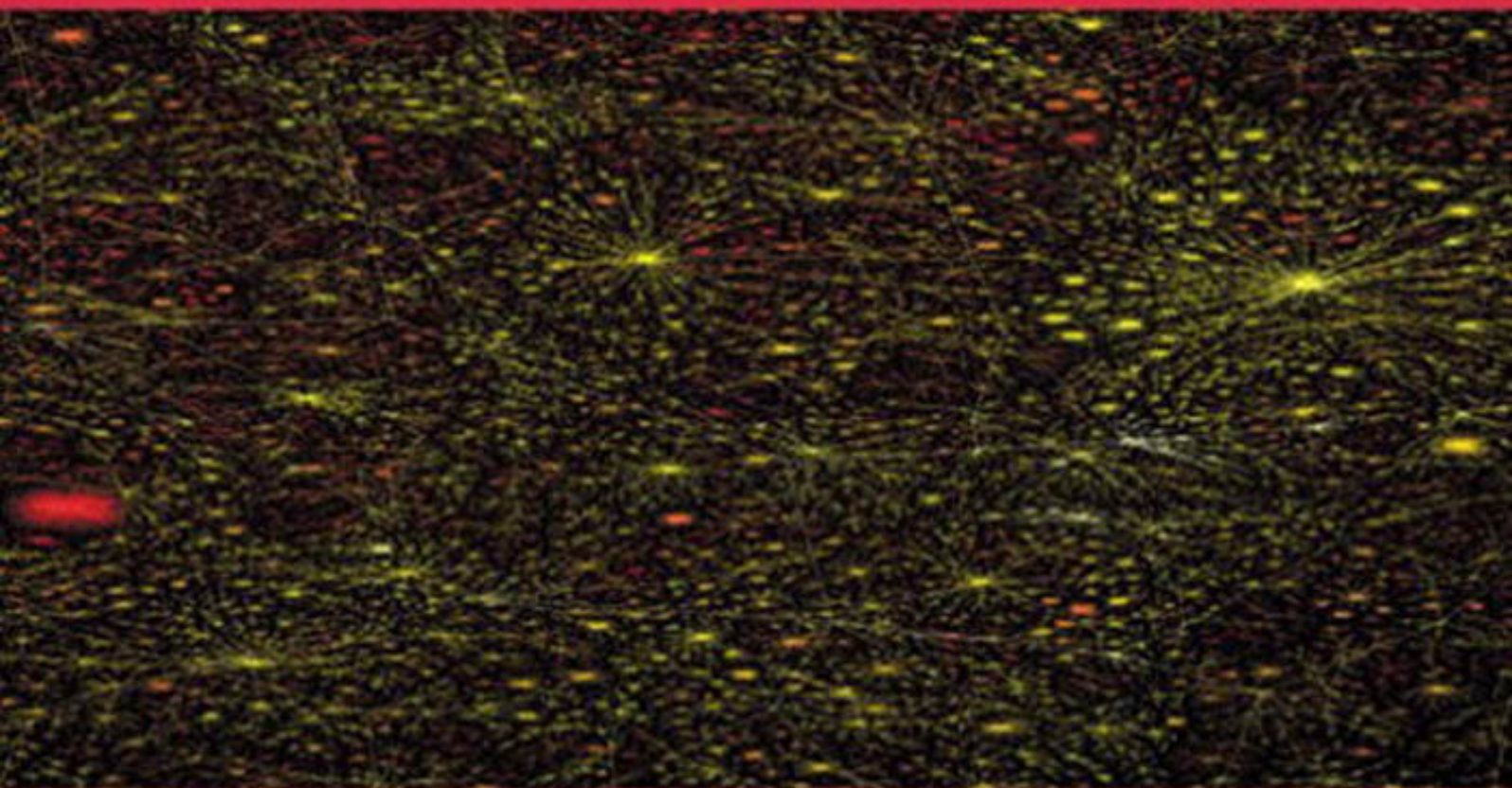
The Boost Graph Library

User Guide and Reference Manual

Jeremy G. Siek
Lie-Quan Lee
Andrew Lumsdaine

Foreword by Alexander Stepanov



C++ In-Depth Series • Bjarne Stroustrup

Library of Congress Cataloging-in-Publication Data

Siek, Jeremy G.
The Boost graph library : user guide and reference manual/ Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine
p. cm.
Includes bibliographical references and index.
ISBN 0-201-72914-8 (alk. paper)
1. C++ (Computer language). 2. Graph theory. I. Lee, Lie-Quan. II. Lumsdaine, Andrew. III. Title.

006.6—dc21

2001053553

ISBN 0-201-72914-8
Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10—MA—0504030201
First printing, December 2001

Contents

Foreword

When I first looked at this book, I felt envious. After all, what led me to the discovery of generic programming was the desire to build a library like the Boost Graph Library (BGL). In 1984 I joined the faculty of Polytechnic University in Brooklyn with some vague ideas about building libraries of software components. Well, to tell you the truth that was my secondary interest—my real interest at that time was to construct formal underpinnings of natural language, something like Aristotle's Organon, but more complete and formal. I was probably the only assistant professor in any Electrical Engineering or Computer Science department who meant to obtain tenure through careful study of Aristotle's Categories. Interestingly enough, the design of the Standard Template Library (STL)—in particular the underlying ontology of objects—is based on my realization that the whole-part relation is a fundamental relation that describes the real world and that it is not at all similar to the element-set relation familiar to us from set theory. Real objects do not share parts: my leg is nobody else's leg. STL containers are like that: two containers do not share parts. There are operations like ***std::list::splice*** that move parts from one container to another; they are similar to organ transplant: my kidney is mine until it is spliced into somebody else.

In any case, I was firmly convinced that software components should be functional in nature and based on John Backus's FP system. The only novel intuition was that functions should be associated with some axioms: for example, the "Russian peasant algorithm" that allows one to compute the nth power in $O(\log n)$ steps is defined for any object that has an associative binary operation defined on it. In other words, I believed that algorithms should be associated with what we now call concepts (see §2.3 of this book), but what I called structure types and what type-theorists call *multi-sorted algebras*.

It was my great luck that Polytechnic had a remarkable person on its faculty, Aaron Kershenbaum, who combined deep knowledge of graph algorithms with an unusual desire to implement them. Aaron saw potential in my attempts to decompose programs into simple primitives, and spent a lot of time teaching me graph algorithms and working with me on implementing them. He also showed me that there were some fundamental things that cannot be done functionally without prohibitive change in the complexity. Although it was often possible for me to implement linear time algorithms functionally without changing the asymptotic complexity, it was impossible in practice to implement logarithmic time algorithms without making them linear. In particular, Aaron explained to me why priority queues were so im-

portant for many graph algorithms (and he was well qualified to do so: Knuth in his Stanford GraphBase book [22] attributes the discovery of how to apply binary heaps to Prim's and Dijkstra's algorithms to Aaron).

It was a moment of great joy when we were able to produce Prim's and Dijkstra's algorithms as two instances of the same generic—we called it "high-order" then—algorithm. It is quite remarkable how close BGL code is to what we had (see, for example, a footnote to §13.4.2). The following code in Scheme shows how the two algorithms were implemented in terms of the same higher-order algorithm. The only difference is in how distance values are combined: using addition for Dijkstra's and by selecting the second operand for Prim's.

```
(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) < ))
```

It took me a long time—almost 10 years—to find a language in which this style of programming could be *effectively* realized. I finally found C++, which enabled me to produce something that people could use. Moreover, C++ greatly influenced my design by providing a crisp C-based machine model. The features of C++ that enabled STL are templates and overloading.

