CS302 - Data Structures

using C++

Topic: The Boost Graph Library

Kostas Alexis

AUTONOMOUS ROBOTS LAB

Boost Graph Library (BGL)

- The BGL graph interface and graph components are generic, in the same sense as the Standard Template Library (STL).

How to Build the BGL

- DON'T! The Boost Graph Library is a header-only library and does not need to be built to be used. The only exceptions are the GraphViz input parser and the GraphML parser.

Genericity in STL

- There are three ways in which the STL is generic.
- **Algorithm/Data-Structure Interoperability**
  - First, each algorithm is written in a data-structure neutral way, allowing a single template function to operate on many different classes of containers. The concept of an iterator is the key ingredient in this decoupling of algorithms and data-structures. The impact of this technique is a reduction in the STL's code size from O(M*N) to O(M+N), where M is the number of algorithms and N is the number of containers.

Genericity in STL

- There are three ways in which the STL is generic.
- **Extension through Function Objects**
  - Its algorithms and containers are extensible. The user can adapt and customize the STL through the use of function objects.

Genericity in STL

- There are three ways in which the STL is generic.
- **Element Type Parameterization**
  - The third way that STL is generic is that its containers are parameterized on the element type. Though hugely important, this is perhaps the least “interesting” way in which STL is generic. Generic programming is often summarized by a brief description of parameterized lists such as std::list<T>.

Genericity in the BGL

- There are three ways in which the BGL is generic.
- **Algorithm/Data-Structure Interoperability**
  - First, the graph algorithms of the BGL are written to an interface that abstracts away the details of the particular graph data-structure. Like the STL, the BGL uses iterators to define the interface for data-structure traversal. There are three distinct graph traversal patterns: traversal of all vertices in the graph, through all of the edges, and along the adjacency structure of the graph.
  - This generic interface allows template functions such as breadth_first_search() to work on a large variety of graph data-structures, from graphs implemented with pointer-linked nodes to graphs encoded in arrays. This flexibility is especially important in the domain of graphs.
  - In contrast, custom-made (or even legacy) graph structures can be used as-is with the generic graph algorithms of the BGL, using external adaptation.

Genericity in the BGL

- There are three ways in which the BGL is generic.
- **Extension through Visitors**
  - The graph algorithms of the BGL are extensible. The BGL introduces the notion of a visitor, which is just a function object with multiple methods. In graph algorithms, there are often several key "event points" at which it is useful to insert user-defined operations. The visitor object has a different method that is invoked at each event point. The particular event points and corresponding visitor methods depend on the particular algorithm. They often include methods like start_vertex(), discover_vertex(), examine_edge(), tree_edge(), and finish_vertex().

Genericity in the BGL

- There are three ways in which the BGL is generic.
- **Vertex and Edge Property Multi-Parametrization**
  - The third way that the BGL is generic is analogous to the parameterization of the element-type in STL containers. We need to associate values (called “properties”) with both the vertices and the edges of the graph. In addition, it will often be necessary to associate multiple properties with each vertex and edge; this is what we mean by multi-parameterization.
  - The STL std::list<T> class has a parameter T for its element type. Similarly, BGL graph classes have template parameters for vertex and edge “properties”. A property specifies the parameterized type of the property and also assigns an identifying tag to the property. This tag is used to distinguish between the multiple properties which an edge or vertex may have. A property value that is attached to a particular vertex or edge can be obtained via a property map. There is a separate property map for each property.

Algorithms in BGL

- The BGL algorithms consist of a core set of algorithm patterns (implemented as generic algorithms) and a larger set of graph algorithms.
- The core algorithm patterns are
  - **Breadth First Search**
  - **Depth First Search**
  - **Uniform Cost Search**

