассоциированным типам графового типа согласно определениям различных графовых концепций BGL (см. раздел 12.1). Когда вы хотите использовать один из ассоциированных типов графа, создайте экземпляр шаблона graph_traits с графовым типом и задайте соответствующий дескриптор типа (typedef). Например, для получения типа дескриптора вершины для некоторого графа можно выполнить следующее:

```
template <typename Graph> void my_graph_algorithm(Graph& g) {
  // Получить экземпляр graph_traits с графовым типом Graph.
  typedef boost::graph_traits<Graph> Traits;
  // Описание типа для доступа к ассоциированному типу.
  typedef typename Traits::vertex_descriptor Vertex;
  // ...
}
```

Неспециализированная версия (по умолчанию) шаблона класса graph_traits подразумевает, что графовый тип предоставляет составные дескрипторы типов для всех ассоциированных типов. Эта версия приведена в листинге 14.13.

Листинг 14.13. *Неспециализированная версия класса graph_traits*

```
namespace boost {
  template <typename G>
  struct graph_traits {
    // итераторы:
    typedef typename G::vertex_descriptor          vertex_descriptor;
    typedef typename G::edge_descriptor            edge_descriptor;
    typedef typename G::adjacency_iterator         adjacency_iterator;
    typedef typename G::out_edge_iterator          out_edge_iterator;
    typedef typename G::in_edge_iterator           in_edge_iterator;
    typedef typename G::vertex_iterator            vertex_iterator;
    typedef typename G::edge_iterator              edge_iterator;
    // категории
    typedef typename G::directed_category          directed_category;
    typedef typename G::edge_parallel_category edge_parallel_category;
    typedef typename G::traversal_category         traversal_category;
    // типы размеров
    typedef typename G::vertices_size_type         vertices_size_type;
    typedef typename G::edges_size_type            edges_size_type;
    typedef typename G::degree_size_type           degree_size_type;
  };
} // namespace boost
```

С другой стороны, graph_traits может быть специализирован по графовому типу. Например, следующий код специализирует graph_traits для структуры Graph библиотеки Stanford GraphBase. Полностью оболочка для SGB-графов описана в заголовочном файле boost/graph/stanford_graph.hpp.

```
namespace boost {
  template <>
  struct graph_traits<Graph*> {
    // ...
  };
}
```

Если тип графа является шаблоном класса, то класс graph_traits может быть частично специализирован. Это означает, что еще остаются некоторые «свободные» параметры. Ниже приведена частичная специализация graph_traits для параметризованного типа GRAPH из LEDA. Полностью интерфейс оболочки для этого типа находится в файле boost/graph/leda_graph.hpp.

```
namespace boost {
  template <typename vtype, typename etype>
  struct graph_traits< GRAPH<vtype,etype> > {
    // ...
  };
}
```

Ни одна конкретная графовая концепция не требует, чтобы были определены все ассоциированные типы. При реализации графового класса, который должен удовлетворять одной или более графовым концепциям, для ассоциированных типов, не затребованных этими концепциями, можно использовать void в качестве типа (когда используются составные дескрипторы типов внутри графового класса) или оставить typedef вне специализации graph_traits для этого графового класса.

Теги категорий

Категория directed_category должна быть дескриптором одного из следующих двух типов.

```
namespace boost {
  struct directed_tag { };
  struct undirected_tag { };
}
```

Категория edge_parallel_category должна быть дескриптором одного из следующих двух типов.

```
namespace boost {
  struct allow_parallel_edge_tag {};
  struct disallow_parallel_edge_tag {};
}
```

Категория traversal_category должна быть дескриптором одного из следующих типов или типа, который наследует от одного из этих классов.

```
namespace boost {
  struct incidence_graph_tag { };
  struct adjacency_graph_tag { };
  struct bidirectional_graph_tag :
```

```
    public virtual incidence_graph_tag { };
  struct vertex_list_graph_tag :
    public virtual incidence_graph_tag,
    public virtual adjacency_graph_tag { };
  struct edge_list_graph_tag { };
  struct vertex_and_edge_list_graph_tag :
    public virtual edge_list_graph_tag,
    public virtual vertex_list_graph_tag { };
  struct adjacency_matrix_tag { };
}
```

Параметры шаблона

Ниже приведен параметр шаблона класса `graph_traits`.

`Graph`	Графовый тип, модель для категории `Graph`.

Где определен

Класс `graph_traits` находится в файле `boost/graph/graph_traits.hpp`.

Методы

Ниже приведены методы класса `graph_traits`.

- `graph_traits::vertex_descriptor`

Тип дескрипторов вершин, ассоциированных с `Graph`.

- `graph_traits::edge_descriptor`

Тип дескрипторов ребер, ассоциированных с `Graph`.

- `graph_traits::vertex_iterator`

Тип итераторов, возвращаемых функцией `vertices()`.

- `graph_traits::edge_iterator`

Тип итераторов, возвращаемых функцией `edges()`.

- `graph_traits::out_edge_iterator`

Тип итераторов, возвращаемых функцией `out_edges()`.

- `graph_traits::adjacency_iterator`

Тип итераторов, возвращаемых функцией `adjacent_vertices()`.

- `graph_traits::directed_category`

Сообщает, является граф ориентированным или неориентированным.

- `graph_traits::edge_parallel_category`

Сообщает, позволяет ли граф иметь параллельные ребра.

- `graph_traits::traversal_category`

Сообщает, какие виды обхода обеспечиваются графом.

- `graph_traits::vertices_size_type`

Беззнаковый целый тип, используемый для работы с количеством вершин в графе.

- `graph_traits::edges_size_type`

Беззнаковый целый тип, используемый для работы с количеством ребер в графе.

- graph_traits::degree_size_type

Беззнаковый целый тип, используемый для работы с количеством исходящих ребер в графе.

14.2.2. adjacency_list_traits

```
adjacency_list_traits<EdgeList, VertexList, Directed>
```

Класс adjacency_list_traits предоставляет альтернативный метод для доступа к некоторым ассоциированным типам класса adjacency_list. Главной причиной создания этого класса является то, что иногда требуются свойства графа, значениями которых являются дескрипторы вершин или ребер. Если вы попытаетесь использовать для этого graph_traits, возникнет проблема с взаимно-рекурсивными типами. Для решения этой проблемы предлагается класс adjacency_list_traits, который предоставляет пользователю доступ к типам дескрипторов вершин и ребер, не требуя задания типов свойств графа.

```
template <typename EdgeList, typename VertexList, typename Directed>
  struct adjacency_list_traits {
    typedef ... vertex_descriptor;
    typedef ... edge_descriptor;
    typedef ... directed_category;
    typedef ... edge_parallel_category;
  };
```

Где определен

Класс adjacency_list_traits находится в файле boost/graph/adjacency_list.hpp.

Параметры шаблона

Ниже приведены параметры шаблона класса adjacency_list_traits.

EdgeList	Тип селектора для реализации контейнера ребер. По умолчанию: vecS.
VertexList	Тип селектора для реализации контейнера вершин. По умолчанию: vecS.
Directed	Селектор для выбора ориентированного или неориентированного графа. По умолчанию: directedS.

Модель для

DefaultConstructible и Assignable.

Методы

Ниже приведены методы класса adjacency_list_traits.

- adjacency_list_traits::vertex_descriptor

Тип объектов, используемых для идентификации вершин графа.

- adjacency_list_traits::edge_descriptor

Тип объектов, используемых для идентификации ребер графа.

- `adjacency_list_traits::directed_category`

Сообщает, является ли граф неориентированным (`undirected_tag`) или ориентированным (`directed_tag`).

- `adjacency_list_traits::edge_parallel_category`

Сообщает, позволяет ли граф добавлять параллельные ребра (`allow_parallel_edge_tag`) или автоматически удаляет их (`disallow_parallel_edge_tag`).

Смотри также

Дополнительная информация находится в описании класса `adjacency_list`.

14.2.3. adjacency_matrix_traits

```
adjacency_matrix_traits<Directed>
```

Класс `adjacency_matrix_traits` предоставляет альтернативный метод для доступа к некоторым ассоциированным типам класса `adjacency_matrix`. Главной причиной создания этого класса является то, что иногда требуются свойства графа, значениями которых являются дескрипторы вершин или ребер. Если вы попытаетесь использовать `graph_traits` для этого, возникнет проблема с взаимно-рекурсивными типами. Для решения этой проблемы предлагается класс `adjacency_matrix_traits`, который предоставляет пользователю доступ к типам дескрипторов вершин и ребер, не требуя задания типов свойств графа.

```
template <typename Directed>
  struct adjacency_matrix_traits {
    typedef ... vertex_descriptor;
    typedef ... edge_descriptor;
    typedef ... directed_category;
    typedef ... edge_parallel_category;
  };
```

Где определен

`adjacency_matrix_traits` находится в файле boost/graph/adjacency_matrix.hpp.

Параметры шаблона

Ниже приведен параметр шаблона класса `adjacency_matrix_traits`.

`Directed`	Указывает, является граф ориентированным или неориентированным. По умолчанию: `directedS`.

Модель для

DefaultConstructible и Assignable.

Методы

Ниже приведены методы класса `adjacency_matrix_traits`.

- `adjacency_matrix_traits::vertex_descriptor`

Тип объектов, используемых для идентификации вершин графа.

- `adjacency_matrix_traits::edge_descriptor`

Тип объектов, используемых для идентификации ребер графа.

- `adjacency_matrix_traits::directed_category`

Сообщает, является граф неориентированным (`undirected_tag`) или ориентированным (`directed_tag`).

- `adjacency_matrix_traits::edge_parallel_category`

Матрица смежности не позволяет добавлять параллельные ребра, поэтому здесь используется `disallow_parallel_edge_tag`.

Смотри также

Дополнительная информация находится в описании класса `adjacency_matrix`.

14.2.4. property_map

```
property_map<Graph, PropertyTag>
```

Класс `property_map` — класс свойств для доступа к типу отображения свойства, хранящемуся внутри графа. Специализация этого класса свойств требуется для типов, которые моделируют концепцию PropertyGraph.

Пример

В следующем примере (листинг 14.14) создается граф с внутренним свойством для хранения имен вершин, а затем осуществляется доступ к типу свойств имен вершин с помощью класса свойств `property_map`. Объект – отображение свойства получается из графа, с использованием функции `get()`.

Листинг 14.14. Пример внутреннего хранения имен вершин

```
    < property-map-traits-eg.cpp > ≡
#include <boost/config.hpp>
#include <string>
#include <boost/graph/adjacency_list.hpp>
int main() {
  using namespace boost;
  typedef adjacency_list < listS, listS, directedS,
    property < vertex_name_t, std::string > >graph_t;
  graph_t g;
  graph_traits < graph_t >::vertex_descriptor u = add_vertex(g);
  property_map < graph_t, vertex_name_t >::type
    name_map = get(vertex_name, g);
  name_map[u] = "Joe";
  std::cout << name_map[u] << std::endl;
  return EXIT_SUCCESS;
   }
```

Программа выводит следующее:

```
Joe
```

Где определен

Класс `property_map` находится в файле **boost/graph/properties.hpp**.

Параметры шаблона

Ниже приведены параметры шаблона класса `property_map`.

`Graph`	Тип графа, который должен быть моделью `PropertyGraph`.
`PropertyTag`	Теговый класс для задания свойства.

Модель для

Нет.

Общедоступные базовые классы

Нет.

Ассоциированные типы

Ниже приведены ассоциированные типы класса `property_map`.

- `property_map<Graph, PropertyTag>::type`

  Тип изменяемого отображения свойства для доступа к внутреннему свойству, указанному `PropertyTag`.

- `property_map<Graph, PropertyTag>::const_type`

  Тип константного отображения свойства для доступа к внутреннему свойству, указанному `PropertyTag`.

Функции — методы класса

Нет.

Функции — не методы класса

Нет.

14.2.5. property

```
property<PropertyTag, T, NextProperty>
```

Класс `property` может использоваться с классами `adjacency_list` и `adjacency_matrix` для указания видов свойств, закрепляемых за вершинами и ребрами графа и за самим графовым объектом.

Параметры шаблона

Ниже приведены параметры шаблона класса `property`.

`PropertyTag`	Тип для обозначения (придания уникального имени) свойству. Имеются несколько предопределенных тегов, и несложно определить новые. Для удобства BGL также предоставляет предопределенные объекты теговых типов (в данном случае — значений перечисления `enum`) для их использования в качестве аргументов функций, которые ожидают объекты теговых типов (например, как функции отображения свойства `get()` к `adjacency_list`).
`T`	Этот тип указывает тип значений свойств.

NextProperty	Этот параметр позволяет типам свойств быть вложенными, так что с графом может быть связано любое количество свойств. По умолчанию: no_property.