I often hear people attacking C++ overloading, and, as is true with most good mechanisms, overloading can be misused. But it is an essential mechanism for the development of useful abstractions. If we look at mathematics, it has been greatly driven by overloading. Extensions of a notion of numbers from natural numbers to integers, to rational numbers, to Gaussian integers, to p-adic numbers, etc, are examples of overloading. One can easily guess things without knowing exact definitions. If I see an expression that uses both addition and multiplication, I assume distributivity. If I see less-than and addition, I assume that if $a < b$ then $a + c < b + c$ (I seldom add uncountable cardinals). Overloading allows us to carry knowledge from one type to another.

It is important to understand that one can write generic algorithms just with overloading, without templates: it does, however, require a lot of typing. That is, for every class that satisfies, say, random access iterator requirements, one has to define all the relevant algorithms by hand. It is tedious, but can be done (only signatures would need to be defined: the bodies will be the same). It should be noted that generics in Ada require hand-instantiation and, therefore, are not that helpful, since every algorithm needs to be instantiated by hand. Templates in C++ solve this problem by allowing one to define things once.

There are still things that are needed for generic programming that are not yet representable in C++. Generic algorithms are algorithms that work on objects with similar interfaces. Not identical interfaces as in object-oriented programming, but similar. It is not just the handling of binary methods (see §2.1.3) that causes the problem, it is the fact that inter-

faces are described in terms of a single type (single-sorted algebra). If we look carefully at things like iterators we observe that they are describable only in terms of multiple types: the iterator type itself, the value type, and the distance type. In other words, we need three types to define the interfaces on one type. And there is no machinery in C++ to do that. The result of this is that we cannot define what iterators are and, therefore, cannot really compile generic algorithms. For example, if we define the reduce algorithm as:

```
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;
}
```

but instead of: *++first != last* we write: *++first<last*, no compiler can detect the bug at the point of definition. Though the standard clearly states that *operator<* does not need to be defined for Input Iterators, there is no way for the compiler to know it. Iterator requirements are just words. We are trying to program with concepts (multi-sorted algebras) in a language that has no support for them.

How hard would it be to extend C++ to really enable this style of programming? First, we need to introduce concepts as a new interface facility. For example, we can define:

```
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;
}
```

Generic functions are functions that take concepts as arguments and in addition to an argument list have a list of type constraints. Now full type checking can be done at the point

of definition without looking at the points of call, and full type-checking can be done at the points of call without looking at the body of the algorithm.

Sometimes we need multiple instances of the same concept. For example,

```
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]));
```

Note that this merge is not as powerful as the STL merge. It cannot merge a list of ***float***s and a vector of ***double***s into a deque of ***int***s. STL algorithms will often do unexpected and, in my opinion, undesirable type conversions. If someone needs to merge ***double***s and ***float***s into ***int***s, he or she should use an explicit function object for asymmetric comparison and a special output iterator for conversion.

C++ provides two different abstraction mechanisms: object-orientedness and templates. Object-orientedness allows for exact interface definition and for run-time dispatch. But it cannot handle binary methods or multi-method dispatching, and its run-time binding is often inefficient. Templates handle richer interfaces and are resolved at compile-time. They can, however, cause a software engineering nightmare because of the lack of separation between interfaces and implementation. For example, I recently tried compiling a 10-line STL-based program using one of the most popular C++ compilers, and ran away in shock after getting several pages of incomprehensible error messages. And often one needs run-time dispatch that cannot be handled by templates. I do believe that introduction of concepts will unify both approaches and resolve both sets of limitations. And after all, it is possible to represent concepts as virtual tables that are extended by pointers to type descriptors: the virtual table for input iterator contains not just pointers to ***operator**** and ***operator++***, but also pointers to the actual type of the iterator, its value type, and its distance type. And then one could introduce pointers to concepts and references to concepts!

Generic programming is a relatively young subdiscipline of computer science. I am happy to see that the small effort—started twenty years ago by Dave Musser, Deepak Kapur, Aaron Kershenbaum and me—led to a new generation of libraries such as BGL and MTL. And I have to congratulate Indiana University on acquiring one of the best generic programming teams in the world. I am sure they will do other amazing things!

Alexander Stepanov
Palo Alto, California
September, 2001[1]

[1] I would like to thank John Wilkinson, Mark Manasse, Marc Najork, and Jeremy Siek for many valuable suggestions.

Preface

The graph abstraction is a powerful problem-solving tool used to describe relationships between discrete objects. Many practical problems can be modeled in their essential form by graphs. Such problems appear in many domains: Internet packet routing, telephone network design, software build systems, Web search engines, molecular biology, automated road-trip planning, scientific computing, and so on. The power of the graph abstraction arises from the fact that the solution to a graph-theoretic problem can be used to solve problems in a wide variety of domains. For example, the problem of solving a maze and the problem of finding groups of Web pages that are mutually reachable can both be solved using depth-first search, an important concept from graph theory. By concentrating on the essence of these problems—the graph model describing discrete objects and the relationships between them—graph theoreticians have created solutions to not just a handful of particular problems, but to entire families of problems.

Now a question arises. If graph theory is generally and broadly applicable to arbitrary problem domains, should not the software that implements graph algorithms be just as broadly applicable? Graph theory would seem to be an ideal area for software reuse. However, up until now the potential for reuse has been far from realized. Graph problems do not typically occur in a pure graph-theoretic form, but rather are embedded in larger domain-specific problems. As a result, the data to be modeled as a graph are often not explicitly represented as a graph but are instead encoded in some application-specific data structure. Even in the case where the application data are explicitly represented as a graph, the particular graph representation chosen by the programmer might not match the representation expected by a library that the programmer wants to use. Moreover, different applications may place different time and space requirements on the graph data structure.

This implies a serious problem for the graph library writer who wants to provide reusable software, for it is impossible to anticipate every possible data structure that might be needed and to write a different version of the graph algorithm specifically for each one. The current state of affairs is that graph algorithms are written in terms of whatever data structure is most convenient for the algorithm and users must convert their data structures to that format in order to use the algorithm. This is an inefficient undertaking, consuming programmer time and computational resources. Often, the cost is perceived not to be worthwhile, and the programmer instead chooses to rewrite the algorithm in terms of his or her own data structure.

This approach is also time consuming and error prone, and will tend to lead to sub-optimal solutions since the application programmer may not be a graph algorithms expert.

Generic Programming

The Standard Template Library (STL) [40] was introduced in 1994 and was adopted shortly thereafter into the C++ Standard. The STL was a library of interchangeable components for solving many fundamental problems on sequences of elements. What set the STL apart from libraries that came before it was that each STL algorithm could work with a wide variety of sequential data structures: linked-lists, arrays, sets, and so on. The iterator abstraction provided an interface between containers and algorithms and the C++ template mechanism provided the needed flexibility to allow implementation without loss of efficiency. Each algorithm in the STL is a function template parameterized by the types of iterators upon which it operates. Any iterator that satisfies a minimal set of requirements can be used regardless of the data structure traversed by the iterator. The systematic approach used in the STL to construct abstractions and interchangeable components is called *generic programming*.

Generic programming lends itself well to solving the reusability problem for graph libraries. With generic programming, graph algorithms can be made much more flexible, allowing them to be easily used in a wide variety applications. Each graph algorithm is written not in terms of a specific data structure, but instead to a graph abstraction that can be easily implemented by many different data structures. Writing generic graph algorithms has the additional advantage of being more natural; the abstraction inherent in the pseudo-code description of an algorithm is retained in the generic function.

The Boost Graph Library (BGL) is the first C++ graph library to apply the notions of generic programming to the construction of graph algorithms.

