The Generic Graph Component Library

Lie-Quan Lee Jeremy G. Siek Andrew Lumsdaine
Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
Tel: (219) 631-3906 Fax: (219) 631-9260
llee1@lsc.nd.edu jsiek@lsc.nd.edu lums@lsc.nd.edu

ABSTRACT

In this paper we present the Generic Graph Component Library (GGCL), a generic programming framework for graph data structures and graph algorithms. Following the theme of the Standard Template Library (STL), the graph algorithms in GGCL do not depend on the particular data structures upon which they operate, meaning a single algorithm can operate on arbitrary concrete representations of graphs. To attain this type of flexibility for graph data structures, which are more complicated than the containers in STL, we introduce several concepts to form the generic interface between the algorithms and the data structures, namely, Vertex, Edge, Visitor, and Decorator. We describe the principal abstractions comprising the GGCL, the algorithms and data structures that it provides, and provide examples that demonstrate the use of GGCL to implement some common graph algorithms. Performance results are presented which demonstrate that the use of novel lightweight implementation techniques and static polymorphism in GGCL results in code which is significantly more efficient than similar libraries written using the object-oriented paradigm.

1 INTRODUCTION

The graph abstraction is widely used to model a large variety of structures and relationships in computer science. Graph algorithms are extremely important in such diverse application areas as design automation, transportation, optimization, and databases. Consequently, the implementation of graph algorithms is an important enterprise that can be greatly facilitated by the availability of high-quality software for realizing graph algorithms. (By "high-quality" in this case we take to mean, such attributes as functionality, reliability, usability, efficiency, maintainability, and portability [15].)

There are several existing general purpose graph libraries, such as LEDA [14], the Graph Template Library (GTL) [5], Combinatorica [21], and Stanford GraphBase [11]. Sources such as Netlib [1] and [22] represent repositories of graph algorithms. These libraries and repositories represent a significant amount of potentially reusable algorithms and data structures. However, none of these libraries faithfully follows the *generic programming* paradigm [3] (also see Section 1.1) and are therefore far more rigid (and much less reusable) than necessary.

These libraries are inflexible in several respects. First, the user is restricted to the graph data structures provided by the library. Second, the graph algorithms often do not provide explicit mechanisms for extension, making it difficult or impossible for users to customize vanilla algorithms to meet their needs. Finally, the manner in which these libraries associate graph properties (such as color or weight) with a graph data structure is often inflexible and hard coded into the algorithms or data structures. Ultimately, these (and other) libraries are fundamentally limited in terms of their flexibility by their design and implementation.

1.1 Generic Programming

Recently, generic programming [3] has emerged as a powerful new paradigm for library development. The fundamental principle of generic programming is to separate algorithms from the concrete data structures on which they operate based on the underlying abstract problem domain concepts, allowing the algorithms and data structures to freely interoperate. That is, in a generic library, algorithms do not manipulate concrete data structures directly, but instead operate on abstract interfaces defined for entire equivalence classes of data structures. A single generic algorithm can thus be applied to any particular data structure that conforms to the requirements of its equivalence class. In the celebrated Standard Template Library (STL) [12], the data structures are containers such as vectors and linked lists. *Iterators* form the abstract interface between algorithms and containers. Each STL algorithm is written in terms of the iterator interface and as a result each algorithm can operate with any of the STL containers. In addition, many of the STL algorithms are parameterized not only on the type of iterator used for traversal, but on the type of operation that is applied during the traversal. For example, the `transform()` algorithm has a parameter for a `UnaryOperator` function object (functor). Likewise, some of the STL containers are parameterized with function objects, such as the `Compare` template parameter for the `std::map` and `std::set` classes.

Concepts The GGCL library is developed using terminology similar to that of the SGI STL [3]. In the parlance of the SGI STL, the set of requirements on a template parameter for a generic algorithm or data structure is called a *concept*. (Generic programming is sometimes referred to as "programming with concepts.") Types that fulfill the requirements of a concept are said to *model* that concept. For example, pointer types such as `int*` model (or, are models of) the concept **RandomAccessIterator** as defined by the STL. The class types `std::vector<T>` and `std::list<T>` are models of the **Container** concept. Concepts can extend other concepts, which is referred to as *refinement*. We use a **bold sans serif** font for all concept identifiers.

