Dr. Dobb's Software Tools for the Professional Programmer

The Generic Graph Component Library

Dr. Dobb's Journal September 2000

Generic programming for graph algorithms

By Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine

The authors are members of the scientific computing lab at the University of Notre Dame. They can be contacted at jsiek@lsc.nd.edu, llee1@ lsc.nd.edu, and lumsg@lsc.nd.edu, respectively.

The Standard Template Library has established a solid foundation for the development of reusable algorithms and data structures in C++. It has provided programmers with a way to think about designing reusable components (generic programming), and has demonstrated the programming techniques necessary to build efficient implementations. However, there are many problem domains beyond those addressed by the STL; consequently, there are many opportunities for applying generic programming. One particularly important domain is that of graph algorithms and data structures. The graph abstraction is widely used to model structures and relationships in many fields. Graph algorithms are extremely important in such diverse application areas as design automation, transportation, optimization, and databases. Our own interest in graph algorithms originates with our work on sparse matrix ordering algorithms for scientific computing.

The domain of graph algorithms is ripe for the application of generic programming. There is a large existing body of useful algorithms, yet the number of ways that people use to represent graphs in memory almost matches the number of applications that use graphs. The ability to freely interchange graph algorithms with graph representations would be an important contribution to the field, and this is what generic programming has to offer.

In January, 1999, we did a survey of existing graph libraries. Some of the libraries we looked at were LEDA (by Kurt Mehlhorn and Stefan Naeher, http:// www.mpi-sb.mpg.de/LEDA/leda.html), the Graph Template Library (GTL) (by Michael Forster, Andreas Pick, and Marcus Raitner, http://www.fmi.uni-passau.de/ Graphlet/GTL/), Combinatorica (see *Implementing Discrete Mathematics*, by Steven Skiena, Addison-Wesley, 1990), and Stanford GraphBase (see *Stanford GraphBase: A Platform for Combinatorial Computing*, by Donald E. Knuth, ACM Press, 1994). We also looked at software repositories such as Netlib (http://www .netlib.org/), which include graph algorithms. These libraries and repositories represent a significant amount of potentially reusable algorithms and data structures. However, none of the libraries applied the principles of generic programming and consequently did not receive the associated benefits of flexibility and efficiency. Therefore, we began construction of our own graph library, the Generic Graph Component Library (GGCL) (see "Generic Graph Algorithms for Sparse Matrix Ordering," ISCOPE '99, and "The Generic Graph Component Library," OOPSLA '99, both by Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine), drawing on previous experience with the development of the Matrix Template Library (see "The Matrix Template Library: A Generic Programming Approach to High-Performance Numerical

Linear Algebra," International Symposium on Computing in Object-Oriented Parallel Environments, 1998; and "Modern Software Tools in Scientific Computing," *A Modern Framework for Portable High-Performance Numerical Linear Algebra*, Birkhauser, 1999, both by Jeremy G. Siek and Andrew Lumsdaine). The most important aspect of designing the library was to define an interface that allowed for maximum flexibility. This is a list of our initial requirements:

- Provide a collection of useful generic graph algorithms.
- Enable almost any underlying graph data structure to be used with the generic graph algorithms.
- Provide a single generic interface to access graph vertex and edge properties, while at the same time not prescribing how the properties are stored in memory.
- Provide a mechanism for user-defined extensions to graph algorithm functionality (that is, the way functors make STL algorithms extensible).
- Define a syntax and naming scheme that is a close match to existing libraries and to notations used in graph algorithm literature.

The Graph Traversal Interface

When designing a generic library, one starts by analyzing how the data structures are used in the algorithms. Listings One, Two, Three, and Four show pseudocode for simple algorithms that demonstrate different ways graph structures are typically accessed. (The initialization step has been omitted from the algorithms. Also, the depth-first search algorithm [Listing One] does not show the operations that would be invoked to do some useful task during the search.) The access patterns can be categorized as follows:

- Traverse all of the vertices in a graph.
- Traverse all of the edges in a graph.
- For some vertex, visit the out-edges.
- For some vertex, visit the adjacent vertices. (Accessing adjacent vertices is really the same access pattern as for out-edges. The adjacent vertex is just the target vertex of the out-edge.)
- Access the edge (if it exists) connecting vertex *u* and *v*.