Где определен

Класс property находится в файле boost/pending/property.hpp.

Теги свойств

Следующие теги свойств (листинг 14.15) определены в boost/graph/properties.hpp.

Листинг 14.15. Определения тегов свойств

```
namespace boost {
  // Теги свойств ребер:
  enum edge_name_t { edge_name };                 // имя
  enum edge_weight_t { edge_weight };             // вес
  enum edge_index_t { edge_index };               // индекс
  enum edge_capacity_t { edge_capacity };         // мощность
  enum edge_residual_capacity_t { edge_residual_capacity };
              // остаточная мощность
  enum edge_reverse_t { edge_reverse };           // обратное
  // Теги свойств вершин:
  enum vertex_name_t { vertex_name };             // имя
  enum vertex_distance_t { vertex_distance };     // расстояние до вершины
  enum vertex_index_t { vertex_index };           // индекс
  enum vertex_color_t { vertex_color };           // цвет
  enum vertex_degree_t { vertex_degree };         // степень
  enum vertex_out_degree_t { vertex_out_degree };
              // степень по исходящим ребрам
  enum vertex_in_degree_t { vertex_in_degree };
              // степень по входящим ребрам
  enum vertex_discover_time_t { vertex_discover_time };
              // время посещения
  enum vertex_finish_time_t { vertex_finish_time };
              // время окончания обработки
  // Тег свойства графа:
  enum graph_name_t { graph_name };               // имя

  BOOST_INSTALL_PROPERTY(vertex, index);
  BOOST_INSTALL_PROPERTY(edge, index);
  // ...
}
```

14.3. Графовые адаптеры

14.3.1. edge_list

```
edge_list<EdgeIterator, ValueType, DiffType>
```

Класс edge_list — это адаптер, который превращает пару итераторов ребер в класс, моделирующий EdgeListGraph. Тип значения (value_type) итератора ребер должен быть std::pair (или, по крайней мере, иметь методы first (первый) и second

(второй)). Типы first_type и second_type должны быть одинаковыми и использоваться в качестве дескриптора вершины графа. Типы шаблонных параметров ValueType и DiffType нужны только в случае, если ваш компилятор не поддерживает частичную специализацию. В противном случае они имеют правильные значения по умолчанию.

Пример

См. раздел 5.3, где дан пример использования edge_list.

Параметры шаблона

Ниже приведены параметры шаблона класса edge_list.

EdgeIterator	Модель для InputIterator, чей value_type должен быть парой дескрипторов вершин.
ValueType	Тип значения для EdgeIterator. По умолчанию: std::iterator_traits<EdgeIterator>::value_type.
DiffType	Тип разности для EdgeIterator. По умолчанию: std::iterator_traits<EdgeIterator>::difference_type.

Модель для

Класс edge_list поддерживается концепцией EdgeListGraph.

Где определен

Класс edge_list находится в файле boost/graph/edge_list.hpp.

Ассоциированные типы

Ниже приведены ассоциированные типы класса edge_list.

- graph_traits<edge_list>::vertex_descriptor

  Тип дескрипторов вершин, ассоциированный с edge_list. Это тот же самый тип, что и first_type для пары std::pair, являющейся типом значения для EdgeIterator.

- graph_traits<edge_list>::edge_descriptor

  Тип дескрипторов ребер, ассоциированный с edge_list.

- graph_traits<edge_list>::edge_iterator

  Тип итераторов, возвращаемый функцией edges(). Категория iterator_category для итератора ребер та же, что и у EdgeIterator.

Функции — методы

Ниже приведена функция — метода класса edge_list.

- edge_list(EdgeIterator first, EdgeIterator last)

  Создает графовый объект из *n* вершин с ребрами, заданными в списке ребер из диапазона [*first*, *last*).

Функции — не методы

Ниже приведены функции — не методы класса edge_list.

- `std::pair<edge_iterator, edge_iterator>`
  `edges(const edge_list& g)`

  Возвращает пару итераторов, обеспечивающих доступ к набору ребер графа g.

- `vertex_descriptor source(edge_descriptor e,`
  `const edge_list& g)`

  Возвращает начальную вершину ребра *e*.

- `vertex_descriptor target(edge_descriptor e,`
  `const edge_list& g)`

  Возвращает конечную вершину ребра *e*.

14.3.2. reverse_graph

```
reverse_graph<BidirectionalGraph>
```

Класс reverse_graph меняет местами входящие и исходящие ребра BidirectionalGraph (двунаправленного) графа, эффективно транспонируя граф. Построение обращенного графа выполняется за постоянное время, таким образом обеспечивая высокоэффективный способ для получения транспонированного представления графа.

Пример

Пример в листинге 14.16 взят из файла examples/reverse-graph-eg.cpp.

Листинг 14.16. Обращение направлений ребер графа

```
typedef adjacency_list < vecS, vecS, bidirectionalS > Graph;

Graph G(5);
add_edge(0, 2, G);  add_edge(1, 1, G);  add_edge(1, 3, G);
add_edge(1, 4, G);  add_edge(2, 1, G);  add_edge(2, 3, G);
add_edge(2, 4, G);  add_edge(3, 1, G);  add_edge(3, 4, G);
add_edge(4, 0, G);  add_edge(4, 1, G);

std::cout << "исходный граф:" << std::endl;
print_graph(G, get(vertex_index, G));

std::cout << std::endl << "обращенный граф:" << std::endl;
print_graph(make_reverse_graph(G), get(vertex_index, G));
```

Эта программа выводит следующее:

```
исходный граф:
0 --> 2
1 --> 1 3 4
2 --> 1 3 4
3 --> 1 4
4 --> 0 1

обращенный граф:
0 --> 4
1 --> 1 2 3 4
2 --> 0
3 --> 1 2
4 --> 1 2 3
```

Параметры шаблона

Ниже приведен параметр шаблона класса reverse_graph.

BidirGraph	Тип графа, для которого строится адаптер.

Модель для

Класс reverse_graph поддерживается концепциями BidirectionalGraph и (необязательно) VertexListGraph и PropertyGraph.

Где определен

Класс reverse_graph находится в файле boost/graph/reverse_graph.hpp.

Ассоциированные типы

Ниже приведены ассоциированные типы класса reverse_graph.

- graph_traits<reverse_graph>::vertex_descriptor

Тип дескрипторов вершин, ассоциированных с обращенным графом.
(Требуется для Graph.)

- graph_traits<reverse_graph>::edge_descriptor

Тип дескрипторов ребер, ассоциированных с обращенным графом.
(Требуется для Graph.)

- graph_traits<reverse_graph>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices().
(Требуется для VertexListGraph.)

- graph_traits<reverse_graph>::edge_iterator

Тип итераторов, возвращаемых функцией edges().
(Требуется для EdgeListGraph.)

- graph_traits<reverse_graph>::out_edge_iterator

Тип итераторов, возвращаемых функцией out_edges().
(Требуется для IncidenceGraph.)

- graph_traits<reverse_graph>::adjacency_iterator

Тип итераторов, возвращаемых функцией adjacent_vertices().
(Требуется для BidirectionalGraph.)

- graph_traits<reverse_graph>::directed_category

Предоставляет информацию о том, является граф ориентированным или неориентированным.
(Требуется для Graph.)

- graph_traits<reverse_graph>::edge_parallel_category

Сообщает, позволяет ли граф осуществлять вставку параллельных ребер (ребер с одинаковыми начальными и одинаковыми конечными вершинами). Теги: allow_parallel_edge_tag и disallow_parallel_edge_tag. Варианты графов с параметрами setS и hash_setS всегда используют disallow_parallel_edge_tag, тогда как другие могут позволить включение параллельных ребер.
(Требуется для Graph.)

- `graph_traits<reverse_graph>::traversal_category`

Категория обхода отражает, какие возможные виды итераторов поддерживаются графовым классом. Для `reverse_graph` это будет тот же тип, что и для `traversal_category` исходного графа.

(Требуется для Graph.)

- `graph_traits<reverse_graph>::vertices_size_type`

Тип для работы с числом вершин в графе.

(Требуется для VertexListGraph.)

- `graph_traits<reverse_graph>::edge_size_type`

Тип для работы с числом ребер в графе.

(Требуется для EdgeListGraph.)

- `graph_traits<reverse_graph>::degree_size_type`

Тип для работы с числом ребер, инцидентных вершине в графе.

(Требуется для IncidenceGraph.)

- `property_map<reverse_graph, Property>::type`
  `property_map<reverse_graph, Property>::const_type`

Тип отображения свойств для свойств вершин и ребер графа. Конкретное свойство указывается шаблонным аргументом `Property` и должно совпадать с одним из свойств в `VertexProperty` или `EdgeProperty` графа.

(Требуется для PropertyGraph.)

Функции — методы

Ниже приведена функция — метод класса `reverse_graph`.

- `reverse_graph(BidirectionalGraph& g)`

Конструктор. Создает обращенное (транспонированное) представление графа `g`.

Функции — не методы класса

Ниже приведены функции — не методы класса `reverse_graph`.

- `template <class BidirectionalGraph>`
  `reverse_graph<BidirectionalGraph>`
  `make_reverse_graph(BidirectionalGraph& g)`

Вспомогательная функция для создания обращенного графа.

- `std::pair<vertex_iterator, vertex_iterator>`
  `vertices(const reverse_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин графа `g`.

(Требуется для VertexListGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `out_edges(vertex_descriptor v, const reverse_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к набору исходящих ребер вершины `v` графа `g`. Эти исходящие ребра соответствуют входящим ребрам исходного графа.

(Требуется для IncidenceGraph.)

- `std::pair<in_edge_iterator, in_edge_iterator>`
  `in_edges(vertex_descriptor v, const reverse_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к набору входящих ребер вершины `v` графа `g`. Эти входящие ребра соответствуют исходящим ребрам исходного графа.

(Требуется для BidirectionalGraph.)

- `std::pair<adjacency_iterator, adjacency_iterator>`
  `adjacent_vertices(vertex_descriptor v,`
  `const reverse_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к смежным вершинам вершины `v` графа `g`.

(Требуется для AdjacencyGraph.).

- `vertex_descriptor source(edge_descriptor e,`
  `const reverse_graph& g)`

Возвращает начальную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,`
  `const reverse_graph& g)`

Возвращает конечную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,`
  `const reverse_graph& g)`

Возвращает число ребер, исходящих из вершины `u`.

(Требуется для IncidenceGraph.)

- `degree_size_type in_degree(vertex_descriptor u,`
  `const reverse_graph& g)`

Возвращает число ребер, входящих в вершину `u`. Операция доступна, только если был указан селектор `bidirectionalS`.

(Требуется для BidirectionalGraph).

- `vertices_size_type num_vertices(const reverse_graph& g)`

Возвращает число вершин в графе `g`.

(Требуется для VertexListGraph.)

- `vertex_descriptor vertex(vertices_size_type n,`
  `const reverse_graph& g)`

Возвращает *n*-ю вершину в списке вершин графа.

- `std::pair<edge_descriptor, bool>`
  `edge(vertex_descriptor u, vertex_descriptor v,`
  `const reverse_graph& g)`

Возвращает ребро, соединяющее вершину `u` с вершиной `v`.

(Требуется для AdjacencyMatrix.)

- `template <class Property>`
  `property_map<reverse_graph, Property>::type`

```
  get(Property, reverse_graph& g)
template <class Property>
  property_map<reverse_graph, Tag>::const_type
  get(Property, const reverse_graph& g)
```

Возвращает объект-отображение свойств, указанный с помощью Property. Свойство должно совпадать с одним из свойств, указанных в шаблонном аргументе VertexProperty графа.

(Требуется для PropertyGraph.)

- ```
  template <class Property, class X>
  typename property_traits<property_map<reverse_graph,
    Property>::const_type>::value_type
  get(Property, const reverse_graph& g, X x)
  ```

Возвращает значение свойства для x, где x — дескриптор вершины или ребра.

- ```
  template <class Property, class X, class Value>
  void put(Property, const reverse_graph& g, X x,
    const Value& value)
  ```

Устанавливает значение свойства для x в value, где x — дескриптор вершины или ребра. Значение value должно быть преобразуемым к типу typename property_traits<property_map<reverse_graph, Property>::type>::value_type

- ```
  template <class GraphProperties, class GraphProperty>
  typename property_value<GraphProperties,
    GraphProperty>::type&
  get_property(reverse_graph& g, GraphProperty);
  ```

Возвращает свойство, указанное GraphProperty, которое относится к графовому объекту. Класс свойств property_value определен в заголовочном файле boost/pending/property.hpp.

- ```
  template <class GraphProperties, class GraphProperty>
  const typename property_value<GraphProperties,
    GraphProperty>::type&
  get_property(const reverse_graph& g, GraphProperty);
  ```

Возвращает свойство, указанное GraphProperty, которое относится к графовому объекту. Класс свойств property_value определен в заголовочном файле boost/pending/property.hpp.