Some BGL History

The Boost Graph Library began its life as the Generic Graph Component Library (GGCL), a software project at the Lab for Scientific Computing (LSC). The LSC, under the direction of Professor Andrew Lumsdaine, was an interdisciplinary laboratory dedicated to research in algorithms, software, tools, and run-time systems for high-performance computational science and engineering.[2] Special emphasis was put on developing industrial-strength, high-performance software using modern programming languages and techniques—most notably, generic programming.

Soon after the Standard Template Library was released, work began at the LSC to apply generic programming to scientific computing. The Matrix Template Library (MTL) was one of the first projects. Many of the lessons learned during construction of the MTL were applied to the design and implementation of the GGCL.

[2]The LSC has since evolved into the Open Systems Laboratory (OSL) ***http://www.osl.iu.edu***. Although the name and location have changed, the research agenda remains the same.

An important class of linear algebra computations in scientific computing is that of sparse matrix computations, an area where graph algorithms play an important role. As the LSC was developing the sparse matrix capabilities of the MTL, the need for high-performance reusable (and generic) graph algorithms became apparent. However, none of the graph libraries available at the time (LEDA, GTL, Stanford GraphBase) were written using the generic programming style of the MTL and the STL, and hence did not fulfill the flexibility and high-performance requirements of the LSC. Other researchers were also expressing interest in a generic C++ graph library. During a meeting with Bjarne Stroustrup, we were introduced to several individuals at AT&T who needed such a library. Other early work in the area of generic graph algorithms included some codes written by Alexander Stepanov, as well as Dietmar Kühl's master's thesis.

With this in mind, and motivated by homework assignments in his algorithms class, Jeremy Siek began prototyping an interface and some graph classes in the spring of 1998. Lie-Quan Lee then developed the first version of the GGCL, which became his master's thesis project.

During the following year, the authors began collaborating with Alexander Stepanov and Matthew Austern. During this time, Stepanov's disjoint-sets-based connected components implementation was added to the GGCL, and work began on providing concept documentation for the GGCL, similar to Austern's STL documentation.

During this year the authors also became aware of Boost and were excited to find an organization interested in creating high-quality, open source C++ libraries. Boost included several people interested in generic graph algorithms, most notably Dietmar Kühl. Some discussions about generic interfaces for graph structures resulted in a revision of the GGCL that closely resembles the current Boost Graph Library interface.

On September 4, 2000, the GGCL passed the Boost formal review (managed by David Abrahams) and became the Boost Graph Library. The first release of the BGL was September 27, 2000. The BGL is not a "frozen" library. It continues to grow as new algorithms are contributed, and it continues to evolve to meet users' needs. We encourage readers to participate in the Boost group and help with extensions to the BGL.

What Is Boost?

Boost is an online community that encourages development and peer-review of free C++ libraries. The emphasis is on portable and high-quality libraries that work well with (and are in the same spirit as) the C++ Standard Library. Members of the community submit proposals (library designs and implementations) for review. The Boost community (led by a review manager) then reviews the library, provides feedback to the contributors, and finally renders a decision as to whether the library should be included in the Boost library collection. The libraries are available at the Boost Web site ***http://www.boost.org***. In addition, the Boost mailing list provides an important forum for discussing library plans and for organizing collaboration.

Obtaining and Installing the BGL Software

The Boost Graph Library is available as part of the Boost library collection, which can be obtained in several different ways. The CD accompanying this book contains version 1.25.1 of the Boost library collection. In addition, releases of the Boost library collection can be obtained with your Web browser at ***http://www.boost.org/boost_all.zip*** for the Windows zip archive of the latest release and ***http://www.boost.org/boost_all.tar.gz*** for the UNIX archive of the latest release. The Boost libraries can also be downloaded via FTP at ***ftp://boost.sourceforge.net/pub-/boost/release/***.

The zip archive of the Boost library collection can be unzipped by using WinZip or other similar tools. The UNIX "tar ball" can be expanded using the following command:

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

Extracting the archive creates a directory whose name consists of the word ***boost*** and a version number. For example, extracting the Boost release 1.25.1 creates a directory ***boost_1_25_1***. Under this top directory, are two principal subdirectories: ***boost*** and ***libs***. The subdirectory ***boost*** contains the header files for all the libraries in the collection. The subdirectory ***libs*** contains a separate subdirectory for each library in the collection. These subdirectories contain library-specific source and documentation files. You can point your Web browser to ***boost_1_25_1/index.htm*** and navigate the whole Boost library collection.

All of the BGL header files are in the directory ***boost/graph/***. However, other Boost header files are needed since BGL uses other Boost components. The HTML documentation is in ***libs/graph/doc/*** and the source code for the examples is in ***libs/graph/example/***. Regression tests for BGL are in ***libs/graph/test/***. The source files in ***libs/graph/src/*** implement the Graphviz file parsers and printers.

Except as described next, there are no compilation and build steps necessary to use BGL. All that is required is that the Boost header file directory be added to your compiler's include path. For example, using Windows 2000, if you have unzipped release 1.25.1 from ***boost_all.zip*** into the top level directory of your C drive, for Borland, GCC, and Metrowerks compilers add ***-Ic:/boost_1_25_1*** to the compiler command line, and for the Microsoft Visual C++ compiler add ***/I "c:/boost_1_25_1"***. For IDEs, add ***c:/boost_1_25_1*** (or whatever you have renamed it to) to the include search paths using the appropriate dialog. Before using the BGL interface to LEDA or Stanford GraphBase, LEDA or GraphBase must be installed according to their installation instructions. To use the ***read_graphviz()*** functions (for reading AT&T Graphviz files), you must build and link to an additional library under ***boost_1_25_1/libs/graph/src***.

The Boost Graph Library is written in ISO/IEC Standard C++ and compiles with most C++ compilers. For an up-to-date summary of the compatibility with a particular compiler, see the "Compiler Status" page at the Boost Web site ***http://www.boost.org/status/-compiler_status.html***.

How to Use This Book

This book is both a user guide and reference manual for the BGL. It is intended to allow the reader to begin using the BGL for real-life graph problems. This book should also be interesting for programmers who wish to learn more about generic programming. Although there are many books about how to use generic libraries (which in almost all cases means how to use the STL or Standard Library), there is very little available about how actually to build generic software. Yet generic programming is a vitally important new paradigm for software development. We hope that, by way of example, this book will show the reader how to do (and not simply use) generic programming and to apply and extend the generic programming paradigm beyond the basic container types and algorithms of the STL.

The third partner to the user guide and reference manual is the BGL code itself. The BGL code is not simply academic and instructional. It is intended to be used.

For students learning about graph algorithms and data structures, BGL provides a comprehensive graph algorithm framework. The student can concentrate on learning the important theory behind graph algorithms without becoming bogged down and distracted in too many implementation details.

For practicing programmers, BGL provides high-quality implementations of graph data structures and algorithms. Programmers will realize significant time saving from this reliability. Time that would have otherwise been spent developing (and debugging) complicated graph data structures and algorithms can now be spent in more productive pursuits. Moreover, the flexible interface to the BGL will allow programmers to apply graph algorithms in settings where a graph may only exist implicitly.

For the graph theoretician, this book makes a persuasive case for the use of generic programming for implementing graph-theoretic algorithms. Algorithms written using the BGL interface will have broad applicability and will be able to be reused in numerous settings.

We assume that the reader has a good grasp of C++. Since there are many sources where the reader can learn about C++, we do not try to teach it here (see the references at the end of the book—*The C++ Programming Language*, Special ed., by Bjarne Stroustrup [42] and *C++ Primer*, 3rd ed., by Josee Lajoie and Stanley B. Lippman [25] are our recommendations). We also assume some familiarity with the STL (see *STL Tutorial and Reference Guide* by David R. Musser, Gillmer J. Derge, and Atul Saini [34] and *Generic Programming and the STL* by Matthew Austern [3]). We do, however, present some of the more advanced C++ features used to implement generic libraries in general and the BGL in particular.