For example, one version of the STL `accumulate()` algorithm is prototyped as follows:

```
template <class InputIterator, class T>
T accumulate(InputIterator first,
             InputIterator last, T init);
```

For proper operation of `accumulate()`, we require that the type of the arguments `first` and `last` be models of the concept **InputIterator**. We note that the C++ language does not provide support for concept checking. That is, although we give the template parameter to `accumulate()` the name of **InputIterator**, the name is merely a placeholder. The C++ language does not enforce that the arguments passed to `accumulate()` actually *be* a model of **InputIterator**. Naturally, if the arguments do not model (or refine) **InputIterator**, it is likely that an error will occur when compiling that particular instantiation of `accumulate()`, but that is not the same (semantically) as identifying that the instantiation itself is in error.

1.2 A Generic Graph Library

The domain of graphs and graph algorithms is a natural one for the application of generic programming. There are many kinds of graph representations, such as adjacency matrix, adjacency list, and dynamic pointer-based graphs and there also numerous graph algorithms. In a generic graph library, we should be able to write each algorithm only once and use it with any graph data structure.

In addition, the algorithms should be flexible, so that algorithm *patterns* such as Depth First Search (DFS) can be reused. For example, one may want to use DFS to traverse a graph and calculate whether vertices are reachable. In another situation, DFS could be used to record the order of vertices. In yet another situation, one may want to use DFS to calculate reachability *and* the order of vertices. These requirements are similar to those of most general purpose libraries, which would perhaps suggest that the generic programming style of the STL might be directly applicable to the creation of a graph library.

However, there are important (and fundamental) differences between the types of algorithms and data structures in STL and the types of algorithms and data structures in a generic graph library. In particular, there are numerous ways in which edge and vertex properties (such as color and weight) are implemented and associated with vertices and edges. One way is to store properties in an array indexed by vertex ID. Another method, suitable for graphs with explicit storage for each vertex, is to store the properties inside the vertex data structure. Rather than imposing one approach over another, a generic graph library should provide an generic means for accessing the properties of a vertex or edge, regardless of the manner in which the properties are stored.

To accommodate the unique properties of graphs and graph algorithms, we introduce several concepts upon which the interface between graphs and graph algorithms will be built: **Vertex**, **Edge**, **Visitor**, and **Decorator**. The latter two concepts are similar in spirit to the "Gang of Four" [6] patterns **Visitor** and **Decorator** but are quite different in terms of implementation techniques.

In the following sections we describe the design and implementation of the Generic Graph Component Library (GGCL). This library was designed and implemented from the ground up with generic programming as its fundamental paradigm. In the next section, we define the abstract graph interface and concepts used by GGCL in more detail. The generic graph algorithms in GGCL are described in Section 3, and Section 4 discusses the main implementation issues. Experimental results demonstrating the performance of GGCL (and comparing the performance to several other graph libraries) are given in Section 5. Finally, our conclusions are provided in Section 6.

2 ABSTRACT GRAPH INTERFACE

The domain of graph data structures and algorithms is in some respects more complicated than that of containers. The abstract iterator interface used by STL is not sufficiently rich to encompass the numerous ways that graph algorithms may traverse a graph. Instead, we formulate an abstract interface that serves the same purpose for graphs that iterators do for basic containers (though iterators still play a large role). Figure 1 depicts the analogy between the STL and the GGCL.

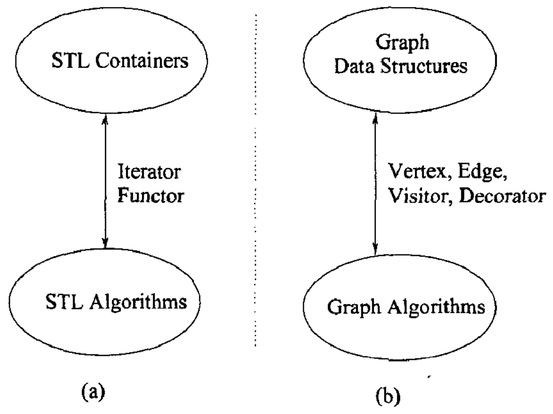

Figure 1: The analogy between the STL and the GGCL.