14.3.3. filtered_graph

```
filtered_graph<Graph, EdgePredicate, VertexPredicate>
```

Класс filtered_graph является адаптером, который создает фильтрованное представление графа. Функция-предикат для ребер и вершин определяет, какие вершины и ребра исходного графа показывать в фильтрованном графе. Любая вершина, для которой предикатная функция возвращает ложь, и любое ребро, для которого предикатная функция возвращает ложь, будут показываться удаленными в результирующем представлении графа. Класс filtered_graph не создает копии исходного графа, но использует ссылки на него. Время жизни исходного графа должно быть дольше любого использования фильтрованного

графа. Объект filtered_graph не изменяет структуру исходного графа, хотя свойства исходного графа можно изменить с помощью отображения свойств фильтрованного графа.

Пример

Следующий функциональный объект, определяемый в листинге 14.17, является примером предиката, который отфильтровывает ребра с неположительным весом.

Листинг 14.17. Предикат, определяющий, имеет ли ребро положительный вес

```
template <typename EdgeWeightMap>
struct positive_edge_weight { // положительный вес ребра?
  positive_edge_weight() { }
  positive_edge_weight(EdgeWeightMap weight) : m_weight(weight) { }
  template <typename Edge>
  bool operator()(const Edge& e) const {
    return 0 < boost::get(m_weight, e);
  }
  EdgeWeightMap m_weight;
};
```

Пример в листинге 14.18 использует фильтрованный граф с описанным выше предикатом positive_edge_weight для создания фильтрованного представления небольшого графа. Ребра (A, C), (C, E) и (E, C) имеют нулевой вес и потому не появляются в фильтрованном графе.

Листинг 14.18. Граф, фильтрованный с помощью предиката

```
typedef adjacency_list<vecS, vecS, directedS,
  no_property, property<edge_weight_t, int> > Graph;
typedef property_map<Graph, edge_weight_t>::type EdgeWeightMap;

enum { A, B, C, D, E, N };
const char* name = "ABCDE";
Graph g(N);
add_edge(A, B, 2, g);  add_edge(A, C, 0, g);
add_edge(C, D, 1, g);  add_edge(C, E, 0, g);
add_edge(D, B, 3, g);  add_edge(E, C, 0, g);

positive_edge_weight<EdgeWeightMap> filter(get(edge_weight, g));
filtered_graph<Graph, positive_edge_weight<EdgeWeightMap> >
  fg(g, filter);

std::cout << "отфильтрованный набор ребер: ";
print_edges(fg, name);

std::cout << "отфильтрованные исходящие ребра:" << std::endl;
print_graph(fg, name);
```

Эта программа выводит следующее:

```
отфильтрованный набор ребер: (A,B) (C,D) (D,B)
отфильтрованные исходящие ребра:
A --> B
B -->
C --> D
D --> B
E -->
```

Где определен

Класс filtered_graph находится в файле boost/graph/filtered_graph.hpp.

Параметры шаблона

Ниже приведены параметры шаблона класса filtered_graph.

Graph	Графовый тип для адаптации.
EdgePredicate	Функциональный объект, выбирающий, какие ребра исходного графа будут присутствовать в отфильтрованном графе. Должен быть моделью Predicate. Тип аргумента — тип дескриптора ребра графа. Также предикат должен быть DefaultConstructible
VertexPredicate	Функциональный объект, выбирающий, какие вершины исходного графа будут присутствовать в отфильтрованном графе. Должен быть моделью Predicate. Тип аргумента — тип дескриптора вершины графа. Также предикат должен быть DefaultConstructible. По умолчанию: keep_all (сохранить все)

Модель для

Концепции, которые моделирует filtered_graph<Graph,EP,VP>, зависят от типа Graph. Если Graph моделирует VertexListGraph, EdgeListGraph, IncidenceGraph, BidirectionalGraph, AdjacencyGraph или PropertyGraph, то же самое делает и filtered_graph<Graph,EP,VP>.

Ассоциированные типы

Ниже приведены ассоциированные типы класса filtered_graph.

- graph_traits<filtered_graph>::vertex_descriptor

Тип дескриптора вершины, ассоциированный с filtered_graph.

(Требуется для Graph.)

- graph_traits<filtered_graph>::edge_descriptor

Тип дескриптора ребра, ассоциированный с filtered_graph.

(Требуется для Graph.)

- graph_traits<filtered_graph>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices(). Тип vertex_iterator — тот же самый, что и для исходного графа.

(Требуется для VertexListGraph.)

- graph_traits<filtered_graph>::edge_iterator

Тип итераторов, возвращаемых функцией edges(). Итератор моделирует концепцию MultiPassInputIterator.

(Требуется для EdgeListGraph.)

- graph_traits<filtered_graph>::out_edge_iterator

Тип итераторов, возвращаемых функцией out_edges(). Итератор моделирует концепцию MultiPassInputIterator.

(Требуется для IncidenceGraph.)

- graph_traits<filtered_graph>::in_edge_iterator

Тип итераторов, возвращаемых функцией in_edges(). Итератор моделирует концепцию MultiPassInputIterator.

(Требуется для BidirectionalGraph.)

- graph_traits<filtered_graph>::adjacency_iterator

Тип итераторов, возвращаемых функцией adjacent_vertices(). Этот итератор моделирует ту же концепцию, что и итератор исходящих вершин.

(Требуется для AdjacencyGraph.)

- graph_traits<filtered_graph>::directed_category

Сообщает, является граф неориентированным (undirected_tag) или ориентированным (directed_tag).

(Требуется для Graph.)

- graph_traits<filtered_graph>::edge_parallel_category

Сообщает, позволяет ли граф осуществлять вставку параллельных ребер (ребер с одинаковыми начальными и одинаковыми конечными вершинами). Тот же самый, что и edge_parallel_category исходного графа.

(Требуется для Graph.)

- graph_traits<filtered_graph>::vertices_size_type

Используется для работы с количеством вершин в графе.

(Требуется для VertexListGraph.)

- graph_traits<filtered_graph>::edges_size_type

Используется для работы с количеством ребер в графе.

(Требуется для EdgeListGraph.)

- graph_traits<filtered_graph>::degree_size_type

Используется для работы с количеством исходящих ребер вершины.

(Требуется для IncidenceGraph.)

- property_map<filtered_graph, PropertyTag>::type
  property_map<filtered_graph, PropertyTag>::const_type

Тип отображения свойств вершины или ребра в графе. Типы отображения в адаптированном графе те же, что и в исходном.

(Требуется для PropertyGraph.)

Функции — методы

Ниже приведены функции — методы класса filtered_graph.

- filtered_graph(Graph& g, EdgePredicate ep)

Конструктор для представления графа g с отфильтрованными ребрами на основании предиката ep.

- filtered_graph(Graph& g, EdgePredicate ep,
  VertexPredicate vp)

Конструктор для отфильтрованного представления графа g на основании предиката для ребер — ep, и для вершин — vp.

Функции — не методы

Функциональность, поддерживаемая классом filtered_graph, зависит от лежащего в основе исходного графа. Например, если тип Graph не реализует in_edges(), именно это будет делать отфильтрованный граф. Ниже перечислены возможные функции, которые filtered_graph может поддерживать, если задан тип Graph, являющийся моделью VertexListGraph, EdgeListGraph, IncidenceGraph, BidirectionalGraph, AdjacencyGraph, PropertyGraph и BidirectionalGraph.

- `std::pair<vertex_iterator, vertex_iterator>`
  `vertices(const filtered_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин графа g.

(Требуется для VertexListGraph.)

- `std::pair<edge_iterator, edge_iterator>`
  `edges(const filtered_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к набору ребер графа g.

(Требуется для EdgeListGraph.)

- `std::pair<adjacency_iterator, adjacency_iterator>`
  `adjacent_vertices(vertex_descriptor v,`
  `const filtered_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к смежным вершинам вершины v в графе g.

(Требуется для AdjacencyGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `out_edges(vertex_descriptor v, const filtered_graph& g)`

Возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам вершины v в графе g. Если граф неориентированный, эти итераторы предоставляют доступ ко всем ребрам, инцидентным вершине v.

(Требуется для IncidenceGraph.)

- `vertex_descriptor source(edge_descriptor e,`
  `const filtered_graph& g)`

Возвращает начальную вершину для ребра e.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,`
  `const filtered_graph& g)`

Возвращает конечную вершину для ребра e.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,`
  `const filtered_graph& g)`

Возвращает число ребер, исходящих из вершины u.

(Требуется для IncidenceGraph.)

- `vertices_size_type num_vertices(const filtered_graph& g)`

Возвращает число вершин в нижележащем графе `g`.

(Требуется для `VertexListGraph`.)

- `edges_size_type num_edges(const filtered_graph& g)`

Возвращает число ребер в графе `g`.

(Требуется для `EdgeListGraph`.)

- ```
  template <typename Property>
    property_map<filtered_graph, Property>::type
    get(Property, filtered_graph& g)
  template <typename Property>
    property_map<filtered_graph, Property>::const_type
    get(Property, const filtered_graph& g)
  ```

Возвращает объект-отображение свойств, заданный `Property`. Свойство `Property` должно совпадать с одним из свойств в шаблонном аргументе `VertexProperty` графа.

(Требуется для `PropertyGraph`.)

- ```
  template <typename Property, typename X>
  typename property_traits<
    typename property_map<filtered_graph,
    Property>::const_type
  >::value_type
  get(Property, const filtered_graph& g, X x)
  ```

Возвращает значение свойства для `x`, где `x` — дескриптор вершины или ребра.

(Требуется для `PropertyGraph`.)

- ```
  template <typename Property, typename X, typename Value>
  void put(Property, const filtered_graph& g, X x,
    const Value& value)
  ```

Устанавливает значение свойства для `x` в `value`. Здесь `x` — дескриптор вершины или ребра. Значение должно быть преобразуемо в тип значения указанного свойства.

(Требуется для `PropertyGraph`.)

14.3.4. Указатель на SGB Graph

`Graph*`

Заголовочный файл `boost/graph/stanford_graph.hpp` из BGL адаптирует Stanford GraphBase-указатель (SGB) [22] на `Graph` в граф, совместимый с BGL. Заметим, что класс графового адаптера не используется, а `Graph*` из SGB сам становится моделью нескольких графовых концепций (см. раздел «Модель для» ниже) с помощью определения нескольких перегруженных функций.

Обязательно применяйте файл изменений `PROTOTYPES` к вашей установке SGB, с тем чтобы заголовочные файлы SGB соответствовали ANSI C (и, значит, могли компилироваться компилятором C++).

Мы благодарны Андреасу Шереру за помощь в реализации и документировании адаптера `Graph*` из библиотеки SGB.
  ```

Пример

Примеры см. в файлах example/miles_span.cpp, example/roget_components.cpp и example/girth.cpp.

Параметры шаблона

Нет.

Модель для

VertexListGraph, IncidenceGraph, AdjacencyGraph и PropertyGraph. Набор тегов свойств, который можно использовать для SGB-графа, дан ниже в разделе «Свойства вершин и ребер».

Где определен

Указатель на SGB Graph находится в файле boost/graph/stanford_graph.hpp.

Ассоциированные типы

Ниже приведены ассоциированные типы указателя на SGB Graph.

- graph_traits<Graph*>::vertex_descriptor

Тип дескриптора вершин, ассоциированный с SGB Graph*. Мы используем тип Vertex* в качестве дескриптора вершины (где Vertex — typedef в заголовочном файле gb_graph.h.)

(Требуется для Graph.)

- graph_traits<Graph*>::edge_descriptor

Тип дескриптора ребер, ассоциированный с SGB Graph*. Используется тип boost::sgb_edge_type. В дополнение к поддержке всех требуемых в BGL операций дескриптора вершины класс boost::sgb_edge имеет следующий конструктор: sgb_edge::sgb_edge(Arc* arc, Vertex* source).

(Требуется для IncidenceGraph.)

- graph_traits<Graph*>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices(). Этот итератор должен моделировать RandomAccessIterator.

(Требуется для VertexListGraph.)

- graph_traits<Graph*>::out_edge_iterator

Тип итераторов, возвращаемых функцией out_edges(). Если EdgeList=vecS, этот итератор моделирует MultiPassInputIterator.

(Требуется для IncidenceGraph.)

- graph_traits<Graph*>::adjacency_iterator

Тип итераторов, возвращаемых функцией adjacent_vertices(). Этот итератор моделирует ту же концепцию, что и out_edge_iterator.

(Требуется для AdjacencyGraph.)

- graph_traits<Graph*>::directed_category

Предоставляет информацию о том, является граф ориентированным или неориентированным. Так как SGB Graph* — ориентированный, этот тип — directed_tag.

(Требуется для Graph.)

- `graph_traits<Graph*>::edge_parallel_category`

Описывает возможности графа по вставке параллельных ребер (ребер с одинаковыми начальными и конечными вершинами). `Graph*` из SGB не препятствует добавлению параллельных ребер, поэтому этот тип имеет `allow_parallel_edge_tag`.

(Требуется для Graph.)

- `graph_traits<Graph*>::traversal_category`

`Graph*` из SGB обеспечивает обход множества вершин, исходящих ребер и смежных вершин. Таким образом, тег категории обхода определен следующим образом:

```
struct sgb_traversal_tag :
    public virtual vertex_list_graph_tag,
    public virtual incidence_graph_tag,
    public virtual adjacency_graph_tag { };
```

(Требуется для Graph.)

- `graph_traits<Graph*>::vertices_size_type`

Тип используется для работы с количеством вершин в графе.

(Требуется для VertexListGraph.)

- `graph_traits<Graph*>::edges_size_type`

Тип используется для работы с количеством ребер в графе.

(Требуется для EdgeListGraph.)

- `graph_traits<Graph*>::degree_size_type`

Тип используется для работы с количеством исходящих ребер вершины.

(Требуется для IncidenceGraph.)

- `property_map<Graph*, PropertyTag>::type`
  `property_map<Graph*, PropertyTag>::const_type`

Тип отображения для свойств вершин и ребер графа. Свойство задается шаблонным аргументом `PropertyTag` и должно быть одним из тегов, описанных ниже в разделе «Свойства вершин и ребер».

(Требуется для PropertyGraph.)

Функции — методы

Указатель на SGB `Graph` функций — методов класс не имеет.

Функции — не методы

Ниже приведены функции — не методы указателя на SGB `Graph`.

- `std::pair<vertex_iterator, vertex_iterator>`
  `vertices(const Graph* g)`

Возвращает пару итераторов, обеспечивающих доступ к набору вершин графа `g`.

(Требуется для VertexListGraph.)

- `std::pair<edge_iterator, edge_iterator>`
  `edges(const Graph* g)`

Возвращает пару итераторов, обеспечивающих доступ к набору ребер `g`.

(Требуется для EdgeListGraph.)

- `std::pair<adjacency_iterator, adjacency_iterator>`
  `adjacent_vertices(vertex_descriptor v, const Graph* g)`

Возвращает пару итераторов, обеспечивающих доступ к вершинам, смежным с вершиной `v` в графе `g`.

(Требуется для AdjacencyGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `out_edges(vertex_descriptor v, const Graph* g)`

Возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам вершины `v` в графе `g`. Если граф неориентированный, эти значения дают доступ ко всем ребрам, инцидентным данной вершине `v`.

(Требуется для IncidenceGraph.)

- `vertex_descriptor source(edge_descriptor e,`
  `const Graph* g)`

Возвращает начальную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,`
  `const Graph* g)`

Возвращает конечную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,`
  `const Graph* g)`

Возвращает число ребер, исходящих из вершины `u`.

(Требуется для IncidenceGraph.)

- `vertices_size_type num_vertices(const Graph* g)`

Возвращает число вершин в графе `g`.

(Требуется для VertexListGraph.)

- `edges_size_type num_edges(const Graph* g)`

Возвращает число ребер в графе `g`.

(Требуется для EdgeListGraph.)

- `vertex_descriptor vertex(vertices_size_type n, const Graph* g)`

Возвращает *n*-ю вершину в списке вершин графа.

- ```
  template <typename PropertyTag>
    property_map<Graph*, PropertyTag>::type
    get(PropertyTag, Graph* g)
  template <typename PropertyTag>
    property_map<Graph*, PropertyTag>::const_type
    get(PropertyTag, const Graph* g)
  ```

Возвращает объект-отображение свойств, заданный `PropertyTag`.

(Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X>
  typename property_traits<
    typename property_map<Graph*, PropertyTag>::const_type
  ```

```
>::value_type
get(PropertyTag, const Graph* g, X x)
```

Возвращает значение свойства для x, где x — дескриптор вершины или ребра. (Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X,
    typename Value>
  void put(PropertyTag, const Graph* g, X x,
    const Value& value)
  ```

Устанавливает значение свойства для x в value. Здесь x — дескриптор вершины или ребра. Значение должно быть преобразуемо в тип значения свойства, соответствующий PropertyTag.

(Требуется для PropertyGraph.)

Свойства вершин и ребер

Структуры Vertex и Arc из SGB предоставляют вспомогательные поля для хранения дополнительной информации. Мы предлагаем BGL-оболочки, которые обеспечивают доступ к этим полям через отображения свойств. Кроме того, предоставлены отображения индекса вершины и длины ребра. Объект-отображение свойств может быть получен из SGB Graph* применением функции get(), описанной в предыдущем разделе, а тип отображения свойства — через класс свойств property_map.

Указанные ниже теги свойств могут быть использованы для задания вспомогательного поля, для которого требуется отображение свойства (листинг 14.19).

Листинг 14.19. Теги свойств для SGB Graph*

```
//  Теги свойств вершин:
template <typename T> u_property;
emplate <typename T> v_property;
template <typename T> w_property;
template <typename T> x_property;
template <typename T> y_property;
template <typename T> z_property;

//  Теги свойств ребер:
emplate <typename T> a_property;
template <typename T> b_property;
```

Шаблонный параметр T для этих тегов ограничен типами в объединении util, декларированном в заголовочном файле gb_graph.h библиотеки SGB. Перечислим эти типы: Vertex*, Arc*, Graph*, char* и long. Отображения свойств для вспомогательных полей являются моделями LvaluePropertyMap.

Отображение свойств для индексов вершин может быть получено с помощью тега vertex_index_t и это отображение свойств моделирует ReadablePropertyMap. Отображение свойств для длин ребер указывается тегом edge_length_t, и это отображение свойств — модель LvaluePropertyMap, чей тип значения — long.

14.3.5. GRAPH<V,E> из библиотеки LEDA

GRAPH<V,E>

Шаблон класса GRAPH из LEDA может быть напрямую использован как BGL-граф благодаря перегруженным функциям, определенным в заголовочном файле boost/graph/leda_graph.hpp.
```

Реализация BGL-интерфейса для класса GRAPH из LEDA обсуждалась в разделе 10.3 при написании адаптеров для графовых классов из сторонних графовых библиотек.

Пример

В листинге 14.20 приведен пример работы с LEDA графом, как если бы это был BGL-граф.

Листинг 14.20. Работа с графом LEDA

```
#include <boost/graph/leda_graph.hpp>
#include <iostream>
#undef string  // макрос из LEDA
int main()
{
  using namespace boost;
  typedef GRAPH < std::string, int >graph_t;
  graph_t g;
  g.new_node("Philoctetes");
  g.new_node("Heracles");
  g.new_node("Alcmena");
  g.new_node("Eurystheus");
  g.new_node("Amphitryon");
  typedef property_map < graph_t, vertex_all_t >::type NodeMap;
  NodeMap node_name_map = get(vertex_all, g);
  graph_traits < graph_t >::vertex_iterator vi, vi_end;
  for (tie(vi, vi_end) = vertices(g); vi != vi_end; ++vi)
    std::cout << node_name_map[*vi] << std::endl;
  return EXIT_SUCCESS;
}
```

Эта программа выводит следующее:

```
Philoctetes
Heracles
Alcmena
Eurystheus
Amphitryon
```

Параметры шаблона

Ниже приведены параметры шаблона класса GRAPH.

V	Тип объекта, прикрепленного к каждой вершине в графе LEDA.
E	Тип объекта, прикрепленного к каждому ребру в графе LEDA.

Модель для

VertexListGraph, BidirectionalGraph и AdjacencyGraph. Также VertexMutablePropertyGraph и EdgeMutablePropertyGraph для тегов свойств vertex_all_t и edge_all_t, которые обеспечивают доступ к объектам V и E в графе LEDA. Тип GRAPH также является PropertyGraph для vertex_index_t и edge_index_t, который предоставляет доступ к идентификационным (ID) номерам, которые LEDA присваивает каждому узлу (вершине).

Где определен

Класс GRAPH находится в файле boost/graph/leda_graph.hpp.

Ассоциированные типы

Ниже приведены ассоциированные типы класса GRAPH.

- graph_traits<GRAPH>::vertex_descriptor

Тип дескрипторов вершин, ассоциированный с GRAPH. Используется тип node из LEDA.

(Требуется для Graph.)

- graph_traits<GRAPH>::edge_descriptor

Тип дескрипторов ребер, ассоциированный с GRAPH. Используется тип edge из LEDA.

(Требуется для Graph.)

- graph_traits<GRAPH>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices().

(Требуется для VertexListGraph.)

- graph_traits<GRAPH>::out_edge_iterator

Тип итераторов, возвращаемых функцией out_edges().

(Требуется для IncidenceGraph.)

- graph_traits<GRAPH>::in_edge_iterator

Тип итераторов, возвращаемых функцией in_edges().

(Требуется для BidirectionalGraph.)

- graph_traits<GRAPH>::adjacency_iterator

Тип итераторов, возвращаемых функцией adjacent_vertices().

(Требуется для AdjacencyGraph.)

- graph_traits<GRAPH>::directed_category

Тип GRAPH из LEDA — для ориентированных графов, поэтому здесь используется directed_tag.

(Требуется для Graph.)

- graph_traits<GRAPH>::edge_parallel_category

Тип GRAPH из LEDA позволяет добавление параллельных ребер, поэтому здесь используется allow_parallel_edge_tag.

(Требуется для Graph.)

- graph_traits<GRAPH>::traversal_category

Описываемый тип графа представляет итераторы вершин, исходящих и входящих ребер, итераторы смежности. Тип тега категории обхода следующий:

```
struct leda_graph_traversal_category :
      public virtual bidirectional_graph_tag.
```

```
    public virtual adjacency_graph_tag,
    public virtual vertex_list_graph_tag { };
```

(Требуется для Graph.)

- `graph_traits<GRAPH>::vertices_size_type`

Тип используется для представления числа вершин графа, а это `int`.

(Требуется для VertexListGraph.)

- `graph_traits<GRAPH>::edges_size_type`

Тип используется для представления числа ребер графа, а это тоже `int`.

(Требуется для EdgeListGraph.)

- `graph_traits<GRAPH>::degree_size_type`

Тип используется для представления числа исходящих ребер графа — это `int`.

(Требуется для IncidenceGraph.)

- `property_map<GRAPH, PropertyTag>::type`
  `property_map<GRAPH, PropertyTag>::const_type`

Тип отображения для свойств вершин и ребер в графе. Конкретное свойство задается шаблонным аргументом `PropertyTag` и должно быть одним из следующих: `vertex_index_t`, `edge_index_t`, `vertex_all_t` или `edge_all_t`. Теги с «all» используются для доступа к объектам `V` и `E` графа LEDA. Теги `vertex_index_t` и `edge_index_t` обеспечивают доступ к идентификационным номерам, которые LEDA присваивает каждому узлу и ребру.

(Требуется для PropertyGraph.)

Функции — методы

У класса `GRAPH` нет дополнительных функций — методов (так как это потребовало бы модификации исходного кода LEDA)

Функции — не методы

Ниже приведены функции — не методы класса `GRAPH`.

- `std::pair<vertex_iterator, vertex_iterator>`
  `vertices(const GRAPH& g)`

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин графа `g`.

(Требуется для VertexListGraph.)

- `std::pair<edge_iterator, edge_iterator>`
  `edges(const adjacency_matrix& g)`

Возвращает пару итераторов, обеспечивающих доступ к набору ребер графа `g`.

(Требуется для EdgeListGraph.)

- `std::pair<adjacency_iterator, adjacency_iterator>`
  `adjacent_vertices(vertex_descriptor v,`
  `const adjacency_matrix& g)`

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин, смежных с данной вершиной `v` графа `g`.

(Требуется для AdjacencyGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>
  out_edges(vertex_descriptor v, const GRAPH& g)`

Возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам вершины `v` в графе `g`. Если граф неориентированный, эти значения дают доступ ко всем ребрам, инцидентным вершине `v`.

(Требуется для IncidenceGraph.)

- `std::pair<in_edge_iterator, in_edge_iterator>
  in_edges(vertex_descriptor v, const GRAPH& g)`

Возвращает пару итераторов, обеспечивающих доступ к входящим ребрам вершины `v` в графе `g`. Операция недоступна, если в качестве шаблонного параметра `Directed` было указано `directedS`, и доступна при `undirectedS` и `bidirectionalS`.