Some necessary graph theory concepts are introduced here, but not in great detail. For a detailed discussion of elementary graph theory see *Introduction to Algorithms* by T. H. Cormen, C. E. Leiserson, and R. L. Rivest [10].

Literate Programming

The program examples in this book are presented using the literate programming style developed by Donald Knuth. The literate programming style consists of writing source code and

documentation together in the same file. A tool then automatically converts the file into both a pure source code file and into a documentation file with pretty-printed source code. The literate programming style makes it easier to ensure that the code examples in the book really compile and run and that they stay consistent with the text.

The source code for each example is broken up into *parts*. Parts can include references to other parts. For example, the following part labeled "Merge sort function definition" refers to the parts labeled "Divide the range in half and sort each half" and "Merge the two halves." An example often starts with a part that provides an outline for the entire computation, which is then followed by other parts that fill in the details. For example, the following function template is a generic implementation of the merge sort algorithm [10]. There are two steps in the algorithm, sorting each half of the range and then merging the two halves.

⟨ Merge sort function definition xxiia ⟩ ≡

```
template <typename RandomAccessIterator, typename Compare>
void merge_sort(RandomAccessIterator first, RandomAccessIterator last, Compare cmp)
{
  if (first + 1 < last) {
    ⟨Divide the range in half and sort each half xxiib⟩
    ⟨Merge the two halves xxiic⟩
  }
}
```

Typically, the size of each part is limited to a few lines of code that carry out a specific task. The names for the parts are chosen to convey the essence of the task.

⟨ Divide the range in half and sort each half xxiib ⟩ ≡

```
RandomAccessIterator mid = first + (last − first)/2;
merge_sort(first, mid, cmp);
merge_sort(mid, last, cmp);
```

The *std::inplace_merge()* function does the main work of this algorithm, creating a single sorted range out of two sorted subranges.

⟨ Merge the two halves xxiic ⟩ ≡

```
std::inplace_merge(first, mid, last, cmp);
```

Parts are labeled with a descriptive name, along with the page number on which the part is defined. If more than one part is defined on a page, the definitions are distinguished by a letter.

Sometimes a file name is used for the label of a part. This means that the part is written out to a file. Many of the examples in the book are written out to files, and can be found in the *libs/graph/example/* directory of the Boost distribution. The following example shows the *merge_sort()* function being output to a header file.

⟨ ***merge-sort.hpp*** xxiii ⟩ ≡

```
#ifndef MERGE_SORT_HPP
#define MERGE_SORT_HPP

  ⟨Merge sort function definition xxiia⟩

#endif // MERGE_SORT_HPP
```

The Electronic Reference

An electronic version of the book is included on the accompanying CD, in the file ***bgl-book.pdf***. The electronic version is searchable and is fully hyperlinked, making it a useful companion for the printed version. The hyperlinks include all internal references such as the literate programming "part" references as well as links to external Web pages.

Acknowledgments

We owe many debts of thanks to a number of individuals who both inspired and encouraged us in developing the BGL and in writing this book.

A most profound thanks goes to Alexander Stepanov and David Musser for their pioneering work in generic programming, for their continued encouragement of our work, and for contributions to the BGL. We especially thank David Musser for his careful proofreading of this book. Matthew Austern's work on documenting the concepts of the STL provided a foundation for creating the concepts in the BGL. We thank Dietmar Kühl for his work on generic graph algorithms and design patterns; especially for the property map abstraction. This work would not have been possible without the expressive power of Bjarne Stroustrup's C++ language.

Dave Abrahams, Jens Maurer, Dietmar Kühl, Beman Dawes, Gary Powell, Greg Colvin and the rest of the group at Boost provided valuable input to the BGL interface, numerous suggestions for improvement, and proofreads of this book. We also thank the following BGL users whose questions helped to motivate and improve BGL (as well as this book): Gordon Woodhull, Dave Longhorn, Joel Phillips, Edward Luke, and Stephen North.

Thanks to a number of individuals who reviewed the book during its development: Jan Christiaan van Winkel, David Musser, Beman Dawes, and Jeffrey Squyres.

A great thanks to our editor Deborah Lafferty; Kim Arney Mulcahy, Cherly Ferguson, and Marcy Barnes, the production coordinators; and the rest of the team at Addison–Wesley. It was a pleasure to work with them.

Our original work on the BGL was supported in part by NSF grant ACI-9982205. Parts of the BGL were completed while the third author was on sabbatical at Lawrence Berkeley National Laboratory (where the first two authors were occasional guests). All of the graph drawings in this book were produced using the dot program from the Graphviz package.

License

The BGL software is released under an open source "artistic" license. A copy of the BGL license is included with the source code in the LICENSE file.

The BGL may be used freely for both commercial and noncommercial use. The main restriction on BGL is that modified source code can only be redistributed if it is clearly marked as a nonstandard version of BGL. The preferred method for the distribution of BGL, and for submitting changes, is through the Boost Web site.

Chapter 3

A BGL Tutorial

As discussed in the previous chapter, *concepts* play a central role in generic programming. Concepts are the interface definitions that allow many different components to be used with the same algorithm. The Boost Graph Library defines a large collection of concepts that cover various aspects of working with a graph, such as traversing a graph or modifying its structure. In this chapter, we introduce these concepts and also provide some motivation for the choice of concepts in the BGL.

From the description of the generic programming process (see page 19), concepts are derived from the algorithms that are used to solve problems in particular domains. In this chapter we examine the problem of tracking file dependencies in a build system. For each subproblem, we examine generalizations that can be made to the solutions, with the goal of increasing the reusability (the genericity) of the solution. The result, at the end of the chapter, is a generic graph algorithm and its application to the file-dependency problem.

Along the way, we also cover some of the more mundane but necessary topics, such as how to create a graph object and fill in the vertices and edges.

3.1 File Dependencies

A common use of the graph abstraction is to represent dependencies. One common type of dependency that we programmers deal with on a routine basis is that of compilation dependencies between files in programs that we write. Information about these dependencies is used by programs such as *make*, or by IDEs such as Visual C++, to determine which files must be recompiled to generate a new version of a program (or, in general, of some target) after a change has been made to a source file.

Figure 3.1 shows a graph that has a vertex for each source file, object file, and library that is used in the ***killerapp*** program. An edge in the graph shows that a target depends on another target in some way (such as a dependency due to inclusion of a header file in a source file, or due to an object file being compiled from a source file).

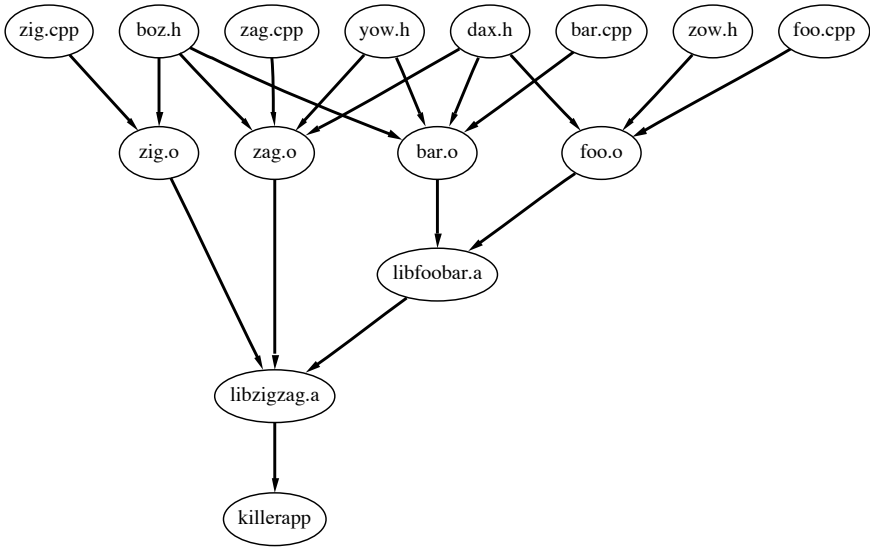


Figure 3.1 A graph representing file dependencies.

