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).

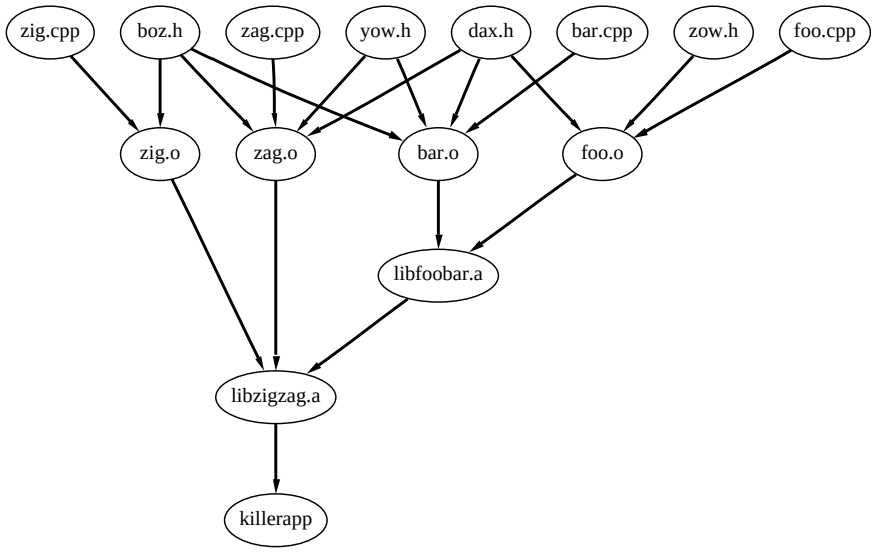

Figure 3.1 A graph representing file dependencies.

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).

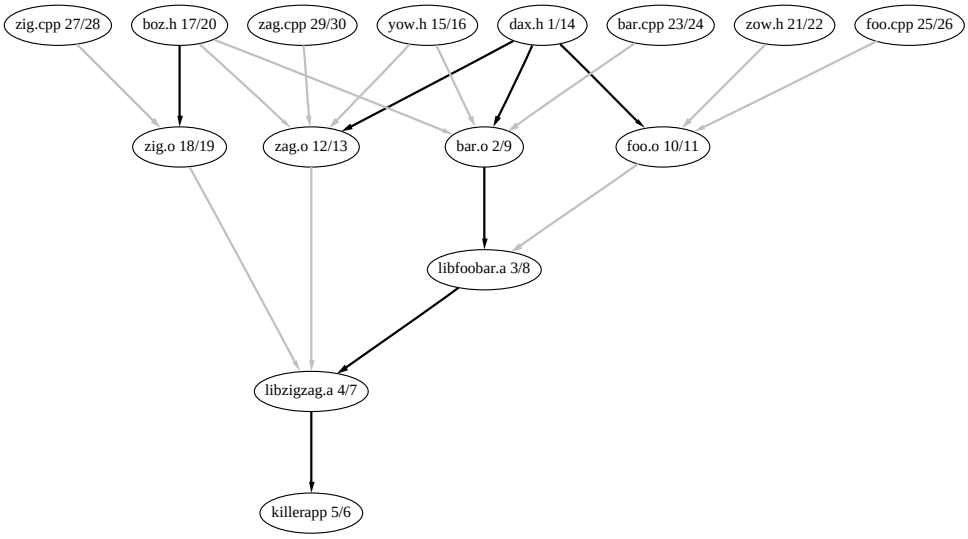

Figure 3.2 A depth-first search of the file dependency graph. Edges in the DFS tree are black and non-tree edges are gray. Each vertex is labeled with its discover and finish time.

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.

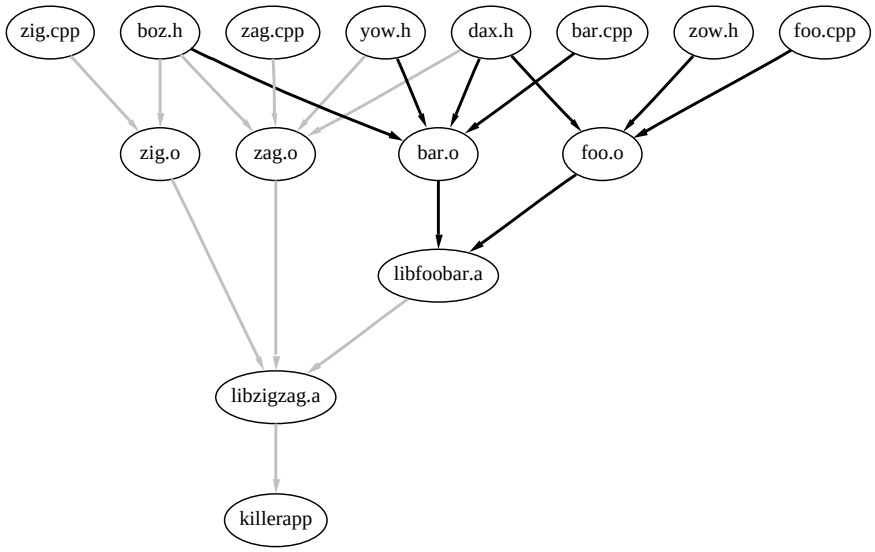

Figure 3.3 Compile time contributions to ***libfoobar.a***.

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.

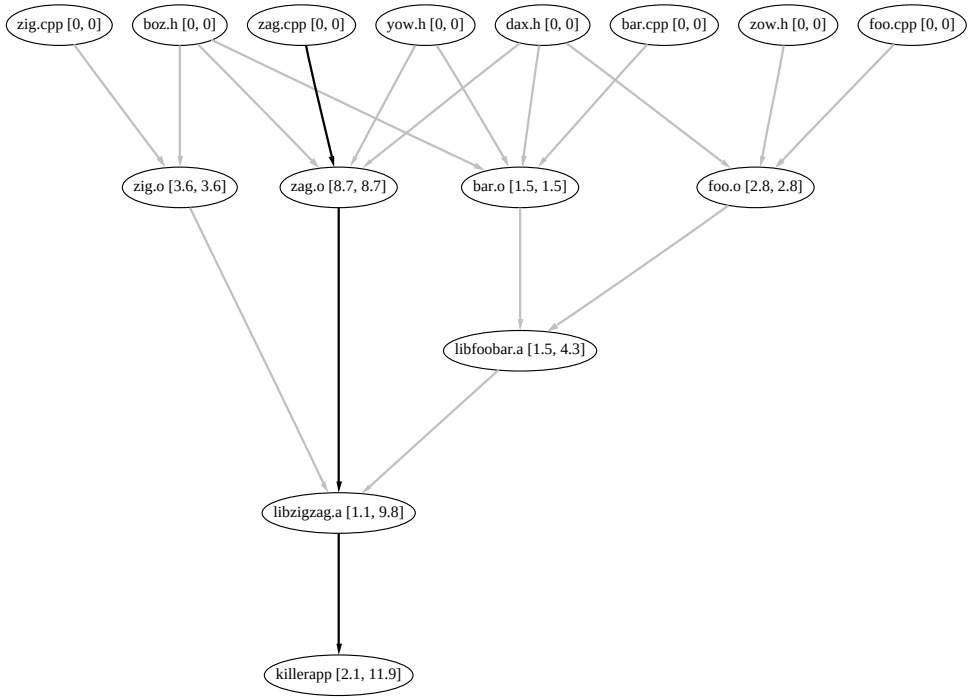

Figure 3.4 For each vertex there are two numbers: compile cost and accumulated compile time. The critical path consists of black lines.

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.