(Требуется для BidirectionalGraph.)

- `vertex_descriptor source(edge_descriptor e,
  const GRAPH& g)`

Возвращает начальную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,
  const GRAPH& g)`

Возвращает конечную вершину ребра `e`.

(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,
  const GRAPH& g)`

Возвращает число ребер, исходящих из вершины `u`.

(Требуется для IncidenceGraph.)

- `degree_size_type in_degree(vertex_descriptor u,
  const GRAPH& g)`

Возвращает число ребер, входящих в вершину `u`. Эта операция доступна, только если `bidirectionalS` было указано в качестве шаблонного параметра `Directed`.

(Требуется для BidirectionalGraph.)

- `vertices_size_type num_vertices(const GRAPH& g)`

Возвращает число вершин в графе `g`.

(Требуется для VertexListGraph.)

- `edges_size_type num_edges(const GRAPH& g)`

Возвращает число ребер в графе `g`.

(Требуется для EdgeListGraph.)

- `std::pair<edge_descriptor, bool>
  add_edge(vertex_descriptor u, vertex_descriptor v,
  GRAPH& g)`

Добавляет ребро (u, v) к графу и возвращает дескриптор ребра для вновь добавленного ребра. Для этого графового типа логический флаг всегда будет ложным.

(Требуется для EdgeMutableGraph.)

- `std::pair<edge_descriptor, bool>`
  `add_edge(vertex_descriptor u, vertex_descriptor v,`
  `const E& ep, GRAPH& g)`

Добавляет ребро (u, v) к графу и закрепляет за ним `ep` в качестве значения для хранения внутреннего свойства.
(Требуется для EdgeMutablePropertyGraph.)

- `void remove_edge(vertex_descriptor u, vertex_descriptor v,`
  `GRAPH& g)`

Удаляет ребро (u, v) из графа.
(Требуется для EdgeMutableGraph.)

- `void remove_edge(edge_descriptor e, GRAPH& g)`

Удаляет ребро (u, v) из графа. Функция отличается от `remove_edge(u, v, g)` в случае мультиграфа: данная функция удаляет одно ребро, тогда как функция `remove_edge(u, v, g)` удаляет все ребра (u, v).
(Требуется для EdgeMutableGraph.)

- `vertex_descriptor add_vertex(GRAPH& g)`

Добавляет вершину к графу и возвращает дескриптор вершины для новой вершины.
(Требуется для VertexMutableGraph.)

- `vertex_descriptor add_vertex(const VertexProperties& p,`
  `GRAPH& g)`

Добавляет вершину (и ее свойство) к графу и возвращает дескриптор вершины для новой вершины.
(Требуется для VertexMutablePropertyGraph.)

- `void clear_vertex(vertex_descriptor u, GRAPH& g)`

Удаляет все ребра, исходящие и входящие, для данной вершины. Вершина остается во множестве вершин графа.
(Требуется для EdgeMutableGraph.)

- `void remove_vertex(vertex_descriptor u, GRAPH& g)`

Удаляет вершину `u` из множества вершин графа.
(Требуется для VertexMutableGraph.)

- `template <typename PropertyTag>`
  `property_map<GRAPH, PropertyTag>::type`
  `get(PropertyTag, GRAPH& g)`

Возвращает изменяемый объект-отображение свойств для свойства вершины, указанной с помощью `PropertyTag`.
(Требуется для PropertyGraph.)

- `template <typename PropertyTag>`
  `property_map<GRAPH, PropertyTag>::const_type`
  `get(PropertyTag, const GRAPH& g)`

Возвращает константный объект-отображение свойств для свойства вершины, указанной с помощью `PropertyTag`.
(Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X>
  typename property_traits<
    typename property_map<GRAPH, PropertyTag>::const_type
  >::value_type
  get(PropertyTag, const GRAPH& g, X x)
  ```

  Возвращает значение свойства для x, где x — дескриптор вершины или ребра. (Требуется для PropertyGraph.)

- ```
  template <typename PropertyTag, typename X,
    typename Value>
  void put(PropertyTag, const GRAPH& g, X x,
    const Value& value)
  ```

  Устанавливает значение свойства для x в value. Здесь x — дескриптор вершины или ребра.
  (Требуется для PropertyGraph.)

14.3.6. std::vector<EdgeList>

```
std::vector<EdgeList>
```

Перегрузка функций в boost/graph/vector_as_graph.hpp делает возможным трактовать тип вроде std::vector<std::list<int>> как граф.

Пример

В этом примере (листинг 14.21) мы создаем граф, используя контейнерные классы из стандартной библиотеки и применяя функцию print_graph() из BGL (которая написана в терминах графового интерфейса BGL) для вывода графа.

Листинг 14.21. Вектор в качестве графа

```
⟨ vector-as-graph.cpp ⟩ ≡

#include <vector>
#include <list>
#include <boost/graph/vector_as_graph.hpp>
#include <boost/graph/graph_utility.hpp>

int main() {
  enum { r, s, t, u, v, w, x, y, N };
  char name[] = "rstuvwxy";
  typedef std::vector < std::list < int > > Graph;
  Graph g(N);
  g[r].push_back(v);  g[s].push_back(r);  g[s].push_back(r);
  g[s].push_back(w);  g[t].push_back(x);  g[u].push_back(t);
  g[w].push_back(t);  g[w].push_back(x);  g[x].push_back(y);
  g[y].push_back(u);
  boost::print_graph(g, name);
  return 0;
}
```

Эта программа выводит следующее:

```
r --> v
s --> r r w
t --> x
```

```
u --> t
v -->
w --> t x
x --> y
y --> u
```

Где определен

Тип vector_as_graph находится в файле boost/graph/vector_as_graph.hpp.

Параметры шаблона

Ниже приведен параметр шаблона типа vector_as_graph.

EdgeList	Container, в котором value_type позволяет преобразование к size_type для std::vector (чтобы значения можно было использовать в качестве дескриптора вершины).

Модель для

VertexListGraph, IncidenceGraph и AdjacencyGraph.

Ассоциированные типы

Ниже приведены ассоциированные типы vector_as_graph.

- graph_traits<std::vector>::vertex_descriptor

Тип дескрипторов вершин, ассоциированных с графом.
(Требуется для Graph.)

- graph_traits<std::vector>::edge_descriptor

Тип дескрипторов ребер, ассоциированных с графом.
(Требуется для Graph.)

- graph_traits<std::vector>::vertex_iterator

Тип итераторов, возвращаемых функцией vertices().
(Требуется для VertexListGraph.)

- graph_traits<std::vector>::out_edge_iterator

Тип итераторов, возвращаемых функцией out_edges().
(Требуется для IncidenceGraph.)

- graph_traits<std::vector>::adjacency_iterator

Тип итераторов, возвращаемых функцией adjacent_vertices().
(Требуется для AdjacencyGraph.)

- graph_traits<std::vector>::directed_category

Графовый тип для ориентированных графов, поэтому здесь используется directed_tag.
(Требуется для Graph.)

- graph_traits<std::vector>::edge_parallel_category

Этот графовый тип позволяет иметь параллельные ребра, поэтому тип категории — allow_parallel_edge_tag.
(Требуется для Graph.)

- `graph_traits<std::vector>::vertices_size_type`

Тип, используемый для представления количества вершин в графе.
(Требуется для VertexListGraph.)

- `graph_traits<std::vector>::degree_size_type`

Тип, используемый для представления количества исходящих ребер для вершины графа.
(Требуется для IncidenceGraph.)

Функции — методы

Нет дополнительных функций — методов класса.

Функции — не методы

Ниже приведены функции — не методы класса `vector_as_graph`.

- `std::pair<vertex_iterator, vertex_iterator>`
  `vertices(const stid::vector& g)`

Возвращает пару итераторов, обеспечивающих доступ к множеству вершин графа `g`.
(Требуется для VertexListGraph.)

- `std::pair<adjacency_iterator,`
  `adjacency_iterator> adjacent_vertices(vertex_descriptor v,`
  `const std::vector& g)`

Возвращает пару итераторов, обеспечивающих доступ к вершинам, смежным вершине `v` в графе `g`.
(Требуется для AdjacencyGraph.)

- `std::pair<out_edge_iterator, out_edge_iterator>`
  `out_edges(vertex_descriptor v, const std::vector& g)`

Возвращает пару итераторов, обеспечивающих доступ к исходящим ребрам вершины `v` в графе `g`. Если граф неориентированный, эти значения дают доступ ко всем ребрам, инцидентным вершине `v`.
(Требуется для IncidenceGraph.)

- `vertex_descriptor source(edge_descriptor e,`
  `const std::vector& g)`

Возвращает начальную вершину для ребра `e`.
(Требуется для IncidenceGraph.)

- `vertex_descriptor target(edge_descriptor e,`
  `const std::vector& g)`

Возвращает конечную вершину для ребра `e`.
(Требуется для IncidenceGraph.)

- `degree_size_type out_degree(vertex_descriptor u,`
  `const std::vector& g)`

Возвращает число ребер, исходящих из вершины `u`.
(Требуется для IncidenceGraph.)

- `vertices_size_type num_vertices(const std::vector& g)`

Возвращает число вершин в графе `g`.
(Требуется для VertexListGraph.)

Библиотека отображений свойств

15

Большинство алгоритмов на графах требуют доступа к различным свойствам, относящимся к вершинам и ребрам графа. Например, такие данные, как длина или мощность ребра, могут быть необходимы для алгоритмов, равно как и вспомогательные флаги, такие как цвет, для индикации посещения вершины. Существует много способов хранить эти данные в памяти: от полей данных объектов-вершин и объектов-ребер до массивов, индексированных некоторым индексом, или свойств, вычисляемых при необходимости. Для отделения обобщенных алгоритмов от деталей представления свойств вводится абстракция, называемая *отображением свойств*[1].

Несколько категорий свойств доступа предоставляют различные возможности:

readable (только чтение). Свойство может быть только прочитано. Данные возвращаются по значению. Многие отображения свойств для входных данных задачи (таких, как веса ребер) могут определяться как отображения свойств только для чтения.

writable (только запись). Свойство может быть только записано. Массив родителей для записи путей в дереве поиска в ширину является примером отображения свойства, которое может быть определено как только для записи.

read/write (чтение и запись). Свойство может быть как записано, так и прочитано. Свойство расстояния для алгоритма кратчайших путей Дейкстры требует как чтения, так и записи.

lvalue (l-значение). Свойство фактически расположено в памяти и есть возможность получить на него ссылку. Отображения свойств этой категории также поддерживают как чтение, так и запись.

Для каждой категории отображений свойств определена теговая структура.

---

[1] В предыдущих статьях, описывающих BGL, концепция средства доступа была названа Decorator («украшатель»). В магистерской диссертации Дитмара Кюля [24] средства доступа к свойствам названы средствами доступа к данным (data accessors).

```
namespace boost {
  struct readable_property_map_tag { };
  struct writable_property_map_tag { };
  struct read_write_property_map_tag :
    public readable_property_map_tag,
    public writable_property_map_tag { };
  struct lvalue_property_map_tag :
    public read_write_property_map_tag { };
}
```

Подобно классу iterator_traits из STL, есть класс property_traits (листинг 15.1), который может быть использован для выведения ассоциированных типов отображения свойств: типа ключа и типа значения, а также категории отображения свойств. Имеется специализация property_traits, чтобы указатели могли быть использованы как объекты отображений свойств.

Листинг 15.1. Класс property_traits

```
namespace boost {
  template <typename PropertyMap>
  struct property_traits {
    typedef typename PropertyMap::key_type key_type;
    typedef typename PropertyMap::value_type value_type;
    typedef typename PropertyMap::reference reference;
    typedef typename PropertyMap::category category;
  };
  //  специализация для использования указателей как отображений свойств
  template <typename T>
  struct property_traits<T*> {
    typedef T value_type;
    typedef T& reference;
    typedef std::ptrdiff_t key_type;
    typedef lvalue_property_map_tag category;
  };
  template <typename T>
  struct property_traits<const T*> {
    typedef T value_type;
    typedef const T& reference;
    typedef std::ptrdiff_t key_type;
    typedef lvalue_property_map_tag category;
  };
}
```

15.1. Концепции отображений свойств

Интерфейс отображений свойств состоит из набора концепций, которые определяют общий механизм для отображения объектов-ключей на соответствующие объекты-значения, скрывая, таким образом, детали реализации отображения от алгоритмов, использующих отображения свойств. Для обеспечения большей гибкости, требования к отображениям свойств по типу объектов-ключей и значений специально не определены. Поскольку операции отображений свойств — глобальные функции, можно перегрузить функции отображения так, что будут использоваться почти произвольные типы отображений свойств и типов ключей. Интерфейс отображений свойств состоит из трех функций: get(), put() и operator[]. В листинге 15.2

показано, как эти три функции могут быть использованы для доступа к адресам, ассоциированным с разными людьми.

Листинг 15.2. Пример использования интерфейса отображения свойства