Answers to many of the questions that arise in creating a build system such as make can be formulated in terms of the dependency graph. We might ask these questions:

- If all of the targets need to be made, in what order should that be accomplished?
- Are there any cycles in the dependencies? A dependency cycle is an error, and an appropriate message should be emitted.
- How many steps are required to make all of the targets? How many steps are required to make all of the targets if independent targets are made simultaneously in parallel (using a network of workstations or a multiprocessor, for example)?

In the following sections these questions are posed in graph terms, and graph algorithms are developed to provide solutions. The graph in Figure 3.1 is used in all of the examples.

3.2 Graph Setup

Before addressing these questions directly, we must first find a way to represent the file-dependency graph of Figure 3.1 in memory. That is, we need to construct a BGL graph object.

Deciding Which Graph Class To Use

There are several BGL graph classes from which to choose. Since BGL algorithms are generic, they can also be used with any conforming user-defined graph class, but in this chapter we restrict our discussion to BGL graph classes. The principle BGL graph classes are the ***adjacency_list*** and ***adjacency_matrix*** classes. The ***adjacency_list*** class is a good choice for most situations, particularly for representing sparse graphs. The file-dependencies graph has only a few edges per vertex, so it is sparse. The ***adjacency_matrix*** class is a good choice for representing dense graphs, but a very bad choice for sparse graphs.

The ***adjacency_list*** class is used exclusively in this chapter. However, most of what is presented here also applies directly to the ***adjacency_matrix*** class because its interface is almost identical to that of the ***adjacency_list*** class. Here we use the same variant of ***adjacency_list*** as was used in §1.4.1.

```
typedef adjacency_list<
  listS,       // Store out-edges of each vertex in a std::list
  vecS,        // Store vertex set in a std::vector
  directedS    // The file dependency graph is directed
  > file_dep_graph;
```

Constructing a Graph Using Edge Iterators

In §1.2.4 we showed how the ***add_vertex()*** and ***add_edge()*** functions can be used to create a graph. Those functions add vertices and edges one at a time, but in many cases one would like to add them all at once. To meet this need the ***adjacency_list*** graph class has a constructor that takes two iterators that define a range of edges. The edge iterators can be any InputIterator that dereference to a ***std::pair*** of integers (i, j) that represent an edge in the graph. The two integers i and j represent vertices where $0 \leq i < |V|$ and $0 \leq j < |V|$. The ***n*** and ***m*** parameters say how many vertices and edges will be in the graph. These parameters are optional, but providing them improves the speed of graph construction. The graph properties parameter ***p*** is attached to the graph object. The function prototype for the constructor that uses edge iterators is as follows:

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

The following code demonstrates the use of the edge iterator constructor to create a graph. The ***std::istream_iterator*** is used to make an input iterator that reads the edges in from the file. The file contains the number of vertices in the graph, followed by pairs of numbers that specify the edges. The second default-constructed input iterator is a placeholder for the end of the input. The ***std::istream_iterator*** is passed directly into the constructor for the graph.

```
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; // read in number of vertices
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);
```

Since the value type of the *std::istream_iterator* is *std::pair*, an input operator needs to be defined for *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 Compilation Order

The first question that we address is that of specifying an order in which to build all of the targets. The primary consideration here is ensuring that before building a given target, all the targets that it depends on are already built. This is, in fact, the same problem as in §1.4.1, scheduling a set of errands.

3.3.1 Topological Sort via DFS

As mentioned in §1.4.2, a topological ordering can be computed using a depth-first search (DFS). To review, a DFS visits all of the vertices in a graph by starting at any vertex and then choosing an edge to follow. At the next vertex another edge is chosen to follow. This process continues until a dead end (a vertex with no out-edges that lead to a vertex not already discovered) is reached. The algorithm then backtracks to the last discovered vertex that is adjacent to a vertex that is not yet discovered. Once all vertices reachable from the starting vertex are explored, one of the remaining unexplored vertices is chosen and the search continues from there. The edges traversed during each of these separate searches form a *depth-first tree*; and all the searches form a *depth-first forest*. A depth-first forest for a given graph is not unique; there are typically several valid DFS forests for a graph because the order in which the adjacent vertices are visited is not specified. Each unique ordering creates a different DFS tree.

Two useful metrics in a DFS are the *discover time* and *finish time* of a vertex. Imagine that there is an integer counter that starts at zero. Every time a vertex is first visited, the value of the counter is recorded as the discover time for that vertex and the value of the counter is incremented. Likewise, once all of the vertices reachable from a given vertex have been

visited, then that vertex is finished. The current value of the counter is recorded as the finish time for that vertex and the counter is incremented. The discover time of a parent in a DFS tree is always earlier than the discover time of a child. Similarly, the finish time of a parent is always later than the finish time of a child. Figure 3.2 shows a depth-first search of the file dependency graph, with the tree edges marked with black lines and with the vertices labeled with their discover and finish times (written as discover/finish).

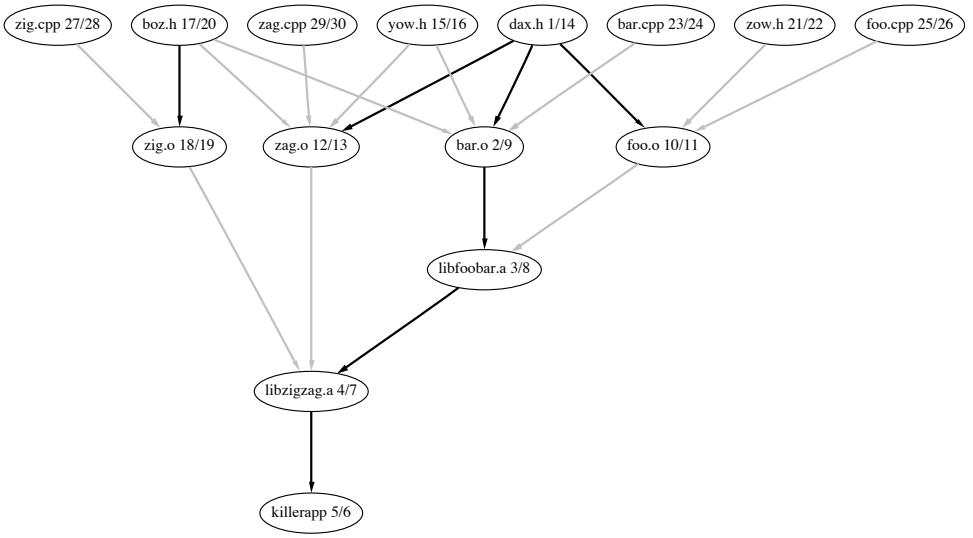


Figure 3.2 A depth-first search of the file dependency graph. Edges in the DFS tree are black and non-tree edges are gray. Each vertex is labeled with its discover and finish time.

The relationship between topological ordering and DFS can be explained by considering three different cases at the point in the DFS when an edge (u, v) is examined. For each case, the finish time of v is always earlier than the finish time of u. Thus, the finish time is simply the topological ordering (in reverse).

1. Vertex v is not yet discovered. This means that v will become a descendant of u and will therefore end up with a finish time earlier than u because DFS finishes all descendants of u before finishing u.

