Boosting Productivity with the Boost Graph Library

Dmitry Batenkov

Every C++ programmer is probably familiar with the Standard Template Library (STL). In short, the STL provides us with generic *containers* (e.g. list<T>, vector<T>) and *algorithms* on these containers (e.g. sort(), reverse(), find()) which are also generic (they can be extended via *function objects* such as a search predicate). Another "generic" aspect of STL which is sometimes overlooked is the decoupling of algorithms from containers via *iterators*. That is, you can have your own custom-implemented container which exposes one or more STL-type iterators, and all the STL algorithms will automatically be available for your container.

Now enter Boost Graph Library (BGL), an STL-like library for graphs. It is developed and distributed as part of the Boost Library - a rich collection of C++ libraries, several of which will be included in the next C++ standard. In this tutorial we will learn to use BGL in common programming tasks involving graphs.

Basic use

There are two standard graph containers: adjacency_list and adjacency_matrix. When using adjacency_matrix, all the edges will be stored in a V-by-V matrix where V is the number of vertices. Therefore, this situation will be suitable for dense graphs, for all other cases adjacency_list is preferred. Strictly speaking, a BGL graph is an abstract concept which is realized by three iterators: an iterator for all vertices, an iterator for all edges and an iterator for the edges of a given vertex. So if you already have a complex graph object and you do not wish to copy it into one of the BGL pre-defined graph types, you can write an adaptor (several such adaptors are provided by BGL). In this tutorial we will use only the predefined adjacency_list.

So let us define our graph:

```
#include <boost/graph/
    adjacency_list.hpp>
using namespace boost;
typedef adjacency_list<vecS,vecS,
    undirectedS> Graph;
```

The first two template parameters indicate what kind of storage to use for outgoing edges of each vertex, and for all the vertices. The choice of these affects the space complexity of the graph object and the time complexity of graph operations. The third parameter indicates whether the graph is directed or not.

Now let us fill our graph. The vertices of the graph are usually represented by integer numbers (0,1,...,V-1), in BGL terminology these are callled *vertex descriptors*. During construction, we can either specify the number of vertices or add them later with add_vertex(). Likewise, edges can be specified during construction via a pair of iterators, or added later with add_edge(). Below we create a random graph:

```
#include <boost/graph/random.hpp>
#include <boost/random/
    linear_congruential.hpp>
#include <ctime>
...
Graph g;
minstd_rand0 rng(time(0));
generate_random_graph(g, n_edges,
      n_vertices, rng);
```

A widely used concept in BGL is that of *property maps*. Many graph algorithms expect some values to be attached to edges and vertices, either as part of their input/output or for internal calculations. Since there are many ways to do this (for example, as a vector indexed by vertex descriptor, or a member variable), the BGL defines an abstract interface for property maps. Note that the "interface" here does not refer to any C++ construct such as a pure abstract class, but rather to a set of rules which are enforced by the compiler at template instantiation time. This leads in many cases to hard-to-understand error messages, and is treated by introducing *concept checking* (an advanced topic which we should not touch here).

Let us apply one of the many algorithms available in BGL to our graph. The list is impressive, growing with each release of Boost. Let's take one of the most simple ones, connected_components. It expects as input a property map object to which it will write the component ID associated with each vertex.

```
typedef graph_traits<Graph>::
      vertex_descriptor V;

vector_property_map<int>
    components_map;
int count = connected_components
    (g, components_map);
cout << "The␣graph␣has" << count
    << "␣connected␣components." <<
          endl;
for (V v = 0; v < num_vertices(g);
        ++v)
      cout << v << "␣belongs␣to␣
        component␣"
    << components_map[v] << endl;
```

Custom properties

Usually we would want to have some application-specific data attached to vertices and/or edges. A convenient way to accomplish this is via *bundled properties* mechanism. Just define a struct with all your custom fields, and pass it as a template parameter to adjacency_list. Then the values can be easily accessed:

```
struct VertexInfo { string name; };
typedef adjacency_list<vecS,vecS,
    undirectedS,VertexInfo> Graph;
...
V v = 0/* get vertex descriptor */;
g[v].name = "Vertex name";
```

One can also assign properties to edges and to the graph as a whole.