The pseudocode from the graph algorithms and the derived categories of graph traversal provide a reasonable basis for defining the actual interface to be used by the generic algorithms. The only real challenge in defining the interface is in taking care not to define (prescribe) too much. For example, an object-oriented approach to a graph library interface might define a particular vertex list class to provide access to all the vertices in a graph:

```
class Graph {

// ...

list<Vertex> adj();

// ...

};
```

However, in the interest of flexibility, we do not necessarily want to define the graph interface based on a particular set of classes. Instead, we should define a set of minimal requirements that any type must meet to be considered a graph. As with STL iterators, the requirements are expressed as a set of valid expressions and associated types (that are accessed through a traits class). We use the term "concept" to refer to a collection of these requirements. By defining the interface in this way we greatly increase the opportunities for reuse. Many different graph data structure implementations (typically optimized for different situations) can all share this common interface and be used by the same generic algorithms. The interface we present here is the third revision of the graph interface. With each revision we have fine-tuned the interface to be more generic and easier to use. This revision (which should be the final one) is a result of collaborations with Dietmar Kühl and other members of the Boost group whom we are working with to define a standard graph interface.

As with the iterator concepts of the STL, we factor the graph into several concepts (*InputIterator*, *RandomAccessIterator,* etc). This enables us to succinctly state the requirements for algorithms without overstating them.

The graph concepts are listed in Figure 1. The arrows denote the refinement relation, where one concept adds to the requirements of another concept. The requirements for each of the graph concepts is in Table 2. The notation used in the requirements table is in Table 1. There are several references in the requirements table to concepts such as *Assignable*, whose definitions can be found in the online documentation for the SGI STL implementation or in Austern's book, *Generic Programming and the STL*.

In addition to the graph concepts, one new iterator concept had to be created -- the *MultiPassInputIterator*. This concept is a refinement of *InputIterator* and adds the requirements that the iterator can be used to make multiple passes through a range, and that if *it*1==*it*2 and *it*1 is dereferenceable then ++*it*1==++*it*2. All of the functions in the graph traversal interface are assumed to be amortized constant time and space. That is, the stated complexity for graph algorithms will be valid so long as the graph traversal operations are amortized constant time.

Property Accessors

Graph algorithms need to access properties associated with the vertices and edges of a graph. For example, problem data (such as the length or capacity of an edge) is used by the algorithms, as well as auxiliary data flags (like color) to indicate whether a vertex has been visited. There are many possibilities for how these properties can be stored in memory ranging from members of vertex and edge objects, to arrays indexed by some index, to properties that are computed when needed, such as the distance between two vertices in a graph with Euclidian distances. To relieve generic algorithms from the details of the underlying property representation, property accessor abstraction is introduced. (In previous papers describing GGCL, the property accessor concept was named *Decorator* and used *operator[]* instead of *put()* and *get()*. In Kuhl's masters thesis, property accessors are called "data accessors.")

Several categories of property accessors provide different access capabilities:

- *readable*. The associated property data can only be read. The data is returned by value. Many property accessors defining the problem input (such as edge weight) can be defined as readable property accessors.
- *writeable*. The associated property can only be written to. The parent array used to record the paths in a breadth-first search tree is an example of a property accessor that would be defined writeable.
- *read/write*. The associated property can both be written and read. The distance property use in Dijkstra's shortest paths algorithm (Listing Two) would need to provide both read and write capabilities.
- *lvalue*. The associated property is actually represented in memory and it is possible to get a reference to it. The property accessors in the *lvalue* category also support the requirements for read/write property accessors.

Table 4 lists the requirements for the respective categories of property accessors, while Table 3 lists the notation used in the property accessor table. In addition to the requirements in the table, property accessors must meet the requirements of *CopyConstructible*. To use property accessors, it is necessary to have some object identifier that is passed to the property accessor. The description of property accessors is purposefully vague on the exact type of the identifier, mainly to allow for flexibility. Since the property accessor operations are global functions, it is possible to overload the accessor functions such that nearly arbitrary object identifiers can be used. In the context of graph algorithms, the object identifiers are typically the vertex and edge descriptors described in Table 2.