```
template <typename AddressMap>
void foo(AddressMap address)
{
  typedef typename boost::property_traits<AddressMap>::
                                              value_type value_type;
  typedef typename boost::property_traits<AddressMap>::key_type key_type;
  value_type old_address, new_address;

  key_type fred = "Fred";
  old_address = get(address, fred);
  new_address = "384 Fitzpatrick Street"
  put(address, fred, new_address);

  key_type joe = "Joe";
  value_type& joes_address = address[joe];
  joes_address = "325 Cushing Avenue";
}
```

Для каждого объекта-отображения свойств имеется набор допустимых ключей, для которых отображение на объекты-значения определено. Вызов функции отображения свойств для недопустимого ключа приводит к неопределенному поведению. Концепции отображений свойств не определяют, как это множество допустимых ключей создается или модифицируется. Функция, которая использует отображение свойств, должна указывать ожидаемый набор допустимых ключей в своих предусловиях.

Обозначения

В следующих разделах использованы такие обозначения:

- `PMap` — тип отображения свойств;
- `pmap` — объект отображения свойств типа `PMap`;
- `key` — объект типа `property_traits<PMap>::key_type`;
- `val` — объект типа `property_traits<PMap>::value_type`.

15.1.1. ReadablePropertyMap

Концепция **ReadablePropertyMap** предоставляет доступ по чтению к объекту-значению, ассоциированному с данным ключом, через вызов функции `get()`. Функция `get()` возвращает копию объекта-значения.

Уточнение для

CopyConstructible

Ассоциированные типы

Ниже приведены ассоциированные типы концепции **ReadablePropertyMap**.

- `property_traits<PMap>::value_type`

  Тип свойства.

- `property_traits<PMap>::reference`

Тип, преобразуемый к типу значения.

- `property_traits<PMap>::key_type`

Тип объекта-ключа, используемого для поиска значения свойства. Отображение свойств может быть параметризовано по типу ключа, в некотором случае этот тип может быть `void`.

- `property_traits<PMap>::category`

Категория свойства: тип, преобразуемый к `readable_property_map_tag`.

Допустимые выражения

Концепция ReadablePropertyMap имеет одно допустимое выражение.

- `get(pmap, key)`

Возвращает тип: `reference`.

Семантика: поиск свойства объекта, ассоциированного с ключом `key`.

15.1.2. WritablePropertyMap

Концепция WritablePropertyMap имеет возможность присваивать объект-значение, ассоциированный с данным объектом-ключом, посредством функции `put()`.

Уточнение для

`CopyConstructible`

Ассоциированные типы

Ниже приведены ассоциированные типы концепции WritablePropertyMap.

- `property_traits<PA>::value_type`

Тип свойства.

- `property_traits<PA>::key_type`

Тип объекта-ключа, используемого для поиска значения свойства. Отображение свойств может быть параметризовано по типу ключа, в некотором случае этот тип может быть `void`.

- `property_traits<PA>::category`

Категория свойства: тип, преобразуемый к `writable_property_map_tag`.

Допустимые выражения

Концепция WritablePropertyMap имеет одно допустимое выражение.

- `put(pmap, key, val)`

Возвращает тип: `void`.

Семантика: присваивает `val` свойству, ассоциированному с ключом `key`.

15.1.3. ReadWritePropertyMap

Концепция ReadWritePropertyMap уточняет концепции ReadablePropertyMap и WritablePropertyMap. ReadWritePropertyMap также добавляет требование, чтобы

property_traits<PA>::category имела тип, преобразуемый к read_write_property_map.

15.1.4. LvaluePropertyMap

Концепция LvaluePropertyMap обеспечивает доступ к ссылке на объект-свойство (вместо копии объекта, как в get()). LvaluePropertyMap может быть изменяемой и неизменяемой. Изменяемая LvaluePropertyMap возвращает ссылку, тогда как неизменяемая — ссылку на константу.

Уточнение для

ReadablePropertyMap для неизменяемой, ReadWritePropertyMap для изменяемой.

Ассоциированные типы

Ниже приведены ассоциированные типы концепции LvaluePropertyMap.

- property_traits<PMap>::reference

  Ссылочный тип, который должен быть ссылкой или константной ссылкой на value_type отображения свойств.

- property_traits<PMap>::category

  Категория свойства: тип, преобразуемый к lvalue_property_map_tag.

Допустимые выражения

Концепция LvaluePropertyMap имеет одно допустимое выражение.

- pmap[key]

  Возвращает тип: reference.

  Семантика: получает ссылку на свойство, идентифицированное ключом.

15.2. Классы отображений свойств

15.2.1. property_traits

```
property_traits<PropertyMap>
```

Класс property_traits предоставляет механизм для доступа к ассоциированным типам отображения свойств. Неспециализированная (использует значения по умолчанию) версия класса property_traits предполагает, что отображение свойства предоставляет определения для всех ассоциированных типов.

```
namespace boost {
  template <typename PA>
  struct property_traits {
    typedef typename PA::key_type key_type;
    typedef typename PA::value_type value_type;
    typedef typename PA::reference reference;
    typedef typename PA::category category;
  };
} // namespace boost
```

Задание типа category должно быть определением для одного из следующих типов или типа, который наследует из одного из следующих типов:

```
namespace boost {
  struct readable_property_map_tag { };
  struct writable_property_map_tag { };
  struct read_write_property_map_tag : readable_property_map_tag,
    writable_property_map_tag { };
  struct lvalue_property_map_tag : read_write_property_map_tag { };
} // namespace boost
```

Часто удобно использовать указатель как объект-отображение свойств, тогда key_type является целочисленным смещением. Такая специализация для property_traits и перегрузка функций отображений приведены в листинге 15.3.

Листинг 15.3. Специализация для property_traits и перегрузка функций

```
namespace boost {
  template <typename T>
  struct property_traits<T*> {
    typedef std::ptrdiff_t key_type;
    typedef T value_type;
    typedef value_type& reference;
    typedef lvalue_property_map_tag category;
  };

  template <typename T>
  void put(T* pa, std::ptrdiff_t k, const T& val) { pa[k] = val; }

  template <typename T>
  const T& get(const T* pa, std::ptrdiff_t k) { return pa[k]; }

  template <typename T>
  T& at(T* pa, std::ptrdiff_t k) { return pa[k]; }
} // namespace boost
```

Параметры шаблона

Класс property_traits имеет один параметр шаблона.

- PropertyMap

Тип отображения свойств.

Где определен

Класс property_traits находится в файле boost/property_map.hpp.

Методы

Ниже приведены методы класса property_traits.

- property_traits::key_type

Тип объекта-ключа, используемого для поиска свойства.

- property_traits::value_type

Тип свойства.

- property_traits::reference

Ссылка на тип значения.

- property_traits::category

Тег категории отображения свойств.

15.2.2. iterator_property_map

```
iterator_property_map<Iterator, IndexMap, T, R>
```

Адаптер iterator_property_map создает оболочку для типа, моделирующего RandomAccessIterator для создания LvaluePropertyMap. Этот адаптер часто полезен для создания отображения свойств из массива, где ключом являются целые числа-смещения от начала массива, а массив содержит объекты-значения. Когда тип ключа — целый, можно просто использовать identity_property_map для параметра шаблона IndexMap. В противном случае придется предоставить отображение свойств, которое преобразует ключ в целое. Например, граф может иметь внутреннее свойство для vertex_index_t, которое может быть получено применением класса property_map.

Пример

Пример в листинге 15.4 демонстрирует создание отображения свойств из массива.

Листинг 15.4. Пример создания отображения свойств

```
#include <iostream>
#include <boost/property_map.hpp>

int main() {
  using namespace boost;
  double x[] = { 0.2, 4.5, 3.2 };
  iterator_property_map < double *, identity_property_map,
                          double, double& > pmap(x);
  std::cout << "x[1] = " << get(pmap, 1) << std::endl;
  put(pmap, 0, 1.7);
  std::cout << "x[0] = " << pmap[0] << std::endl;
  return 0;
}
```

Вывод будет следующий:

```
x[1] = 4.5
x[0] = 1.7
```

Где определен

Адаптер iterator_property_map находится в файле boost/graph/property_map.hpp.

Параметры шаблона

Ниже приведены параметры шаблона адаптера iterator_property_map.

Iterator	Адаптируемый тип итератора. Должен моделировать RandomAccessIterator
IndexMap	Отображение свойств, которое преобразует тип ключа в целое смещение. Должно быть моделью ReadablePropertyMap
T	Тип значения итератора. По умолчанию:typename std::iterator_traits<Iterator>::value_type

R	Тип ссылки итератора. По умолчанию: typename std::iterator_traits<Iterator>::reference

Модель для

LvaluePropertyMap

Ассоциированные типы

Все типы, требуемые для LvaluePropertyMap.

Функции — методы

Ниже приведены функции — методы адаптера iterator_property_map.

- iterator_property_map(Iterator iter = Iterator(), IndexMap index_map = IndexMap())

Конструктор.

- template <typename Key>
- reference operator[ ](Key k) const;

Возвращает *(iter + get(index_map, k)).

Функции — не методы

Ниже приведены функции — не методы адаптера iterator_property_map.

- template <typename Iterator, typename IndexMap>
- iterator_property_map<Iterator, IndexMap,
- typename std::iterator_traits<Iterator>::value_type,
- typename std::iterator_traits<Iterator>::reference>
- make_iterator_property_map(Iterator iter,
- IndexMap index_map)

Создать отображение свойств итератора.

15.2.3. Теги свойств

В листинге 15.5 приведены теги для некоторых свойств BGL.

Листинг 15.5. *Теги свойств*

```
namespace boost {
  enum vertex_index_t { vertex_index = 1};          \\ индекс вершины
  enum edge_index_t { edge_index = 2};              \\ индекс ребра
  enum edge_name_t { edge_name = 3 };               \\ имя ребра
  enum edge_weight_t { edge_weight = 4 };           \\ вес ребра
  enum vertex_name_t { vertex_name = 5 };           \\ имя вершины
  enum graph_name_t { graph_name = 6 };             \\ имя графа
  enum vertex_distance_t { vertex_distance = 7};    \\ расстояние вершины
  enum vertex_color_t { vertex_color = 8 };         \\ цвет вершины
  enum vertex_degree_t { vertex_degree = 9 };       \\ степень вершины
  enum vertex_in_degree_t { vertex_in_degree = 10 };
        \\ степень вершины по входящим ребрам
```

продолжение ⇩

Листинг 15.5 (*продолжение*)

```
    enum vertex_out_degree_t { vertex_out_degree = 11 };
           \\ степень вершины по исходящим ребрам
    enum vertex_discover_time_t { vertex_discover_time = 12 };
           \\ время посещения вершины
    enum vertex_finish_time_t { vertex_finish_time = 13 };
           \\ время окончания обработки вершины
}
namespace boost {
  \\ установить свойство
  BOOST_INSTALL_PROPERTY(vertex, index);
  BOOST_INSTALL_PROPERTY(edge, index);
  BOOST_INSTALL_PROPERTY(edge, name);
  ...
}
```

15.3. Создание пользовательских отображений свойств

Главным назначением интерфейса отображений свойств является повышение гибкости обобщенных алгоритмов. Это позволяет хранить свойства многими возможными способами, в то же время предоставляя алгоритмам общий интерфейс. Следующий раздел содержит пример использования отображений свойств для адаптации к сторонней графовой библиотеке Stanford GraphBase (SGB) (см. раздел 14.3.4). После этого мы рассмотрим реализацию отображения свойств с использованием `std::map`.

15.3.1. Отображения свойств для Stanford GraphBase

Адаптер BGL для Stanford GraphBase включает в себя отображение свойств для доступа к различным полям структур `Vertex` и `Arc` из SGB. В этом разделе мы описываем одну из сторон реализации адаптера SGB в качестве примера реализации отображений свойств.

SGB использует следующую структуру `Vertex` для хранения информации о вершинах в графе. Указатель `arcs` — связный список исходящих ребер вершины. Поле `name` и второстепенные поля от `u` до `z` — свойства вершины (`util` — это объединение C++ (union), позволяющее хранить различные сущности в вершине). Этот раздел описывает, как создать отображение свойств для доступа к полю `name`.