Extending algorithms with visitors
Several of the more generic BGL algorithms, such as BFS and DFS, can be easily extended in such a way that custom actions will be performed at predefined event points in the algorithm. So let us take DFS for example. There are several "event points" defined, such as when a vertex, a tree edge or a back edge is discovered. We will "plug into" the examine_edge event and assign each edge a unique number indicating the order of discovery by the DFS. One way to do this is to implement a class inheriting from default_dfs_visitor and override the appropriate method. However, a question arises: where to put the above unique number? The answer is provided by the idea of property maps: our visitor will be parametrized by a type which should implement a property map interface, and it will receive the map object in the constructor. We will assume that the property value for every edge is initialized to -1. [Note that get() and put() methods are part of the the property map ``interface".]

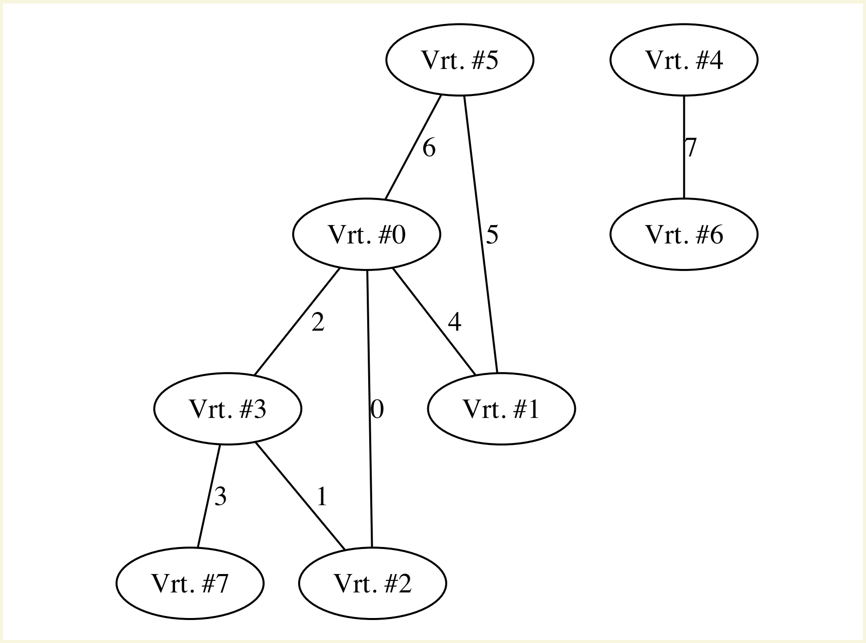
```
template<class OrderMap>
class dfs_edge_recorder : public
    default_dfs_visitor {
public:
    dfs_edge_recorder(OrderMap m) :
        m_order(0), m_map(m) {}

    template <class E, class G>
    void examine_edge(E e, G g)
    {
        if (get(m_map,e) == -1)
         put(m_map, e, m_order++);
    }

private:
    int m_order;
    OrderMap m_map;
};
```

One possibility is to store the number in a bundled edge property. So we define

Figure 1: All the edges are labeled in the order they are discovered by the DFS. The drawing was produced with GraphViz library.



```
struct EdgeInfo { int order; };
typedef adjacency_list<vecS, vecS,
        undirectedS,VertexInfo,
        EdgeInfo> Graph;
...
```

First we need to initialize the order of each edge to -1.

```
typedef property_map<Graph, int
        EdgeInfo::*>::type
        EdgeOrderMap;
EdgeOrderMap order_map =get (&
        EdgeInfo::order, g);
graph_traits<Graph>::
        edge_iterator ei, ei_end;
for (tie(ei, ei_end) = edges(g);
        ei != ei_end; ++ei)
    put(order_map, *ei, -1);
```

Finally, let us run the DFS.

```
dfs_edge_recorder<EdgeOrderMap>
        edge_recorder(order_map);
depth_first_search(g, visitor(
        edge_recorder));
```

As you can see, the decoupling of algorithms from storage is very powerful concept. Our dfs_edge_recorder is completely generic and can be used with any graph type and any property map indexed by edge descriptor type.

Conclusion
The BGL design takes some time to learn. Also, the C++ template mechanism and syntax might be disliked by some people. Despite all these, the BGL is ever-growing and constantly improving repository of reusable graph algorithms, and so it is definitely worth the effort. The documentation is very good, together with lots of examples.

The source code for this tutorial and installation instructions can be found at: [http://xrds.acm.org/code.cfm].

LINKS

Boost Graph Library Documentation
http://www.boost.org/doc/libs/1_45_0/libs/graph/doc/table_of_contents.html

MATLAB BGL
http://www.stanford.edu/~dgleich/programs/matlab_bgl/

BGL Tutorial
http://www.informit.com/articles/article.aspx?p=25756