Exploring the Boost Graph Library

Arpan Sen

December 03, 2012

Get to know the Boost Graph Library, a generic graph library that helps C++ developers convert practical engineering problems into graph-theory problems.

Making axiomatic statements about computing is usually a matter of hot debate. However, graph theory being one of the most important theoretical pillars of modern computing is not one of those statements. Myriad fields of engineering—from designing routers and networks to designing the chips that form the core of your mobile—are but applications of graph theory.

As C++ application-software developers, we often have a direct need to transform a practical engineering problem into an equivalent graph-theory problem. Having a robust C++-based generic-graph library that helps us do just that would obviously be welcome: The Boost Graph Library (BGL) fills in that precise void.

In this article, you create an undirected, and then a directed graph followed by the usual traversal routines. Then, you apply some classical algorithms—all without adding a lot of code. That is the magic of BGL.

Download and installation

BGL is available as a free download from the Boost site (see Related topics for a link). BGL is a header-only library, therefore, subsequent use of this library in application code requires including the relevant headers in your source code. However, BGL does need the serialization library to link. Here is a typical command line:

```bash
g++ test_bgl.cpp /usr/boost/boost_1_44/ -L/usr/boost/boost_1_44/lib
```

For the purposes of this article, you need the Boost 1.44 release.

Adjacency lists

At the heart of any graph implementation lies an adjacency list or matrix. Listing 1 shows how the adjacency list is declared in the Boost header adjacency_list.hpp.

```
```
Listing 1. Declaring an adjacency list in Boost

```cpp
template <class OutEdgeListS = vecS,
    // a Sequence or an AssociativeContainer class VertexListS = vecS,
    // a Sequence or a RandomAccessContainer class DirectedS = directedS,
    class VertexProperty = no_property,
    class EdgeProperty = no_property,
    class GraphProperty = no_property,
    class EdgeListS = listS>
    class adjacency_list {  }
```

To keep matters simple, let us concentrate on the first three template parameters.

The `OutEdgeList` template parameter decides what kind of container will be used to store the edge-list information. Recall from your graph-theory basics that, for a directed graph, all vertices that have only incoming edges have an empty corresponding adjacency list. The default value is set to `vecS`, which corresponds to `std::vector`. The `VertexList` template parameter decides what kind of container is used to represent the vertex list of the graph, again the default being `std::vector`. The `DirectedS` template parameter decides whether this is a directed or undirected graph based on whether the value provided is `directedS` or `undirectedS`.

Creating a graph in BGL

With that declaration of the adjacency list in tow, the code in Listing 2 creates a simple undirected graph in BGL. The edge information will be stored in `std::list` and the vertices in `std::vector`.

Listing 2. Creating an undirected graph

```cpp
#include <boost/graph/adjacency_list.hpp>
using namespace boost;
typedef boost::adjacency_list<listS, vecS, undirectedS> mygraph;
int main()
{
    mygraph g;
    add_edge (0, 1, g);
    add_edge (0, 3, g);
    add_edge (1, 2, g);
    add_edge (2, 3, g);
}
```

In Listing 2, the graph `g` is created without providing any vertex or edge details in the constructor. Edges and vertices would be created at run time using helper functions like `add_edge` and `add_vertex`. The `add_edge` function does what its name implies: It adds an edge between two vertices of the graph. At the end of execution in Listing 2, `g` should have four vertices labeled 0, 1, 2, and 3, with 1 connected to 0 and 2, and so on.

Traversing the graph

Traversing the graph involves using the `vertex_iterator` and the `adjacency_iterator` classes. The former iterates over all the vertices of the graph; the latter iterates over the corresponding adjacent vertices. Listing 3 shows the code.