Graph Visitors

In the same way that function objects or functors are used to make STL algorithms more flexible, we can use functor-like objects to make the graph algorithms more flexible. (As of this writing, the *GraphVisitor* concept has not been considered for part of the Boost graph interface.) We use the name "GraphVisitor" for this concept because the intent is similar to the Gang of Four Visitor pattern (see *Design Patterns,* by Gamma et al.). We wish to add operations to be performed on the graph without changing the source code for the graphs or for the generic algorithms. Table 5 shows the definition of the *GraphVisitor* concept. In the table, *v* is a visitor object, *u* and *s* are vertices, *e* is an edge, *g* is a graph, and *bag* is a bag (a container with the methods *push()*, *pop()*, and *top()*, such as a stack or queue).

The *GraphVisitor* is more complex than a function object, since there are several well-defined entry points at which the user may want to introduce a call-back. For example, *discover()* is invoked when an undiscovered vertex is encountered within the algorithm. The *process()* method is invoked when an edge is encountered. The *GraphVisitor* concept plays an important role in the GGCL algorithms.

The Generic Graph Component Library

The Generic Graph Component Library is a collection of algorithms and data structures that meet the interface requirements described in the previous sections. Currently, the algorithms include many of the classical graph algorithms and several sparse matrix ordering algorithms, though the list is continually growing. A recent addition (see Table 6) is the Self Avoiding Walk (SAW) used in adaptive mesh refinement algorithms.

Perhaps one of the most exciting aspects of the GGCL implementation (at least from the implementor's point of view) is the way it takes advantage of the internal reuse that naturally occurs within graph algorithms. The basis of this reuse is the graph search pattern that underlies most graph algorithms.

With the use of the *GraphVisitor* concept, it is possible for GGCL to actually define a generalized graph *search()* function, which can then be used to implement most other graph algorithms. The code for *graph_search()* is given in Listing Six. The *tie()* function used in *graph_search()* is a nice utility function created by Jaakko Jarvi that lets the values of a pair (or *n*-tuples) be assigned to variables in a way reminiscent of the language ML.

With the use of the *graph_search()* function, the implementation of *breadth_first search()* and *depth_first search()* becomes somewhat trivial. The BFS uses a queue while DFS uses a stack, and they both use the coloring visitor to keep track of which vertices have been visited. The implementation of BFS and DFS is given in Listing Seven, and the coloring visitor in Listing Eight. The template argument *Super* in the coloring visitor may appear somewhat mystifying at first. We are using the Mixin technique to layer visitors, allowing multiple visitors to be applied during a single traversal of the graph.

The implementation of topological *sort()* is a great example of the power of graph visitors. As described in *Introduction to Algorithms*, by Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest (MIT Press, 1990), a topological sort can be accomplished with a depth-first search, where each vertex is output as it is finished. The GGCL algorithm is coded exactly in this way, as in Listing Nine.

GGCL Graph Data Structures

The GGCL data structures components are highly modular, designed to provide plug-and-play levels of flexibility. There are facilities for attaching properties to GGCL graphs in a variety of ways, and GGCL graphs are designed to work well with the STL containers (or customized containers) as the underlying storage. For example, the *vec_adj_list* GGCL module turns a *std::vector<EdgeList>* into a type that conforms to the GGCL graph interface. It does this without wrapping *std::vector<EdgeList>* in an adaptor object. Instead, it merely overloads a few global functions. The *std::vector<EdgeList>* can then be passed as is to the GGCL graph algorithm. We call this technique "external adaptation."

The graph interface is defined solely in terms of global functions, so GGCL provides overloads for the interface functions and specifies *std::vector<EdgeList>* as the graph argument. An object of type *std::vector<EdgeList>* can then be passed as is to a GGCL graph algorithm. Listing Ten gives a short example of this.

GGCL also provides support for many of the other common graph representations, including adjacency matrices and pointer-based (dynamic) graphs.

Efficiency and Performance

Efficiency is typically advertised as yet another advantage of generic programming -- and these claims are not simply hype. The efficiency that can be gained through the use of generic programming is astonishing. For example, the Matrix Template Library, a generic linear algebra library written completely in C++, is able to achieve performance as good as or better than vendor-tuned math libraries.

The flexibility within the GGCL is derived exclusively from static polymorphism (templates), not from dynamic polymorphism. As a result, all dispatch decisions are made at compile time, allowing the compiler to inline every function in the GGCL graph interface. Hence the abstraction penalty of the GGCL interface is completely eliminated. The machine instructions produced by the compiler are equivalent to what would be produced from hand-coded graph algorithms in C or Fortran.