2. Vertex v was discovered in an earlier DFS tree. Therefore, the finish time of v must be earlier than that of u.

3. Vertex v was discovered earlier in the current DFS-tree. If this case occurs, the graph contains a cycle and a topological ordering of the graph is not possible. A *cycle* is a path of edges such that the first vertex and last vertex of the path are the same vertex.

The main part of the depth-first search is a recursive algorithm that calls itself on each adjacent vertex. We will create a function named *topo_sort_dfs()* that will implement a depth-first search modified to compute a topological ordering. This first version of the function will be a straightforward, nongeneric function. In the following sections we will make modifications that will finally result in a generic algorithm.

The parameters to *topo_sort_dfs()* include the graph, the starting vertex, a pointer to an array to record the topological order, and an array for recording which vertices have been visited. The *topo_order* pointer starts at the end of the array and then decrements to obtain the topological ordering from the reverse topological ordering. Note that *topo_order* is passed by reference so that the decrement made to it in each recursive call modifies the original object (if *topo_order* were instead passed by value, the decrement would happen instead to a copy of the original object).

```
void
topo_sort_dfs(const file_dep_graph& g, vertex_t u, vertex_t*& topo_order, int* mark)
{
  mark[u] = 1; // 1 means visited, 0 means not yet visited
  ⟨For each adjacent vertex, make recursive call 47⟩
  *--topo_order = u;
}
```

The *vertex_t* type and *edge_t* types are the vertex and edge descriptors for the *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 Marking Vertices Using External Properties

Each vertex should be visited only once during the search. To record whether a vertex has been visited, we can mark it by creating an array that stores the mark for each vertex. In general, we use the term *external property storage* to refer to the technique of storing vertex or edge properties (marks are one such property) in a data structure like an array or hash table that is separate from the graph object (i.e., that is *external* to the graph). Property values are looked up based on some key that can be easily obtained from a vertex or edge descriptor. In this example, we use a version of *adjacency_list* where the the vertex descriptors are integers from zero to *num_vertices(g) - 1*. As a result, the vertex descriptors themselves can be used as indexes into the mark array.

3.3.3 Accessing Adjacent Vertices

In the *topo_sort_dfs()* function we need to access all the vertices adjacent to the vertex *u*. The BGL concept AdjacencyGraph defines the interface for accessing adjacent vertices. The function *adjacent_vertices()* takes a vertex and graph object as arguments and returns a pair

of iterators whose value type is a vertex descriptor. The first iterator points to the first adjacent vertex, and the second iterator points past the end of the last adjacent vertex. The adjacent vertices are not necessarily ordered in any way. The type of the iterators is the ***adjacency_iterator*** type obtained from the ***graph_traits*** class. The reference section for ***adjacency_list*** (§14.1.1) reveals that the graph type we are using, ***adjacency_list***, models the AdjacencyGraph concept. We may therefore correctly use the function ***adjacent_vertices()*** with our file dependency graph. The code for traversing the adjacent vertices in ***topo_sort_dfs()*** follows.

⟨ For each adjacent vertex, make recursive call 47 ⟩ ≡

```
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 Traversing All the Vertices

One way to ensure that an ordering is obtained for every vertex in the graph (and not just those vertices reachable from a particular starting vertex) is to surround the call to ***topo_sort_dfs()*** with a loop through every vertex in the graph. The interface for traversing all the vertices in a graph is defined in the VertexListGraph concept. The ***vertices()*** function takes a graph object and returns a pair of vertex iterators. The loop through all the vertices and the creation of the mark array is encapsulated in a function called ***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]);
}
```

To make the output from ***topo_sort()*** more user friendly, we need to convert the vertex integers to their associated target names. We have the list of target names stored in a file (in the order that matches the vertex number) so we read in this file and store the names in an array, which we then use when printing the names of the vertices.

```
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];
```

Now we create the order array to store the results and then apply the topological sort function.

```
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;
```

The output is

```
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 Cyclic Dependencies

One important assumption in the last section is that the file dependency graph does not have any cycles. As stated in §3.3.1, a graph with cycles does not have a topological ordering. A well-formed makefile will have no cycles, but errors do occur, and our build system should be able to catch and report such errors.

Depth-first search can also be used for the problem of detecting cycles. If DFS is applied to a graph that has a cycle, then one of the branches of a DFS tree will loop back on itself. That is, there will be an edge from a vertex to one of its ancestors in the tree. This kind of edge is called a *back edge*. This occurrence can be detected if we change how we mark vertices. Instead of marking each vertex as visited or not visited, we use a three-way coloring scheme: white means undiscovered, gray means discovered but still searching descendants, and black means the vertex and all of its descendants have been discovered. Three-way coloring is useful for several graph algorithms, so the header file ***boost/graph/properties.hpp*** defines the following enumerated type.

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

A cycle in the graph is identified by an adjacent vertex that is gray, meaning that an edge loops back to an ancestor. The following code is a version of DFS instrumented to detect cycles.

```
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; // cycle detected, return immediately
    else if (color[*vi] == gray_color) // *vi is an ancestor!
      return true;
  color[u] = black_color;
  return false;
}
```

As with the topological sort, in the ***has_cycle()*** function the recursive DFS function call is placed inside of a loop through all of the vertices so that we catch all of the DFS trees in the graph.

```
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 true;
  return false;
}
```

3.5 Toward a Generic DFS: Visitors

At this point we have completed two functions, ***topo_sort()*** and ***has_cycle()***, each of which is implemented using depth-first search, although in slightly different ways. However, the fundamental similarities between the two functions provide an excellent opportunity for code reuse. It would be much better if we had a single generic algorithm for depth-first search that expresses the commonality between ***topo_sort()*** and ***has_cycle()*** and then used parameters to customize the DFS for each of the different problems.

The design of the STL gives us a hint for how to create a suitably parameterized DFS algorithm. Many of the STL algorithms can be customized by providing a user-defined function object. In the same way, we would like to parameterize DFS in such a way that ***topo_sort()*** and ***has_cycle()*** can be realized by passing in a function object.

Unfortunately, the situation here is a little more complicated than in typical STL algorithms. In particular, there are several different locations in the DFS algorithm where customized actions must occur. For instance, the ***topo_sort()*** function records the ordering at the

bottom of the recursive function, whereas the *has_cycle()* function needs to insert an operation inside the loop that examines the adjacent vertices.

The solution to this problem is to use a function object with more than one callback member function. Instead of a single *operator()* function, we use a class with several member functions that are called at different locations (we refer to these places as *event points*). This kind of function object is called an *algorithm visitor*. The DFS visitor will have five member functions: *discover_vertex()*, *tree_edge()*, *back_edge()*, *forward_or_cross_edge()*, and *finish_vertex()*. Also, instead of iterating over the adjacent vertices, we iterator over out-edges to allow passing edge descriptors to the visitor functions and thereby provide more information to the user-defined visitor. This code for a DFS function has a template parameter for a visitor:

```
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);
      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);
  }
}
```

The five member functions of the visitor provide the flexibility we need, but a user that only wants to add one action should not have to write four empty member functions. This is easily solved by creating a default visitor from which user-defined visitors can be derived.

```
struct default_dfs_visitor {
  template <typename V, typename G>
  void discover_vertex(V, const G&) { }

  template <typename E, typename G>
  void tree_edge(E, const G&) { }

  template <typename E, typename G>
  void back_edge(E, const G&) { }

  template <typename E, typename G>
  void forward_or_cross_edge(E, const G&) { }

  template <typename V, typename G>
  void finish_vertex(V, const G&) { }
};
```

To demonstrate that this generic DFS can solve our problems, we reimplement the *topo_sort()* and *has_cycle()* functions. First we need to create a visitor that records the topological ordering on the "finish vertex" event point. The code for this visitor follows.