```
typedef struct vertex_struct {
  struct arc_struct* arcs;
  char* name;
  util u, v, w, x, y, z;
} Vertex;
```

Основная идея реализации этого отображения свойств — определение функций `operator[]()`, `get()` и `put()` в терминах доступа к методам структуры данных. Эту работу проще сделать с помощью класса `put_get_helper`, в котором реализованы `put()` и `get()` посредством `operator[]`. Следовательно, остается реализовать только операцию `operator[]`. Кроме того, ассоциированные типы, требуемые отображением свойств, также должны быть определены.

В листинге 15.6 приведена реализация sgb_vertex_name_map. Мы используем класс put_get_helper (он определен в boost/property_map.hpp) для упрощения создания этого отображения свойств. Мы реализуем operator[](), а put_get_helper реализует put() и get(). Первый аргумент-тип в шаблоне класса put_get_helper — тип возвращаемого значения для operator[], который в нашем случае есть char*. Второй аргумент — тип самого отображения свойств. Тип reference должен действительно быть ссылкой, только если отображение свойств должно быть LvaluePropertyMap. В нашем случае мы создаем ReadablePropertyMap. Адаптер SGB использует Vertex* для дескриптора вершины vertex_descriptor графа, так что это key_type отображения свойств.

Листинг 15.6. Реализация sgb_vertex_name_map

```
class sgb_vertex_name_map
  : public put_get_helper< char*, sgb_vertex_name_map > {
public:
  typedef boost::readable_property_map_tag category;
  typedef char* value_type;
  typedef char* reference;
  typedef Vertex* key_type;
  reference operator[ ](Vertex* v) const { return v->name; }
};
```

15.3.2. Отображение свойств из std::map

В предыдущем примере объект-отображение свойств не нуждался в сохранении какого-либо состояния, так как объект-значение мог быть получен непосредственно по ключу. Так бывает не всегда. Обычно ключ используется для поиска объекта-значения в некоторой вспомогательной структуре данных. Очевидным кандидатом на такую структуру является std::map. Отображение свойств, которое использует std::map в качестве реализации, нуждается в хранении указателя на этот ассоциативный контейнер. Мы сделали тип контейнера параметром шаблона, так что отображение свойств может быть использовано с другими контейнерами, такими как hash_map. Концепция, описывающая такой вид контейнера, называется UniquePairAssociativeContainer (листинг 15.7).

Листинг 15.7. Реализация associative_property_map

```
template <typename UniquePairAssociativeContainer>
class associative_property_map
  : public put_get_helper<
    typename UniquePairAssociativeContainer::value_type::second_type&,
    associative_property_map<UniquePairAssociativeContainer> >
{
  typedef UniquePairAssociativeContainer C;
public:
  typedef typename C::key_type key_type;
  typedef typename C::value_type::second_type value_type;
  typedef value_type& reference;
  typedef lvalue_property_map_tag category;
  associative_property_map() : m_c(0) { }
  associative_property_map(C& c) : m_c(&c) { }
  reference operator[ ](const key_type& k) const {
    return (*m_c)[k];
  }
private:
  C* m_c;
};
```

Вспомогательные концепции, классы и функции 16

16.1. Buffer

Концепция Buffer (буфер) — это нечто, во что можно поместить некоторые элементы, а затем удалить. Концепция Buffer имеет очень мало требований. Она не требует какого-то определенного порядка хранения элементов или порядка, в котором они удаляются. Однако обычно имеется некоторое правило упорядочения.

Обозначения

В данной главе будут использованы следующие обозначения:

- `B` — тип, моделирующий Buffer;
- `T` — тип значения элементов `B`;
- `t` — объект типа `T`.

Требования

Чтобы тип моделировал Buffer, он должен иметь следующие методы класса.

- `B::value_type`

  Тип объекта, хранимого в буфере. Тип значения должен быть Assignable.

- `B::size_type`

  Беззнаковый целый тип для представления числа объектов в буфере.

- `b.push(t)`

  Вставляет `t` в буфер. При этом `b.size()` увеличивается на единицу.

- `b.pop()`

  Удаляет объект из Buffer. Это тот самый объект, который возвращает `b.top()`. При этом `b.size()` уменьшается на единицу.
  Предусловие: `b.empty()` — ложь.

- `b.top()`

Возвращает ссылку (или константную ссылку) на некоторый объект в буфере. Предусловие: `b.empty()` — ложь.

- `b.size()`

Возвращает число объектов в буфере.

Инвариант: `b.size() >= 0`.

- `b.empty()`

Тип возвращаемого значения — `bool`. Результат эквивалентен `b.size() == 0`.

Гарантии сложности

Для концепции `Buffer` имеются следующие гарантии сложности.

- `push()`, `pop()` и `size()` должны выполняться не более чем за линейное время от размера буфера.
- `top()` и `empty()` должны выполняться за амортизированное постоянное время.

Модели

`std::stack`, `boost::mutable_queue`, `boost::priority_queue` и `boost::queue`.

16.2. ColorValue

Концепция ColorValue описывает требования к типу, используемому для значений цветов. Многие алгоритмы BGL применяют отображение свойства «цвет» для отслеживания продвижения алгоритма по графу. Тип значений цвета должен быть EqualityComparable. Класс `color_traits` специализирован для `T` так, чтобы были определены следующие функции. Здесь `T` — тип, который моделирует ColorValue.

- `color_traits<T>::white()`

Возвращает тип: `T`.

Семантика: возвращает объект, который представляет белый цвет.

- `color_traits<T>::gray()`

Возвращает тип: `T`.

Семантика: возвращает объект, который представляет серый цвет.

- `color_traits<T>::black()`

Возвращает тип: `T`.

Семантика: возвращает объект, который представляет черный цвет.

16.3. MultiPassInputIterator

Концепция MultiPassInputIterator многопроходного итератора является уточнением InputIterator. Она добавляет требования, чтобы итератор мог быть использован для неоднократного перебора всех значений, и если `it1 == it2` и `it1` можно разыменовать, то `*++it1 == *++it2`.

Итератор MultiPassInputIterator подобен ForwardIterator. Разница состоит в том, что ForwardIterator требует, чтобы тип reference был value_type&, тогда как MultiPassInputIterator похож на InputIterator в том, что тип reference должен быть преобразуем к типу value_type.

16.4. Monoid

Концепция Monoid описывает простой вид алгебраической системы. Она состоит из множества элементов *S*, бинарной операции и нейтрального элемента (identity element). В C++ моноид представляет собой объект-функцию, реализующий бинарную операцию, множество объектов, представляющих элементы *S*, и объект — нейтральный элемент.

Уточнение для

Тип элемента множества *S* должен быть моделью Assignable и CopyConstructible. Тип функционального объекта должен моделировать BinaryFunction.

Правильные выражения

Тип X — это тип элемента. Объекты a, b и c — объекты типа X, представляющие элементы множества S. Объект i — объект типа X, обладающий свойствами нейтрального элемента (приведены ниже). Объект op — это функциональный объект, который реализует операцию на моноиде.

- op(a, b)

Возвращает тип: X.
Семантика: см. ниже.

- a == b

Возвращает тип: bool.
Семантика: возвращает истину, если a и b представляют один и тот же элемент *S*.

- a != b

Возвращает тип: bool.
Семантика: возвращает истину, если a и b представляют разные элементы *S*.

Инварианты

- Замкнутость.

Результат op(a, b) также является элементом *S*.

- Ассоциативность.

```
op(op(a, b), c) == op(a, op(b, c))
```

- Определение нейтрального элемента.

```
op(a, i) == a
```

16.5. mutable_queue

```
mutable_queue<IndexedType, Container, Compare, ID>
```

Адаптер mutable_queue представляет собой специальный вид очереди по приоритетам (реализована с использованием кучи), которая имеет операцию обнов-

ления (update). Это позволяет изменять порядок элементов в очереди. После того как критерий упорядочения для объекта элемента x меняется, необходимо вызывать Q.update(x). Чтобы эффективно находить x в очереди, необходимо написать функтор (объект-функцию) для отображения x в уникальный идентификатор (ID), который mutable_queue затем использует для отображения размещения элемента в куче. Генерируемые идентификаторы должны быть от 0 до N, где N — значение, переданное конструктору mutable_queue.

Параметры шаблона

Ниже приведены параметры шаблона адаптера mutable_queue.

- IndexedType

Если ID не поддерживается, то должна быть определена функция index(t) (где t — объект типа IndexedType), которая возвращает некоторый целый тип.

- Container

Модель для RandomAccessContainer. Тип значения контейнера должен быть таким же, как тип IndexedType.
По умолчанию: std::vector<IndexedType>.

- Compare

Модель для BinaryPredicate (бинарный предикат), которая получает объекты типа IndexedType в качестве аргументов.
По умолчанию: std::less<typename Container::value_type>.

- ID

Модель для ReadablePropertyMap, которая принимает IndexedType в качестве типа ключа, а возвращает некоторый целый тип в качестве типа значения.
По умолчанию: identity_property_map.

Методы

Ниже приведены методы адаптера mutable_queue.

- value_type

Тот же тип, что и IndexedType.

- size_type

Тип, используемый для представления размера очереди.

- mutable_queue(size_type n, const Compare& c,
  const ID& id = ID())

Конструктор. Резервируется место для n элементов.

- template <class InputIterator>
  mutable_queue(InputIterator first, InputIterator last,
  const Compare& c, const ID& id = ID())

Конструктор. Контейнер std::vector по умолчанию заполняется из диапазона [first, last).

- bool empty() const

Возвращает истину, если очередь пуста.

- `void pop()`

Удаляет объект из очереди.

- `value_type& top()`

Возвращает ссылку на первый элемент очереди.

- `value_type& front()`

Другое имя для `top()`.

- `void push(const value_type& x)`

Вставляет копию объекта `x` в очередь.

- `void update(const value_type& x)`

«Значение» элемента изменилось, и должно быть заново произведено упорядочение кучи. Этот метод подразумевает, что имеется старый элемент `y` в куче, для которого `index(y) == index(x)`, и что `x` — новое значение элемента.

16.6. Непересекающиеся множества

16.6.1. disjoint_sets

```
disjoint_sets<RankMap, ParentMap, FindCompress>
```

Класс `disjoint_sets` предоставляет операции на *непересекающихся множествах* (НМ), иногда называемых структурой данных для объединения и поиска (union-find data structure). Структура данных НМ поддерживает набор $S = S_1, S_2, ..., S_k$ непересекающихся множеств. Каждое множество идентифицируется своим *представителем*, взятым из множества. Множества представлены корневыми деревьями, которые закодированы в отображении свойства ParentMap. Для ускорения операций используются две эвристики: *объединение по рангу* и *сжатие путей*.

Параметры шаблона

Ниже приведены параметры шаблона класса `disjoint_sets`.

`RankMap`	Должен быть моделью ReadWritePropertyMap с целым типом значения и типом ключа, совпадающим с типом элемента множества
`ParentMap`	Должен быть моделью ReadWritePropertyMap. Типы ключа и значения должны совпадать с типом элемента множества
`FindCompress`	Должен быть одним из функциональных объектов, которые мы обсудим позднее в этом разделе. По умолчанию: `find_with_full_path_compression`

Пример

Типичный образец использования непересекающихся множеств можно обнаружить в алгоритме `kruskal_minimum_spanning_tree()`. В примере в листинге 16.1 мы

вызываем link() вместо union_set(), так как u и v получаются из find_set(), а значит, уже являются представителями своих множеств.

Листинг 16.1. Использование непересекающихся множеств

```
// ...
  disjoint_sets<RankMap, ParentMap, FindCompress> dsets(rank, p);

  for (ui = vertices(G).first; ui != vertices(G).second; ++ui)
    dsets.make_set(*ui);
  // ...
  while ( !Q.empty() ) {
    e = Q.front();
    Q.pop();
    u = dsets.find_set(source(e));
    v = dsets.find_set(target(e));
    if ( u != v ) {
      *out++ = e;
      dsets.link(u, v);  // связывание
    }
  }
```

Методы

Ниже приведены возможные методы класса disjoint_sets.

- `disjoint_sets(RankMap r, ParentMap p)`

  Конструктор.

- `disjoint_sets(const disjoint_sets& x)`

  Конструктор копирования.

- ```
  template <typename Element>
  void make_set(Element x)
  ```

  Создает множество из одного элемента, содержащего элемент x.

- ```
  template <typename Element>
  void link(Element x, Element y)
  ```

  Объединить два множества, представленные элементами x и y.

- ```
  template <typename Element>
  void union_set(Element x, Element y)
  ```

  Объединить два множества, содержащие элементы x и y. Это эквивалентно link(find_set(x), find_set(y)).

- ```
  template <typename Element>
  Element find_set(Element x)
  ```

  Возвращает представителя множества, содержащего элемент x.

- ```
  template <typename ElementIterator>
  std::size_t count_sets(ElementIterator first,
    ElementIterator last)
  ```

  Возвращает число непересекающихся множеств.