Comparison to General-Purpose Libraries

Using a concise predefined implementation of adjacency list graph representation in GGCL, we compare the performance of the BFS algorithm (calculating the distance from the source and the predecessor for each vertex) with those in LEDA (Version 3.8), a popular object-oriented graph library, and those in GTL.

Figure 2 shows the results of the BFS applied to randomly generated graphs having a varying number of edges and a varying number of vertices. The y-axis is logarithmic. All results were obtained on a Sun Microsystems Ultra 30 with the UltraSPARC-II 296MHz microprocessor. For these experiments, GGCL is 5 to 7 times faster than LEDA.

Comparison to Special-Purpose Library

In addition, we demonstrate the performance of a GGCL-based implementation of the multiple minimum degree algorithm using selected matrices from the Harwell-Boeing collection and the University of Florida's sparse matrix collection. Our tests compare the execution time of our implementation against that of the equivalent SPARSPAK Fortran algorithm (GENMMD). For each case, our implementation and GENMMD produced identical orderings. The performance of our implementation is essentially equal to that of the Fortran implementation and even surpasses the Fortran implementation in a few cases.

Availability

The source code and complete documentation for the GGCL can be downloaded from the GGCL web site at http://www .lsc.nd.edu/research/ggcl and from *DDJ* (see "Resource Center," page 5). The graph interface described here can be found at the Boost home page at http://www.boost.org/.

References

Austern, Matthew H. *Generic Programming and the STL*. Addison-Wesley-Longman, 1998.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

George, Alan, and Joseph W.H. Liu. "User's Guide for SPARSPAK: Waterloo Sparse Linear Equations Packages," *Technical Report, Department of Computer Science,* University of Waterloo, Waterloo, Ontario, 1980.

Heber, G., R. Biswas, and G.R. Gao. "Self-Avoiding Walks Over Adaptive Unstructured Grids," *Parallel and Distributed Processing*. Springer-Verlag, 1999.

Jarvi, Jaakko "Ml-style Tuple Assignment in Standard C++: Extending the Multiple Return Value Formalism," Technical Report 267, *TUCS,* April 1999.

Kuhl, Dietmar. *Design Patterns for the Implementation of Graph Algorithms*, Technische UniversitŠt Berlin, July 1996.

Liu, Joseph W.H. "Modification of the Minimum-degree Algorithm by Multiple Elimination," *ACM Transaction on Mathematical Software*, 1985.

Lo Russo, Graziano. "An Interview with Alexander Stepanov," *Edizioni Infomedia srl*, 1997.

Myers, Nathan. "A New and Useful Template Technique: Traits," *C++ Report*, June 1995.

Samaragdakis, Yannis and Don Batory. "Implementing Layered Designs With Mixin Layers," *Europe Conference on Object-Oriented Programming*, 1998.

Siek, Jeremy G. and Andrew Lumsdaine. "The Matrix Template Library: A Generic Programming Approach to High- performance Numerical Linear Algebra," *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

DDJ

Listing One

```
depth_first_search(Graph G, Color color)
{
   for each vertex u in vertices(G) {
     color[u] = white;
     visit(u, color);
   }
}
visit(Vertex u, Color color)
{
   color[u] = gray;
   for each vertex v in adj(u) {
      if (color[v] = white)
         visit(v);
   }
   color[u] = black;
}
```

Back to Article

Listing Two

```
dijkstra_shortest_paths(Graph G, Vertex s, Distance d, Weight w, Parent p)
{
   PriorityQueue Q;
   push(s, Q);
   while (Q is not empty) {
      vertex u = pop(Q);
      for each edge e=(u,v) in out-edges(u)
        if (d[v] > d[u] + w(e))
           p[v] = u;
   }
}
```

Back to Article

Listing Three

```
bellman_ford_shortest_paths(Graph G, Weight w, Distance d, Parent p)
{
   for k = 0...num_vertices(G)
      for each edge e=(u,v) in edges(G)
          if (relax(e, g, w, d))
              p[v] = u;
    for each each e=(u,v) in edges(g)
      if (d[u] + w[e] < d[v])
         return false;
}
```

Back to Article

Listing Four

```
extend_shortest_paths(AdjacencyMatrix G, Distance d, Weight w)
{
   for (i=1...N)
      for (j=1...N)
         for (k=1..N) {
           edge e_ij = G(i,j), e_ik = G(i,k), e_kj = G(k,j);
           d[e_ij] = min( d[e_ij], d[e_ik] + w[e_kj];
       }
}
```