```
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;
};
```

Only two lines of code are required in the body of *topo_sort()* when implemented using generic DFS. One line creates the visitor object and one line calls the generic DFS.

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

To reimplement the *has_cycle()* function, we use a visitor that records that the graph has a cycle whenever the back edge event point occurs.

```
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;
};
```

The new *has_cycle()* function creates a cycle detector object and passes it to the generic DFS.

```
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 Graph Setup: Internal Properties

Before addressing the next question about file dependencies, we are going to take some time out to switch to a different graph type. In the previous sections we used arrays to store information such as vertex names. When vertex or edge properties have the same lifetime as the graph object, it can be more convenient to have the properties somehow embedded in the graph itself (we call these *internal properties*). If you were writing your own graph class you might add data members for these properties to a vertex or edge struct.

The *adjacency_list* class has template parameters that allow arbitrary properties to be attached to the vertices and edge: the *VertexProperties* and *EdgeProperties* parameters. These template parameters expect the argument types to be the *property*<*Tag, T*> class, where *Tag* is a type that specifies the property and *T* gives the type of the property object. There are a number of predefined property tags (see §15.2.3) such as *vertex_name_t* and *edge_weight_t*. For example, to attach a *std::string* to each vertex use the following property type:

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

If the predefined property tags do not meet your needs, you can create a new one. One way to do this is to define an enumerated type named *vertex_xxx_t* or *edge_xxx_t* that contains an enum value with the same name minus the *_t* and give the enum value a unique number. Then use *BOOST_INSTALL_PROPERTY* to create the required specializations of the *property_kind* and *property_num* traits classes.[1] Here we create compile-time cost property that we will use in the next section to compute the total compile time.

```
namespace boost {
  enum vertex_compile_cost_t { vertex_compile_cost = 111 }; // a unique #
  BOOST_INSTALL_PROPERTY(vertex, compile_cost);
}
```

The *property* class has an optional third parameter that can be used to nest multiple *property* classes thereby attaching multiple properties to each vertex or edge. Here we create a new typedef for the graph, this time adding two vertex properties and an edge property.

[1] Defining new property tags would be much simpler if more C++ compilers were standards conformant.

```
typedef adjacency_list<
  listS,       // Store out-edges of each vertex in a std::list
  listS,       // Store vertex set in a std::list
  directedS, // The file dependency graph is directed
  // vertex properties
  property<vertex_name_t, std::string,
    property<vertex_compile_cost_t, float,
      property<vertex_distance_t, float,
        property<vertex_color_t, default_color_type> > > >,
  // an edge property
  property<edge_weight_t, float>
  > file_dep_graph2;
```

We have also changed the second template argument to ***adjacency_list*** from ***vecS*** to ***listS***. This has some important implications. If we were to remove a vertex from the graph it would happen in constant time (with ***vecS*** the vertex removal time is linear in the number of vertices and edges). On the down side, the vertex descriptor type is no longer an integer, so storing properties in arrays and using the vertex as an offset will no longer work. However, the separate storage is no longer needed because we now have the vertex properties stored in the graph.

In §1.2.2 we introduced the notion of a property map. To review, a property map is an object that can be used to map from a key (such as a vertex) to a value (such as a vertex name). When properties have been specified for an ***adjacency_list*** (as we have just done), property maps for these properties can be obtained using the PropertyGraph interface. The following code shows an example of obtaining two property maps: one for vertex names and another for compile-time cost. The ***property_map*** traits class provides the type of the property map.

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

The ***get()*** function returns a property map object.

```
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);
```

There will be another file containing the estimated compile time for each makefile target. We read this file using a ***std::ifstream*** and write the properties into the graph using the property maps, ***name_map*** and ***compile_cost_map***. These property maps are models of ***LvaluePropertyMap*** so they have an ***operator[]()*** that maps from vertex descriptors to a reference to the appriopriate vertex property object.

```
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];
}
```

In the following sections we will modify the topological sort and DFS functions to use the property map interface to access vertex properties instead of hard-coding access with a pointer to an array.

3.7 Compilation Time

The next questions we need to answer are, "How long will a compile take?" and "How long will a compile take on a parallel computer?" The first question is easy to answer. We simply sum the compile time for all the vertices in the graph. Just for fun, we do this computation using the ***std::accumulate*** function. To use this function we need iterators that, when dereferenced, yield the compile cost for the vertex. The vertex iterators of the graph do not provide this capability. When dereferenced, they yield vertex descriptors. Instead, we use the ***graph_property_iter_range*** class (see §16.8) to generate the appropriate iterators.

```
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 << "total (sequential) compile time: "
          << std::accumulate(ci, ci_end, 0.0) << std::endl;
```

The output of the code sequence is

```
total (sequential) compile time: 21.3
```

Now suppose we have a parallel super computer with hundreds of processors. If there are build targets that do not depend on each other, then they can be compiled at the same time on different processors. How long will the compile take now? To answer this, we need to determine the critical path through the file dependency graph. Or, to put it another way, we need to find the longest path through the graph.

The black lines in Figure 3.3 show the file dependency of ***libfoobar.a***. Suppose that we have already determined when ***bar.o*** and ***foo.o*** will finish compiling. Then the compile time for ***libfoobar.a*** will be the longer of the times for ***bar.o*** and ***foo.o*** plus the cost for linking them together to form the library file.

Now that we know how to compute the "distance" for each vertex, in what order should we go through the vertices? Certainly if there is an edge (u, v) in the graph, then we better compute the distance for u before v because computing the distance to v requires the distance to u. This should sound familiar. We need to consider the vertices in topological order.

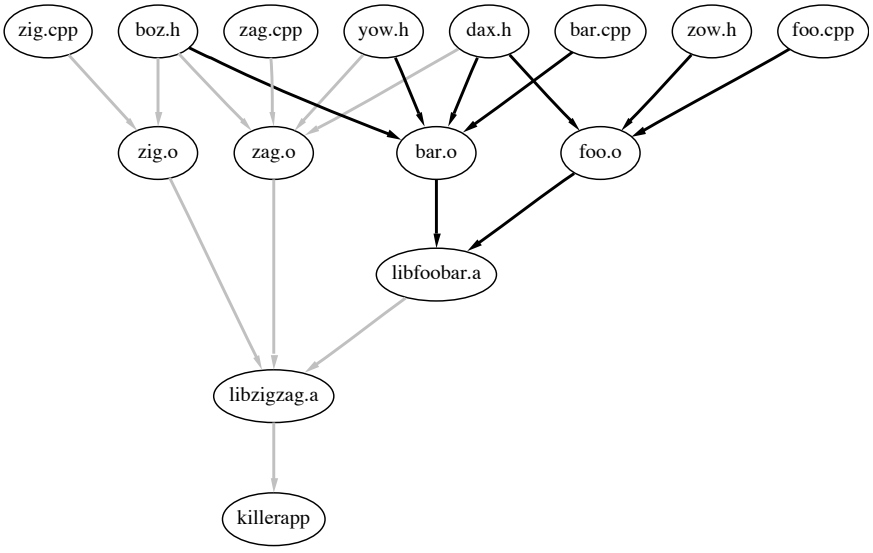


Figure 3.3 Compile time contributions to ***libfoobar.a***.

3.8 A Generic Topological Sort and DFS

Due to the change in graph type (from ***file_dep_graph*** to ***file_dep_graph2***) we can no longer use the ***topo_sort*()** function that we developed in §3.4. Not only does the graph type not match, but also the ***color*** array used inside of ***generic_dfs_v1*()** relies on the fact that vertex descriptors are integers (which is not true for ***file_dep_graph2***). These problems give us an opportunity to create an even more generic version of topological sort and the underlying DFS. We parameterize the ***topo_sort*()** function in the following way.