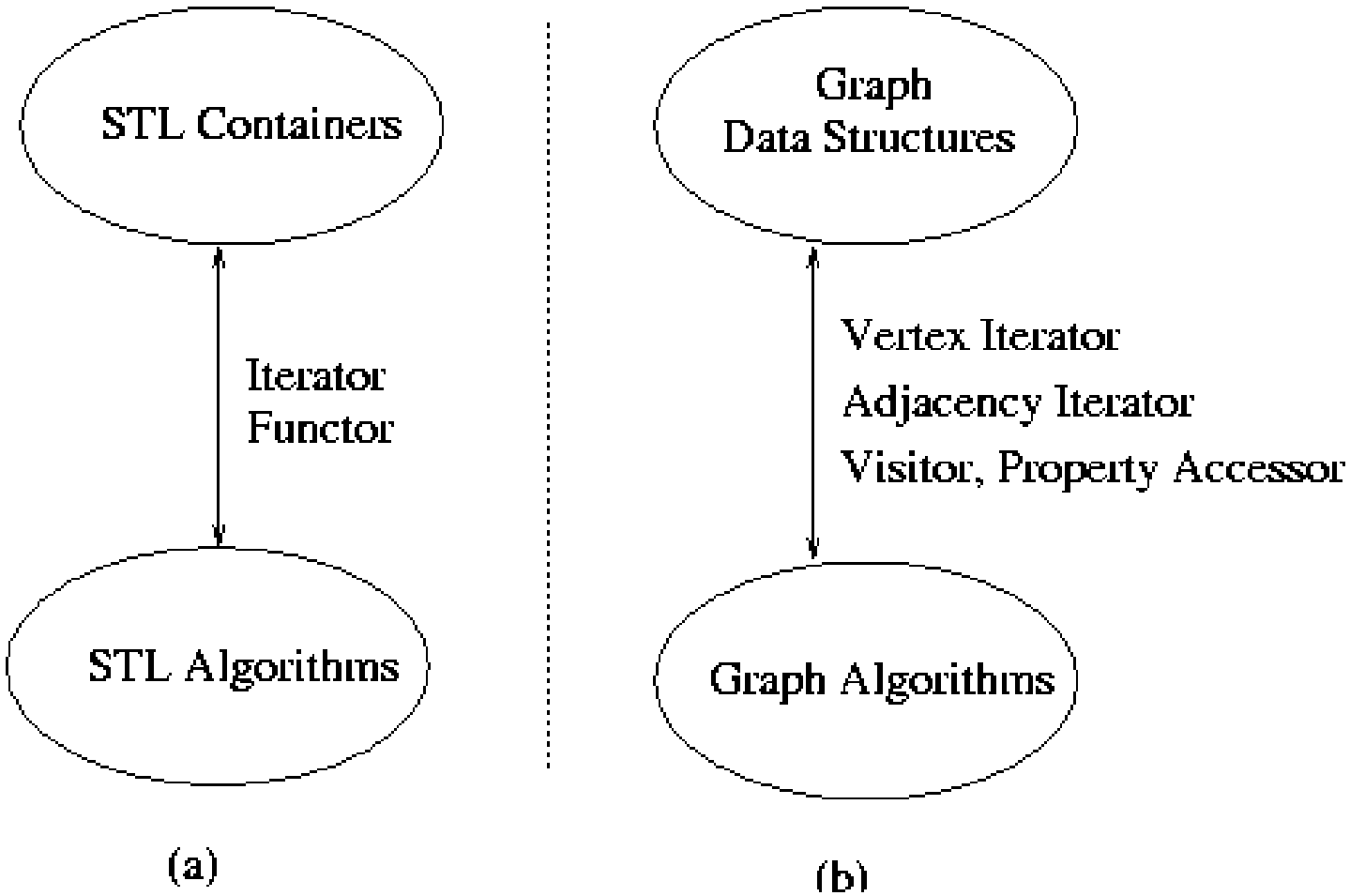
Algorithms in BGL

- By themselves, the algorithm patterns do not compute any meaningful quantities over graphs; they are merely building blocks for constructing graph algorithms. The graph algorithms in the BGL currently include
  - **Dijkstra's Shortest Paths**
  - **Bellman-Ford Shortest Paths**
  - **Johnson's All-Pairs Shortest Paths**
  - **Kruskal's Minimum Spanning Tree**
  - **Prim's Minimum Spanning Tree**
  - **Connected Component**
  - **Strongly Connected Components**
  - **Dynamic Connected Components**
  - **Topological Sort**
  - **Reverse Cuthill Mckee Ordering**
  - **Smallest Last Vertex Ordering**
  - **Sequential Vertex Coloring**

Data Structures in BGL

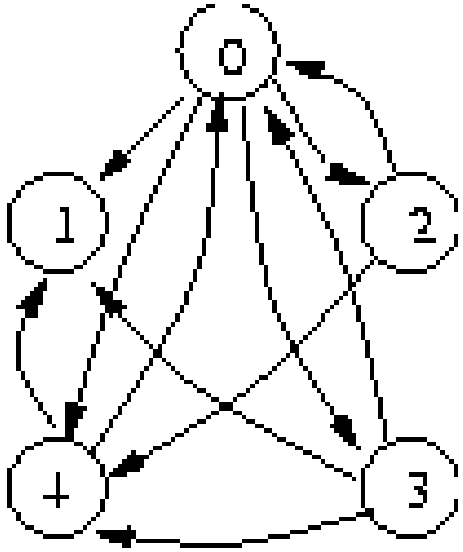
- The BGL currently provides two graph classes and an edge list adaptor:
  - **adjacency_list**
  - **adjacency_matrix**
  - **edge_list**

Analogy between STL and BGL



Graph Abstraction

- The graph abstraction consists of a set of vertices (or nodes), and a set of edges (or arcs) that connect the vertices.
- The Figure depicts a directed graph with five vertices (labeled 0 through 4) and 11 edges. The edges leaving a vertex are called the out-edges of the vertex. The edges {(0,1),(0,2),(0,3),(0,4)} are all out-edges of vertex 0. The edges entering a vertex are called the in-edges of the vertex. The edges {(0,4),(2,4),(3,4)} are all in-edges of vertex 4.