Listing 3. Traversing the graph using BGL

```cpp
#include <boost/graph/adjacency_list.hpp>
```
using namespace boost;
typedef boost::adjacency_list<listS, vecS, undirectedS> mygraph;
int main()
{
  mygraph g;
  add_edge (0, 1, g);
  add_edge (0, 3, g);
  add_edge (1, 2, g);
  add_edge (2, 3, g);
  mygraph::vertex_iterator vertexIt, vertexEnd;
  mygraph::adjacency_iterator neighbourIt, neighbourEnd;
  tie(vertexIt, vertexEnd) = vertices(g);
  for (; vertexIt != vertexEnd; ++vertexIt)
  {
    cout << *vertexIt << " is connected with ";
    tie(neighbourIt, neighbourEnd) = adjacent_vertices(*vertexIt, g);
    for (; neighbourIt != neighbourEnd; ++neighbourIt)
      cout << *neighbourIt << " ";
    cout << \n;  
  }
}

Creating a directed graph
To create a directed graph, simply modify the graph type in Listing 3 to directedS:

#include <boost/graph/adjacency_list.hpp> using namespace boost;
typedef boost::adjacency_list<listS, vecS, directedS> mygraph;
int main() {
  mygraph g;
  add_edge (0, 1, g);
  add_edge (0, 3, g);
  add_edge (1, 2, g);
  add_edge (2, 3, g);
  // Same as Listing 3}

The helper function vertices returns a std::pair<vertex_iterator and vertex_iterator>, with the former pointing to the first vertex in the graph. The result is captured in the tuple tie (vertexIt, vertexEnd), and vertexIt is subsequently used to traverse the graph. Likewise, the helper function adjacent_vertices returns std::pair<adjacency_iterator, adjacency_iterator>, with the first adjacency_iterator pointing to the first element in the adjacency list.

Configuring an adjacency list
One of the cool things about BGL is that it is highly configurable. BGL lets you configure the set of vertices and the collection of edges using any of the following selector types (these are all defined in the header; all you need to do is use them while declaring the graph):

- vecS selects std::vector
- lists for std::list
- slistS selects std::slist
- setS selects std::set
- multiSetS selects std::multiset
- hash_setS selects std::hash_set

If your code would have many vertex insertions but not much deletion, then the VertexList could be vecS or lists. push_back is usually constant time except for when a vector needs reallocation. If you will have many insertions and deletions, then a lists is a better choice than vecs, because deleting an element from a vector is usually more expensive, time wise.
Another way to create an undirected graph

Instead of using the adjacency list-based version to create an undirected graph, you can use the BGL-provided `undirected_graph` class (defined in undirected_graph.hpp). However, this class internally uses an adjacency list, and using graphs based on adjacency lists always provides for greater flexibility. Listing 4 shows the code.

Listing 4. Creating an undirected graph using undirected_graph.hpp

```cpp
#include <undirected_graph.hpp> #include <iostream> using namespace boost;
using namespace std;

int main()
{
    undirected_graph<> g;
    undirected_graph<>::vertex_descriptor u = g.add_vertex();
    undirected_graph<>::vertex_descriptor v = g.add_vertex();
    undirected_graph<>::vertex_descriptor w = g.add_vertex();
    undirected_graph<>::vertex_descriptor x = g.add_vertex();
    add_edge(u, v, g); add_edge(u, w, g); add_edge(u, x, g);
    cout << "Degree of u: " << degree(u, g);
    return 0;
}
```

In Listing 4, I added individual vertices to a graph using `add_vertex` and edges using `add_edge`. Finally, the degree of an individual vertex is displayed using the `degree` method call. Listing 5 provides the declaration and definition of `undirected_graph` (from Boost sources).

Listing 5. Deciphering a BGL undirected_graph

```cpp
template < typename VertexProp = no_property,
            typename EdgeProp = no_property,
            typename GraphProp = no_property> class undirected_graph
{ // public:
    typedef adjacency_list<listS,
                           listS, undirectedS,
                           vertex_property,
                           edge_property,
                           GraphProp,
                           listS> graph_type;
    private: graph_type m_graph; //
};
```

Tracking the in-edges and out-edges of a vertex

You access the outgoing edges of a vertex using the `out_edges` helper function; to access the incoming edges, use `in_edges`. A great thing about BGL is that you can directly output an edge using `cout`, and it shows the connecting vertices. Listing 6 shows the code.