- ```
  template <typename ElementIterator>
  void compress_sets(ElementIterator first,
    ElementIterator last)
  ```

  Выравнивает дерево родителей так, что родитель каждого элемента является его представителем.

Сложность

Временная сложность порядка $O(m\alpha(m, n))$, где α — функция, обратная функции Аккермана, m — число операций на непересекающихся множествах (`make_set()`, `find_set()` и `link()`) и n — число элементов. Функция, обратная функции Аккермана, растет очень медленно, намного медленнее логарифма.

16.6.2. find_with_path_halving

```
find_with_path_halving
```

Этот функтор находит вершину-представителя для компоненты, в которой содержится элемент `x`, и в то же время сжимает дерево, используя деление пути пополам.

```
template <typename ParentMap, typename Element>
Element operator()(ParentMap p, Element x)
```

16.6.3. find_with_full_path_compression

```
find_with_full_path_compression
```

Этот функтор находит вершину-представителя для компоненты, в которой содержится элемент `x`, и в то же время сжимает дерево, используя полное сжатие пути.

```
template <typename ParentMap, typename Element>
Element operator()(ParentMap p, Element x)
```

16.7. tie

```
template <typename T1, typename T2>
tuple<T1, T2> tie(T1& a, T2& b);
```

Эта функция из Boost Tuple Library (библиотека кортежей), написанная Яакко Ярви, делает более удобной работу с функциями, которые возвращают пары (или, в общем случае, кортежи). Функция `tie()` позволяет присваивать два значения пары двум различным переменным.

Где определена

Функция `tie()` находится в файле **boost/tuple/tuple.hpp**.

Пример

В примере функция `tie()` используется вместе с функцией `vertices()`, которая возвращает пару типа `std::pair<vertex_iterator, vertex_iterator>`. Пара итераторов присваивается переменным `i` и `end`.

```
graph_traits<graph t>::vertex_iterator i, end;
for(tie(i, end) = vertices(g); i != end; ++i)
  // ...
```

В листинге 16.2 приведен другой пример, который использует `tie()` для работы с `std::set`.

Листинг 16.2. Пример использования функции tie().

```
#include <set>
#include <algorithm>
#include <iostream>
#include <boost/tuple/tuple.hpp>

int main() {
  typedef std::set<int> SetT;
  SetT::iterator i, end;
  bool inserted;

  int vals[5] = { 5, 2, 4, 9, 1 };
  SetT s(vals, vals + 5);
  int new_vals[2] = { 3, 9 };

  for (int k = 0; k < 2; ++k) {
    // Используем tie() со значением типа pair<iterator, bool>
    boost::tie(i,inserted) = s.insert(new_vals[k]);
    if (!inserted)
      std::cout << *i << " уже находилось во множестве."
                << std::endl;
    else std::cout << *i << " успешно вставлено." << std::endl;
  }

  return EXIT_SUCCESS;
}
```

Программа выводит следующее:

```
3 успешно вставлено.
9 уже находилось во множестве.
```

16.8. graph_property_iter_range

```
graph_property_iter_range<Graph, PropertyTag>
```

Этот класс генерирует пару итераторов begin/end, которые предоставляют доступ к свойству вершины для всех вершин в графе или к свойству ребра для всех ребер в графе.

Пример

В примере (листинг 16.3) перебираются все вершины в графе, с присвоением строк в свойство «имя». Затем снова перебираются все вершины и выводятся имена в стандартный вывод.

Листинг 16.3. Файл graph-property-iter-eg.cpp

```
⟨ graph-property-iter-eg.cpp ⟩ ≡
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/property_iter_range.hpp>

int
main()
```

продолжение ⇩

Листинг 16.3 (*продолжение*)

```
{
  using namespace boost;
  typedef adjacency_list < listS, vecS, directedS,
    property < vertex_name_t, std::string > >graph_t;
  graph_t g(3);

  const char *vertex_names[] = { "Kubrick", "Clark", "Hal" };
  int i = 0;
  graph_property_iter_range < graph_t, vertex_name_t >::iterator v, v_end;
  for (tie(v, v_end) = get_property_iter_range(g, vertex_name);
       v != v_end; ++v, ++i)
    *v = vertex_names[i];

  tie(v, v_end) = get_property_iter_range(g, vertex_name);
  std::copy(v, v_end, std::ostream_iterator < std::string >
            (std::cout, " "));
  std::cout << std::endl;
  return 0;
}
```

Программа выводит следующее:

```
Kubrick Clark Hal
```

Где определена

Класс `graph_property_iter_range` находится в файле **boost/graph/property_iter_range.hpp**.

Параметры шаблона

Ниже приведены параметры шаблона класса `graph_property_iter_range`.

`Graph`	Графовый тип должен быть моделью **PropertyGraph**.
`PropertyTag`	Тег указывает, к какому свойству ребра или вершины происходит доступ.

Ассоциированные типы

Ниже приведены ассоциированные типы класса `graph_property_iter_range`.

- `graph_property_iter_range::iterator`

Изменяемый итератор, тип значения которого — свойство, указанное тегом свойства.

- `graph_property_iter_range::const_iterator`

Константный итератор, тип значения которого — свойство, указанное тегом свойства.

- `graph_property_iter_range::type`

Тип `std::pair<iterator, iterator>`.

- `graph_property_iter_range::const_type`

Тип `std::pair<const_iterator, const_iterator>`.

Функции — методы

Не имеет.

Функции — не методы

Ниже приведены функции — не методы класса graph_property_iter_range.

- ```
  template<typename Graph, typename Tag>
  typename graph_property_iter_range<Graph, Tag>::type
  get_property_iter_range(Graph& graph, const Tag& tag)
  ```

  Возвращает пару изменяемых итераторов, которые дают доступ к свойству, указанному тегом. Итераторы пробегают по всем вершинам или по всем ребрам графа.

- ```
  template<typename Graph, typename Tag>
  typename graph_property_iter_range<Graph, Tag>::const_type
  get_property_iter_range(const Graph& graph,
    const Tag& tag)
  ```

  Возвращает пару константных итераторов, которые дают доступ к свойству, указанному тегом. Итераторы пробегают по всем вершинам или по всем ребрам графа.

Библиография

1. Dimacs implementation file format. http://dimacs.rutgers.edu/Challenges/.
2. A. Alexandrescu. Better template error messages. C/C++ Users Journal, March 1999.
3. M. H. Austern. Generic Programming and the STL. Professional computing series. Addison-Wesley, 1999.
4. G. Baumgartner and V. F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. Software–Practice and Experience, 25(8):863–889, August 1995.
5. R. Bellman. On a routing problem. Quarterly of Applied Mathematics, 16(1): 87–90, 1958.
6. Boost. Boost C++ Libraries. http://www.boost.org/.
7. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. In 9th World Wide Web Conference, 2000.
8. K. B. Bruce, L. Cardelli, G. Castagna, the Hopkins Objects Group, G. T. Leavens, and B. Pierce. On binary methods. Theory and Practice of Object Systems, 1:221–242, 1995.
9. B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical report, Stanford University, 1994.
10. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. McGraw-Hill, 1990.
11. E. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269–271, 1959.
12. L. R. Ford and D. R. Fulkerson. Maximal flow through a network. Canadian Journal of Mathematics, pages 399–404, 1956.
13. L. R. Ford and D. R. Fulkerson. Flows in Networks. Princeton University Press, 1962.

14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing. Addison-Welsey, 1995.

15. M. Garey and D. Johnson. Computers and Intractibility: A Guide to the Theory of NPCompleteness. W.H. Freeman, New York, 1979.

16. A. V. Goldberg. A new max-flow algorithm. Technical Report MIT/LCS/TM-291, MIT, 1985.

17. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. Journal of the ACM, 1988.

18. R. Graham and P. Hell. On the history of the minimum spanning tree problem. Annals of the History of Computing, 7(1):43–57, 1985.

19. C. Hedrick. Routing information protocol. Internet Requests For Comments (RFC) 1058, June 1988.

20. A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. Sov. Math. Dokl., 1974.

21. S. E. Keene. Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS. Addison-Wesley, 1989.

22. D. E. Knuth. Stanford GraphBase: A Platform for Combinatorial Computing. ACM Press, 1994.

23. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In Proceedings of the American Mathematical Sofiety, volume 7, pages 48–50, 1956.

24. D. Kühl. Design patterns for the implementation of graph algorithms. Master's thesis, Technische Universität Berlin, July 1996.

25. J. Lajoie and S. B. Lippman. C++ Primer. Addison Wesley, 3rd edition, 1998.

26. E. L. Lawler. Combinatorial Opimization: Networks and Matroids. Holt, Rinehart, and Winston, 1976.

27. D. Matula. Determining edge connectivity in o(mn). In Sumposium on Foundations of Computer Science, pages 249–251, 1987.

28. J. McQuillan. The new routing algorithm for the arpanet. IEEE Transactions on Communications, May 1980.

29. K. Mehlhorn and S. Näher. The LEDA Platform of Combinatorial and Geometric Computing. Cambridge University Press, 1999.

30. B. Meyer. Object-oriented Software Construction. Prentice Hall International Series in Computer Science. Prentice Hall, 1988.

31. E. Moore. The shortest path through a maze. In International Symposium on the Theory of Switching, pages 285–292. Harvard University Press, 1959.

32. R. Morgan. Building an Optimizing Compiler. Butterworth-Heinemann, 1998.

33. J. Moy. Rfc 1583: Ospf version 2. Network Working Group Request for Comment, March 1994.

34. D. R. Musser, G. J. Derge, and A. Saini. STL Tutorial and Reference Guide. Addison-Wesley, 2nd edition, 2001.

35. D. R. Musser and A. A. Stepanov. A library of generic algorithms in ada. In Using Ada (1987 International Ada Conference), pages 216–225, New York, NY, Dec. 1987. ACM SIGAda.

36. N. C. Myers. Traits: a new and useful template technique. C++ Report, June 1995.

37. R. Perlman. Fault-tolerant broadcast of routing information. Computer Networks, December 1983.

38. R. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.

39. J. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In First Workshop on C++ Template Programming, Erfurt, Germany, October 10 2000.

40. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

41. B. Stroustrup. Design and Evolution of C++. Addison-Wesley, 1994.

42. B. Stroustrup. The C++ Programming Language. Addison Wesley, special edition, 2000.

43. R. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2):146–160, 1972.

44. R. E. Tarjan. Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics, 1983.

45. B. L. van der Waerden. Algebra. Frederick Ungar Publishing, 1970.

46. H. C. Warnsdorff. Des roesselsprungs einfachste und allgemeinste loesung. Schmalkalden, 1823.

Дополнение к библиографии

Теория графов

1. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.
2. *Беллман Р., Дрейфус С.* Прикладные задачи динамического программирования. — М.: Наука, 1965.
3. *Евстигнеев В. А., Касьянов В. Н.* Теория графов: Алгоритмы обработки деревьев. — Новосибирск: Наука, 1994.
4. *Касьянов В. Н., Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. — СПб.: БХВ-Петербург, 2003.
5. *Кристофидес Н.* Теория графов. Алгоритмический подход. — М.: Мир, 1978.
6. *Липский В.* Комбинаторика для программистов. — М.: Мир, 1988.
7. *Оре О.* Теория графов. — М.: Наука, 1968.
8. *Форд Л. Р., Фалкерсон Д. Р.* Потоки в сетях. — М.: Мир, 1966.
9. *Харари Ф.* Теория графов. — М.: Мир, 1973.

C++ и STL

1. *Александреску А.* Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования. — Киев: Издательский дом «Вильямс», 2002.
2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. — М.: Издательский дом «БИНОМ», 2001.
3. *Вандевурд Д., Джосаттис Н.* Шаблоны C++: Справочник разработчика. — Киев: Издательский дом «Вильямс», 2003.

4. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования: Паттерны проектирования. — СПб.: Питер, 2001.
5. *Джосьютис Н.* C++ стандартная библиотека. Для профессионалов. — СПб.: Питер, 2003.
6. *Мейерс С.* Эффективное использование STL: Библиотека программиста. — СПб: Питер, 2002
7. *Остерн М. Г.* Обобщенное программированиеи и STL: использование и наращивание стандартной библиотеки C++. — СПб.: Невский Диалект, 2004.
8. *Плаугер П., Ли М., Массер Д., Степанов А.* STL — стандартная библиотека шаблонов C+. — СПб.: БХВ-Петербург, 2004.
9. *Саттер Г.* Решение сложных задач на C++. Серия C++ In-Depth. — Киев: Издательский дом «Вильямс», 2003.
10. *Страуструп Б.* Язык программирования C++. — М.: Издательский дом «БИНОМ», 2001.
11. *Страуструп Б.* Дизайн и эволюция языка C++. — М.: ДМК, 2000.

Алфавитный указатель

Ф

Ц

Ч