Constructing a Graph

```
#include <iostream> // for std::cout
#include <utility> // for std::pair
#include <algorithm> // for std::for_each
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace boost;
int main(int,char*[])
{
          // create a typedef for the Graph type
          typedef adjacency_list<vecS, vecS, bidirectionalS> Graph;

          // Make convenient labels for the vertices
          enum { A, B, C, D, E, N };
          const int num_vertices = N;
          const char* name = "ABCDE";

          // writing out the edges in the graph
          typedef std::pair<int, int> Edge;
          Edge edge_array[] =
          { Edge(A,B), Edge(A,D), Edge(C,A), Edge(D,C),
          Edge(C,E), Edge(B,D), Edge(D,E) };
          const int num_edges = sizeof(edge_array)/sizeof(edge_array[0]);

          // declare a graph object
          Graph g(num_vertices);

          // add the edges to the graph object
          for (int i = 0; i < num_edges; ++i)
                    add_edge(edge_array[i].first, edge_array[i].second, g);
          ...
          return 0;
}
```

- The adjacency_list class provides a generalized version of the classic "adjacency list" data structure.
- The adjacency_list is a template class with six template parameters, though here we only fill in the first three parameters and use the defaults for the remaining three.
- The first two template arguments (vecS, vecS) determine the data structure used to represent the out-edges for each vertex in the graph and the data structure used to represent the graph's vertex set
- The third argument, bidirectionalS, selects a directed graph that provides access to both out and in-edges.
- The other options for the third argument are directedS which selects a directed graph with only out-edges, and undirectedS which selects an undirected graph.

Constructing a Graph

```
#include <iostream> // for std::cout
#include <utility> // for std::pair
#include <algorithm> // for std::for_each
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

using namespace boost;
int main(int,char*[])
{
          // create a typedef for the Graph type
          typedef adjacency_list<vecS, vecS, bidirectionalS> Graph;

          // Make convenient labels for the vertices
          enum { A, B, C, D, E, N };
          const int num_vertices = N;
          const char* name = "ABCDE";

          // writing out the edges in the graph
          typedef std::pair<int, int> Edge;
          Edge edge_array[] =
          { Edge(A,B), Edge(A,D), Edge(C,A), Edge(D,C),
          Edge(C,E), Edge(B,D), Edge(D,E) };
          const int num_edges = sizeof(edge_array)/sizeof(edge_array[0]);

          // declare a graph object
          Graph g(num_vertices);

          // add the edges to the graph object
          for (int i = 0; i < num_edges; ++i)
                    add_edge(edge_array[i].first, edge_array[i].second, g);
          ...
          return 0;
}
```

- Instead of calling the add_edge() function for each edge, we could use the edge iterator constructor of the graph. This is typically more efficient than using add_edge(). Pointers to the edge_array can be viewed as iterators, so we can call the iterator constructor by passing pointers to the beginning and end of the array.
- Instead of creating a graph with a certain number of vertices to begin with, it is also possible to add and remove vertices with the add_vertex() and remove_vertex() functions, also of the MutableGraph interface.

Accessing the Vertex Set

```
int main(int,char*[])
{
        // ...
        // get the property map for vertex indices
        typedef property_map<Graph, vertex_index_t>::type IndexMap;
        IndexMap index = get(vertex_index, g);

        std::cout << "vertices(g) = ";
        typedef graph_traits<Graph>::vertex_iterator vertex_iter;
        std::pair<vertex_iter, vertex_iter> vp;
        for (vp = vertices(g); vp.first != vp.second; ++vp.first)
        std::cout << index[*vp.first] << " ";
        std::cout << std::endl;
        // ...
        return 0;
}
```

The output is: vertices(g) = 0 1 2 3 4

- Now that we have created a graph, we can use the graph interface to access the graph data in different ways.
- First we can access all of the vertices in the graph using the vertices() function of the VertexListGraph interface.
- This function returns a std::pair of vertex iterators (the first iterator points to the "beginning" of the vertices and the second iterator points "past the end").
- Dereferencing a vertex iterator gives a vertex object.
- The type of the vertex iterator is given by the graph_traits class.
- Note that different graph classes can have different associated vertex iterator types, which is why we need the graph_traits class. Given some graph type, the graph_traits class will provide access to the vertex_iterator type.