Listing 6. Iterating over the edges of a directed graph

```cpp
#include <boost/graph/adjacency_list.hpp> using namespace boost;
typedef boost::adjacency_list<listS, vecS, directedS> mygraph;
int main()
{
    mygraph g;
    add_edge (0, 1, g);
    add_edge (0, 3, g);
```
add_edge (1, 2, g);
add_edge (2, 3, g);
mygraph::vertex_iterator vertexIt, vertexEnd;
mygraph::in_edge_iterator inedgeIt, inedgeEnd;
mygraph::in_edge_iterator outedgeIt, outedgeEnd; tie(vertexIt, vertexEnd) = vertices(g);
for (; vertexIt != vertexEnd; ++vertexIt)
{
    cout << "incoming edges for " << *vertexIt << ": ";
    tie(inedgeIt, inedgeEnd) = in_edges(*vertexIt, g);
    for(; inedgeIt != inedgeEnd; ++inedgeIt)
    {
        cout << *inedgeIt << " ";
    }
    cout << "\n";
}
for (; vertexIt != vertexEnd; ++vertexIt)
{
    std::cout << "out-edges for " << *vertexIt << : 
    tie(outedgeIt, outedgeEnd) = out_edges(*vertexIt, g); // Similar to incoming edges
}

If you compiled the code in Listing 6, you would get errors. To fix the errors, just replace directedS
with bidirectionalS in the declaration of mygraph. When using the directedS tag in BGL, you
are allowed to use only the out_edges helper function and associated iterators. Using in_edges
requires changing the graph type to bidirectionalS, although this is still more or less a directed
graph.

**Note:** Using in_edges comes with an additional space cost (twice the per-edge cost if you
compare it with directedS), so be sure it is something you can live with.

### Some useful BGL functions

Now, let us take stock of some of the more important utility functions that BGL provides and that
have not been discussed, yet.

You can use the following functions for graph access:

- **std::pair<edge_iterator, edge_iterator> edges(const adjacency_list& g).** Returns an
  iterator pair corresponding to the edges in graph g
- **vertices_size_type num_vertices(const adjacency_list& g).** Returns the number of
  vertices in g
- **edges_size_type num_edges(const adjacency_list& g).** Returns the number of edges in g
- **vertex_descriptor source(edge_descriptor e, const adjacency_list& g).** Returns the
  source vertex of an edge
- **vertex_descriptor target(edge_descriptor e, const adjacency_list& g).** Returns the
  target vertex of an edge
- **degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g).** Returns
  the in-degree of a vertex
- **degree_size_type out_degree(vertex_descriptor u, const adjacency_list& g).** Returns
  the out-degree of a vertex

Listing 7 shows the code performing a fair number of graph accesses.
Listing 7. Graph accesses using BGL

```c++
// usual typedefs here, refer to previous listings
int main()
{
    mygraph g;
    add_edge (0, 1, 8, g);
    add_edge (0, 3, 18, g);
    add_edge (1, 2, 20, g);
    add_edge (2, 3, 2, g);
    add_edge (3, 1, 1, g);
    add_edge (1, 3, 7, g);
    cout << "Number of edges: " << num_edges(g) << "\n";
    cout << "Number of vertices: " << num_vertices(g) << "\n"
    mygraph::vertex_iterator vertexIt, vertexEnd; tie(vertexIt, vertexEnd) = vertices(g);
    for (; vertexIt != vertexEnd; ++vertexIt)
    {
        std::cout << "in-degree for " << *vertexIt << " : " << in_degree(*vertexIt, g) << "\n";
        std::cout << "out-degree for " << *vertexIt << " : " << out_degree(*vertexIt, g) << "\n"
    }
    mygraph::edge_iterator edgeIt, edgeEnd; tie(edgeIt, edgeEnd) = edges(g);
    for (; edgeIt != edgeEnd; ++edgeIt)
    {
        std::cout << "edge " << source(*edgeIt, g) << "-->
        target(*edgeIt, g) << "\n"
    }
}
```

You can use the following functions for graph modification:

- **void remove_edge(vertex_descriptor u, vertex_descriptor v, adjacency_list& g).** Removes an edge from g
- **void remove_edge(edge_descriptor e, adjacency_list& g).** Removes an edge from g
- **void clear_vertex(vertex_descriptor u, adjacency_list& g).** Removes all edges to and from u
- **void clear_out_edges(vertex_descriptor u, adjacency_list& g).** Removes all outgoing edges from vertex u in the directed graph g (not applicable for undirected graphs)
- **void clear_in_edges(vertex_descriptor u, adjacency_list& g).** Removes all incoming edges from vertex u in the directed graph g (not applicable for undirected graphs)
- **void remove_vertex(vertex_descriptor u, adjacency_list& g).** Removes a vertex from graph g (It is expected that all edges associated with this vertex have already been removed using clear_vertex or another appropriate function.)

### Creating a directed weighted graph using BGL

Now that you are comfortable with directed graphs, the next logical task is to create a weighted directed graph with BGL. If you look back to the declaration of the adjacency list in **Listing 1**, you see that one of the template parameters is called EdgeProperty. It is this template parameter that you use to construct the graph.

A property is a parameter that can be assigned to both vertices and edges. You define a property using a tag name and a type corresponding to the property. BGL has several available tag names, including edge_weight_t and vertex_name_t. For example, to store names in graph vertices, you
can define a property as `typedef property<vertex_name_t, std::string> VertexNameProperty`, and then pass this property to the `VertexProperty` parameter in the template declaration for the graph.

Here is the declaration for a property for edge weights:

```cpp
typedef property<edge_weight_t, int> EdgeWeightProperty;
```

Now that you have created `EdgeWeightProperty`, tweak your definition of `mygraph`:

```cpp
typedef boost::adjacency_list<listS, vecS, directedS, no_property, EdgeWeightProperty> mygraph;
```

Finally, when you add the edges in the graph, use the overloaded version of `add_edge`, which accepts a weight as the third argument. **Listing 8** provides the complete code.

### Listing 8. Creating a weighted directed graph

```cpp
#include <boost/graph/adjacency_list.hpp>
using namespace boost;
typedef property<edge_weight_t, int> EdgeWeightProperty;
typedef boost::adjacency_list<listS, vecS, directedS, no_property, EdgeWeightProperty> mygraph;
int main()
{
    mygraph g;
    add_edge (0, 1, 8, g);
    add_edge (0, 3, 18, g);
    add_edge (1, 2, 20, g);
    add_edge (2, 3, 2, g);
    add_edge (3, 1, 1, g);
    add_edge (1, 3, 7, g);
}
```

A minimum spanning tree in BGL

One of the best things about BGL is that it comes with a whole host of predefined algorithms that work on graphs. Kruskal, Prim, Dijkstra—you name it and BGL has it. To see what I mean, modify the code in **Listing 8** to have an undirected graph with weighted edges, and then use Kruskal's algorithm on it to get the minimum spanning tree. BGL provides individual algorithms in separate headers, so to use Kruskal, you must include `boost/graph/kruskal_min_spanning_tree.hpp`. **Listing 9** shows the code.

### Listing 9. Minimum spanning tree using Kruskal's algorithm

```cpp
#include <boost/graph/adjacency_list.hpp> // typedef boost::adjacency_list<listS, vecS, directedS, no_property, EdgeWeightProperty> mygraph;
typedef mygraph::edge_descriptor Edge;
int main()
{
    mygraph g;
    add_edge (0, 1, 8, g);
    add_edge (0, 3, 18, g);
    add_edge (1, 2, 20, g);
    add_edge (2, 3, 2, g);
    add_edge (3, 1, 1, g);
    add_edge (1, 3, 7, g);
```
add_edge (2, 3, 2, g);
add_edge (3, 1, 1, g);
add_edge (1, 3, 7, g);
std::list < Edge > spanning_tree;
kruskal_minimum_spanning_tree (g, std::back_inserter(spanning_tree));
for (std::list < Edge >::iterator ei = spanning_tree.begin(); ei != spanning_tree.end(); ++ei)
{
    cout << *ei << " ";
}
cout << "\n";
}

The function kruskal_minimum_spanning_tree does the magic behind the scenes. It takes in the graph and an iterator to a container where the edges are stored. Note the declaration of spanning_tree: I used a list here, but it can be anything—for example, a vector. All BGL cares for is that the second argument must be the Standard Template Library (STL) output-iterator type.