- The specific type ***file_dep_graph*** is replaced by the template parameter ***Graph***. Merely changing to a template parameter does not help us unless there is a standard interface shared by all the graph types that we wish to use with the algorithm. This is where the BGL graph traversal concepts come in. For ***topo_sort*()** we need a graph type that models the VertexListGraph and IncidenceGraph concepts.

- Using a ***vertex_t**** for the ordering output is overly restrictive. A more generalized way to output a sequence of elements is to use an output iterator, just as the algorithms in the C++ Standard Library do. This gives the user much more options in terms of where to store the results.

- We need to add a parameter for the color map. To make this as general as possible, we only want to require what is *essential*. In this case, the ***topo_sort*()** function needs to be able to map from a vertex descriptor to a marker object for that vertex. The Boost Property Map Library (see Chapter 15) defines a minimalistic interface for performing

this mapping. Here we use the LvaluePropertyMap interface. The internal ***color_map*** that we obtained from the graph in §3.6 implements the LvaluePropertyMap interface, as does the color array we used in §3.3.4. A pointer to an array of color markers can be used as a property map because there are function overloads in ***boost/property_map.hpp*** that adapt pointers to satisfy the LvaluePropertyMap interface.

The following is the implementation of our generic ***topo_sort()***. The ***topo_visitor*** and ***generic_dfs_v2()*** are discussed next.

```
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);
}
```

The ***topo_visitor*** class is now a class template to accommodate the output iterator. Instead of decrementing, we now increment the output iterator (decrementing an output iterator is not allowed). To get the same reversal behavior as in the first version of ***topo_sort()***, the user can pass in a reverse iterator or something like a front insert iterator for a list.

```
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;
};
```

The generic DFS changes in a similar fashion, with the graph type and color map becoming parameterized. In addition, we do not *a priori* know the color type, so we must get the color type by asking the ***ColorMap*** for its value type (though the ***property_traits*** class). Instead of using constants such as ***white_color***, we use the color functions defined in ***color_traits***.

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

The logic from the ***dfs_v1*** does not need to change; however, there are a few small changes required due to making the graph type parameterized. Instead of hard-coding ***vertex_t*** as the vertex descriptor type, we extract the appropriate vertex descriptor from the graph type using ***graph_traits***. The fully generic DFS function follows. This function is essentially the same as the BGL ***depth_first_visit()***.

```
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);
}
```

The real BGL ***depth_first_search()*** and ***topological_sort()*** functions are quite similar to the generic functions that we developed in this section. We give a detailed example of using the BGL ***depth_first_search()*** function in §4.2, and the documentation for ***depth_first_search()*** is in §13.2.3. The documentation for ***topological_sort()*** is in §13.2.5.

3.9 Parallel Compilation Time

Now that we have a generic topological sort and DFS, we are ready to solve the problem of finding how long the compilation will take on a parallel computer. First, we perform a topological sort, storing the results in the ***topo_order*** vector. We pass the reverse iterator of the vector into ***topo_sort()*** so that we end up with the topological order (and not the reverse topological order).

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

Before calculating the compile times we need to set up the distance map (which we are using to store the compile time totals). For vertices that have no incoming edges (we call these source vertices), we initialize their distance to zero because compilation of these makefile targets can start right away. All other vertices are given a distance of infinity. We find the source vertices by marking all vertices that have incoming edges.

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

// find source vertices with zero in-degree by marking all vertices with incoming edges
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;

// initialize distances to zero, or for source vertices to the compile cost
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;
```

Now we are ready to compute the distances. We go through all of the vertices stored in *topo_order*, and for each one we update the distance (total compile time) for each adjacent vertex. What we are doing here is somewhat different than what was described earlier. Before, we talked about each vertex looking "up" the graph to compute its distance. Here, we have reformulated the computation so that instead we are pushing distances "down" the graph. The reason for this change is that looking "up" the graph requires access to in-edges, which our graph type does not provide.

```
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];
}
```

The maximum distance value from among all the vertices tells us the total parallel compile time. Again we use *graph_property_iter_range* to create property iterators over vertex distances. The *std::max_element()* function does the work of locating the maximum.

```
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);
std::cout << "total (parallel) compile time: "
          << *std::max_element(ci, ci_end) << std::endl;
```

The output is

***total (parallel) compile time:* 11.9**

Figure 3.4 shows two numbers for each makefile target: the compile cost for the target and the time at which the target will finish compiling during a parallel compile.

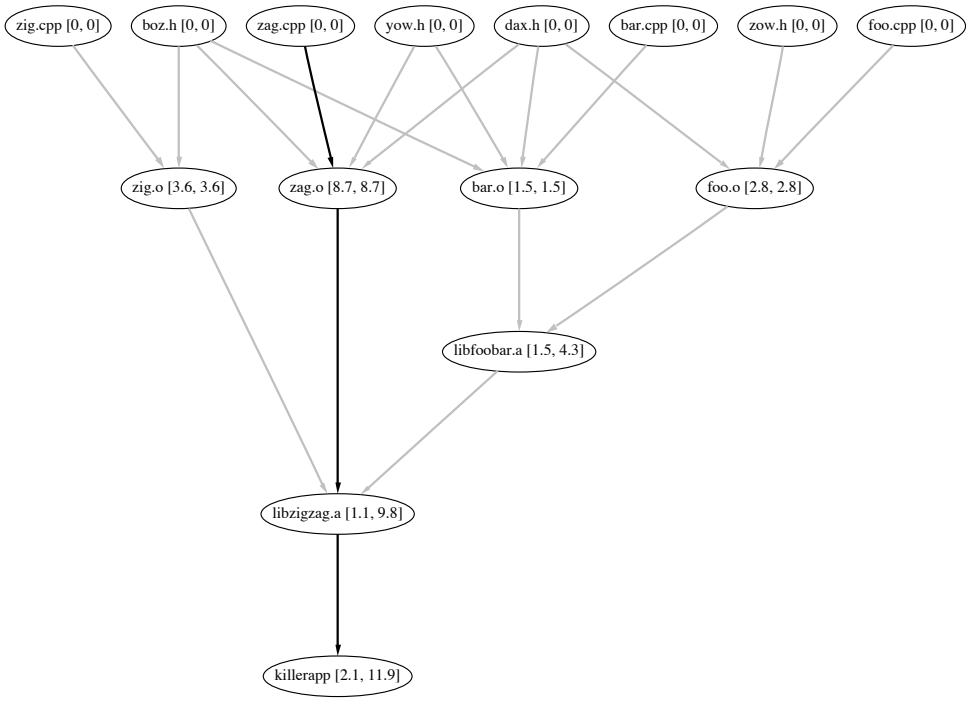


Figure 3.4 For each vertex there are two numbers: compile cost and accumulated compile time. The critical path consists of black lines.

3.10 Summary

In this chapter we have applied BGL to answer several questions that would come up in constructing a software build system: In what order should targets be built? Are there any cyclic dependencies? How long will compilation take? In answering these questions we looked at topological ordering of a directed graph and how this can be computed via a depth-first search.

To implement the solutions we used the BGL ***adjacency_list*** to represent the file dependency graph. We wrote straightforward implementations of topological sort and cycle detection. We then identified common pieces of code and factored them out into a generic implementation of depth-first search. We used algorithm visitors to parameterize the DFS and then wrote specific visitors to implement the topological sort and the cycle detection.

We then looked at using a different variation of the ***adjacency_list*** class that allowed properties such as vertex name and compile cost to be attached to the vertices of the graph. We then further generalized the generic DFS by parameterizing the graph type and the property access method. The chapter finished with an application of the generic topological sort and DFS to compute the time it would take to compile all the targets on a parallel computer.

Index