Accessing the Edge Set

```
int main(int,char*[])
{
        // ...
        std::cout << "edges(g) = ";
        graph_traits<Graph>::edge_iterator ei, ei_end;
        for (tie(ei, ei_end) = edges(g); ei != ei_end; ++ei)
                std::cout << "(" << index[source(*ei, g)]
                << "," << index[target(*ei, g)] << ") ";
        std::cout << std::endl;
        // ...
        return 0;
}
```

- The set of edges for a graph can be accessed with the edges() function of the EdgeListGraph interface.
- Similar to the vertices() function, this returns a pair of iterators, but in this case the iterators are edge iterators.
- Dereferencing an edge iterator gives an edge object.
- The source() and target() functions return the two vertices that are connected by the edge.
- Instead of explicitly creating a std::pair for the iterators, this time we will use the tie() helper function.
- This handy function can be used to assign the parts of a std::pair into two separate variables, in this case ei and ei_end. This is usually more convenient than creating a std::pair and is our method of choice for the BGL.

The output is: edges(g) = (0,1) (0,3) (2,0) (3,2) (2,4) (1,3) (3,4)

The Adjacency Structure

```
int main(int,char*[])
{
        //...
        std::for_each(vertices(g).first, vertices(g).second,
        exercise_vertex<Graph>(g));
        return 0;
}
```

- In the next few examples we will explore the adjacency structure of the graph from the point of view of a particular vertex.
- We will look at the vertices' in-edges, out-edges, and its adjacent vertices.
- We will encapsulate this in an "exercise vertex" function, and apply it to each vertex in the graph.
- To demonstrate the STL-interoperability of BGL, we will use the STL for_each() function to iterate through the vertices and apply the function.

The Adjacency Structure

```
template <class Graph> struct exercise_vertex {
          exercise_vertex(Graph& g_) : g(g_) {}
          //...
          Graph& g;
};
```

- We use a functor for exercise_vertex instead of just a function because the graph object will be needed when we access information about each vertex;
- Using a functor gives us a place to keep a reference to the graph object during the execution of the std::for_each().
- Also we template the functor on the graph type so that it is reusable with different graph classes. Here is the start of the exercise_vertex functor:

Vertex Descriptors

```
template <class Graph> struct exercise_vertex {
          //...
          typedef typename graph_traits<Graph>
                    ::vertex_descriptor Vertex;

          void operator()(const Vertex& v) const
          {
                    //...
          }
          //...
};
```

- The first thing we need to know in order to write the operator() method of the functor is the type for the vertex objects of the graph.
- The vertex type will be the parameter to the operator() method. To be precise, we do not deal with actual vertex objects, but rather with vertex descriptors.
- Many graph representations (such as adjacency lists) do not store actual vertex objects, while others do (e.g., pointer-linked graphs).
- The vertex descriptor is something provided by each graph type that can be used to access information about the graph via the out_edges(), in_edges(), adjacent_vertices(), and property map functions that are described in the following sections. The vertex_descriptor type is obtained through the graph_traits class.

Out-Edges

```
template <class Graph> struct exercise_vertex {
          //...
          void operator()(const Vertex& v) const
          {
                    typedef graph_traits<Graph> GraphTraits;
                    typename property_map<Graph, vertex_index_t>::type
                    index = get(vertex_index, g);

                    std::cout << "out-edges: ";
                    typename GraphTraits::out_edge_iterator out_i, out_end;
                    typename GraphTraits::edge_descriptor e;
                    for (tie(out_i, out_end) = out_edges(v, g);
                              out_i != out_end; ++out_i) {
                              e = *out_i;
                              Vertex src = source(e, g), targ = target(e, g);
                              std::cout << "(" << index[src] << ","
                              << index[targ] << ") ";
                    }
                    std::cout << std::endl;
                    //...
          }
          //...
};
```

- The out-edges of a vertex are accessed with the out_edges() function of the IncidenceGraph interface.
- The out_edges() function takes two arguments:
  - the first argument is the vertex and
  - the second is the graph object.
- The function returns a pair of iterators which provide access to all of the out-edges of a vertex (similar to how the vertices() function returned a pair of iterators).
- The iterators are called out-edge iterators and dereferencing one of these iterators gives an edge descriptor object.
- An edge descriptor plays the same kind of role as the vertex descriptor object, it is a "black box" provided by the graph type.
- The code snippet prints the source-target pairs for each out-edge of vertex v.

For vertex 0 the output is: out-edges: (0,1) (0,2) (0,3) (0,4)

Thank you