**Depth-first search using BGL**

Breadth-first search and depth-first search (DFS) form the crux of graph traversals, and BGL comes with a lot of support for these operations. The relevant header file to be included is boost/graph/depth_first_search.hpp; the corresponding routine is depth_first_search. BGL provides for multiple interfaces to depth_first_search; all of them need what is known as a visitor object to be passed to the method.

A visitor in BGL plays the role of a functor in STL except that it does a lot more. A visitor does not have a single method like operator () but instead has the flexibility to define several methods, such as initialize_index, start_index, discover_index, and examine_edge. It is safe to say that BGL lets you customize the DFS by providing these hooks. Let us first look into a sample code using DFS (Listing 10).

**Listing 10. DFS using BGL**

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <iostream>
using namespace std;
using namespace boost;
typedef property<edge_weight_t, int> EdgeWeightProperty;
typedef boost::adjacency_list
    < listS, vecS, undirectedS, no_property, EdgeWeightProperty>
    mygraph;

class custom_dfs_visitor : public boost::default_dfs_visitor
{
    public: template < typename Vertex, typename Graph >
    void discover_vertex(Vertex u, const Graph & g)
    const { std::cout << "At " << u << std::endl; }
    template < typename Edge, typename Graph >
    void examine_edge(Edge e, const Graph& g)
    const { std::cout << "Examining edges " << e << std::endl; }
};

int main()
{
    mygraph g; add_edge (0, 1, 8, g);
    add_edge (0, 3, 18, g);
    ```
add_edge (1, 2, 20, g);
add_edge (2, 3, 2, g);
add_edge (3, 1, 1, g);
add_edge (1, 3, 7, g);
custom_dfs_visitor vis;
depth_first_search(g, visitor(vis));
}

Listing 10 declares a class called custom_dfs_visitor that defines two hooks: discover_vertex and examine_edges. The former is invoked when a vertex is encountered for the first time; the latter is invoked on every outgoing edge from the vertex after the vertex is discovered.

So, what would have happened if in the visitor (vis) you had vis be of type boost_default_visitor? Yes, you guessed it: nothing on the display. Table 1 provides a quick summary of some of the hooks BGL provides.

Table 1. BGL hooks for traversing using DFS

<table>
<thead>
<tr>
<th>Hook</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_vertex(u, g)</td>
<td>Invoked on the source vertex before traversal begins</td>
</tr>
<tr>
<td>discover_vertex(u, g)</td>
<td>Invoked when a vertex is invoked for the first time</td>
</tr>
<tr>
<td>finish_vertex(u, g)</td>
<td>If u is the root of a tree, finish_vertex is invoked after the same is invoked on all other elements of the tree. If u is a leaf, then this method is invoked after all outgoing edges from u have been examined.</td>
</tr>
<tr>
<td>examine_edge(u, g)</td>
<td>Invoked on every outgoing edge of u after it is discovered</td>
</tr>
<tr>
<td>tree_edge(u, g)</td>
<td>Invoked on an edge after it becomes a member of the edges that form the search tree</td>
</tr>
<tr>
<td>back_edge(u, g)</td>
<td>Invoked on the back edges of a graph; used for an undirected graph, and because (u, v) and (v, u) are the same edges, both tree_edge and back_edge are invoked</td>
</tr>
</tbody>
</table>

Note that BGL supports other visitors, like dijkstra_visitor, bellman_visitor, and astar_visitor.

Conclusion

That is it for this article. You have learned how to create undirected, directed, and weighted graphs in BGL; played with the accessors and modifier functions; and tried your hand at implementing some classical graph algorithms on the graphs you created. BGL offers a lot more, and this article has just skimmed the surface. Be sure to look into the BGL documentation for details.
Related topics

- Check out the BGL documentation.
- The Boost Graph Library: User Guide and Reference Manual by Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine is a fantastic resource for the BGL.
- Download the BGL.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)

Explored by: Boost Graph Library