Back to Article

Listing Five

```
// WRONG! Overspecified requirements
template <class RandomAccessIter1, class RandomAccessIter2>
void copy(RandomAccessIter1 first, RandomAccessIter1 last,
       RandomAccessIter2 result)
{
    while (first != last)
       *first++ = *result++;
}
// Just right
template <class InputIterator, class OutputIterator>
void copy(InputIterator first, InputIterator last, OutputIterator result)
{
   while (first != last)
       *first++ = *result++;
}
```

Back to Article

Listing Six

```
template
<class IncidenceGraph, class Vertex, class Bag, class Visitor>
void graph_search(IncidenceGraph& g, Vertex s, Bag& bag, Visitor visitor)
{
   Vertex u;
   typename graph_traits<IncidenceGraph>::incidence_iterator i, end;
   visitor.start(s);
   bag.push(s);
   while (! bag.empty()) {
      u = bag.top();
      if (visitor.is_undiscovered(u)) {
          visitor.discover(u, bag);
          for (tie(i,end) = out_edges(u,g); i != end; ++i)
            visitor.process(*i, g, bag);
      } else {
        visitor.finish(u);
        bag.pop();
      }
   }
}
```

Back to Article

Listing Seven

```
template
  <class IncidenceGraph, class Vertex, class Color, class Visitor>
void breadth_first_search(IncidenceGraph& g,
                 Vertex s, Color color, Visitor visitor)
{
    boost::queue<Vertex> q;
    graph_search(g, s, q, visit_color(color, visitor));
}
template
<class VertexListGraph, class Color, class Visitor>
void depth_first_search(VertexListGraph& g, Color color, Visitor visitor)
{
   std::stack<Vertex> std;
   typename graph_traits<VertexGraph>::vertex_iterator i, end;
   for (tie(i,end) = vertices(g); i != end; ++i)
       graph_search(g, *i, stk, visit_color(color, visitor));
}
```

Back to Article

Listing Eight

```
template < class Color, class Super = null_visitor>
class coloring_visitor : public Super {
   // constructors and typedefs ...
   template <class Vertex>
   void initialize(Vertex u) {
      set(color, u, color_tr::white());
      Super::initialize(u);
   }
   template <class Vertex, class Bag>
   void discover(Vertex u, Bag& bag) {
      set(color, u, color_tr::gray());
      Super::discover(u, bag);
   }
   template <class Vertex>
   void finish(Vertex u) {
      if (get(color, u) != color_tr::black()) {
      set(color, u, color_tr::black());
      Super::finish(u);
   }
}
template <class Edge, class Graph, class Bag>
bool process(Edge e, Graph& g, Bag& bag) {
   typedef typename graph_traits<Graph>::vertex_descriptor Vertex;
   Vertex v = target(e, g);
   if ( is_undiscovered(v) ) {
      bag.push(v);
      Super::process(e, g, bag);
      return true;
    } else if ( FocusOnEdge )
      Super::process(e, g, bag);
      return false;
   }
   template <class Vertex>
   bool is_undiscovered(Vertex u) {
       return (get(color,u) == color_tr::white());
   }
protected:
   Color color;
};
// Helper class for creating color visitors
template <class Color, class Super>
coloring_visitor<Color, Super>
visit_color(Color c, Super b) {
   return coloring_visitor<Color, Super>(c, b);
}
```

Back to Article

Listing Nine

```
template
<class VertexListGraph, class OutputIter, class Color, class Visitor>
void topological_sort(
   VertexListGraph& G, OutputIter result, Color color, Visitor visitor)
{
   topo_sort_visitor<OutputIter, Visitor> topo_visit(c, visitor);
   depth_first_search(G, topo_visit, color);
}
template <class OutputIterator, class Super = null_visitor>
struct topo_sort_visitor : public Super {
   //constructors ...
   template <class Vertex>
   void finish(Vertex u) {
      *result = u; ++result;
      Super::finish(u);
   }
   OutputIterator result;
};
```

Back to Article

Listing Ten

```
#include <ggcl/vec_adj_list.hpp>
// ...
typedef std::vector< std::list<int> > Graph;
Graph g(N);
// fill the graph...
std::vector<int> color(N, WHITE), discover(N), finish(N);
depth_first_search(g, color.begin(),
    visit_time(discover.begin(), finish.begin()));
```

Back to Article