2.1 Formal Graph Definition

The appropriate abstract graph interface can be derived directly from the formal definition of a graph [4]. A graph G is a pair (V,E), where V is a finite set and E is a binary relation on V. V is called a *vertex set* whose elements are called *vertices*. E is called an *edge set* whose elements are called *edges*. An edge is an ordered or unordered pair (u,v) where $u,v \in V$. If (u,v) is and edge in graph G, then vertex v is *adjacent* to vertex u. Edge (u,v) is an *out-edge* of

Expression	Return Type	Description
`X::vertex_type`		A model of Vertex
`e.source()`	`vertex_type`	The *source* vertex of edge e
`e.target()`	`vertex_type`	The *target* vertex of edge e

Table 3: The specification of the Edge concept.

vertex *u* and an *in-edge* of vertex *v*. In a *directed* graph edges are ordered pairs while in a *undirected* graph edges are unordered pairs. In a *directed* graph an edge *(u,v)* leaves from the *source* vertex *u* to the *target* vertex *v*.

2.2 Graph Concepts

The three main concepts necessary to define our graph interface are Graph, Vertex, and Edge. Each of our concept definitions derives directly from the formal graph definition. By design we have tried to keep the interface close to that of existing graph libraries and to the common graph algorithm notations.

Graph The Graph concept merely contains a set of vertices and a set of edges and a tag to specify whether it is a directed graph or an undirected graph. Table 1 lists the Graph requirements, including its associated types. Note that the specific types of the sets are not specified. The only requirement is that *vertex set* be a model of ContainerRef and its `value_type` a model of Vertex. The *edge set* must be a model of ContainerRef and its `value_type` a model of Edge. [1]

Vertex The Vertex concept provides access to the adjacent vertices, the out-edges of the vertex and optionally the in-edges. Table 2 lists the Vertex requirements, including its associated types.

Edge An Edge is an ordered or unordered pair of vertices. The elements comprising the Edge are the *source* vertex and the *target* vertex. In the unordered case it is just assumed that the position of the *source* and *target* vertices are interchangeable (and, correspdondingly, that the Graph is undirected). Table 3 lists the Edge requirements.

Decorator As mentioned in the introduction, we would like to have a generic mechanism for accessing vertex and edge properties of a graph (e.g., color or weight) from within an algorithm. The generic access method is necessary to support the numerous ways in which the properties can be stored as well as the numerous ways in which access to that storage can be implemented. We give the name Decorator to this concept since it is similar to the intent of the "Gang of Four" Decorator pattern [6], (which dynamically add properties to an object).

[1]The ContainerRef concept is very similar to the Container concept of the STL, except that the ContainerRef concept lacks the notion of "ownership", so making a copy of a ContainerRef object merely creates an alias to the same underlying container. Obviously, a reference to a Container object satisfies this requirements.

Table 4 gives the definition of the Decorator concept. A Decorator is very similar to a functor, or function object. We use the method of `operator[]` instead of `operator()` since it is a better match for the commonly used graph algorithm notations.

Visitor 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. We use the name Visitor for this concept because the intent is similar to the well known visitor pattern [6]. 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 Visitor concept. In the table, v is a visitor object, u and s are vertices, and e is an edge. Our Visitor is somewhat 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 Visitor concept plays an important role in the GGCL algorithms.

The Decorator and Visitor concepts are used in the GGCL graph algorithm interfaces to allow for maximum flexibility. Below is the prototype for the GGCL depth first search algorithm, which includes parameters for both a Decorator and a Visitor object. There are two overloaded versions of the interface, the first one in which there is a default ColorDecorator. The default decorator accesses the color property directly from the graph vertices. This is analogous to the STL algorithms. For example, there are two overloaded versions of the `lower_bound()` algorithm. The default uses whatever less-than operator is defined for the element type, while the other version takes an explicit BinaryOperator functor argument.

```
template <class Graph, class Visitor>
void dfs(Graph& G, Visitor visit);

template <class Graph, class Visitor, class ColorD>
void dfs(Graph& G, Visitor visit, ColorD color);
```

3 GENERIC GRAPH ALGORITHMS

The generic graph algorithms are written solely in terms of the abstract graph interface defined in the previous section. They do not make assumptions about the actual graph type or the underlying data structure. This enables a high degree of reuse for the algorithms.

Breadth First Search Our first example is the classic Breadth First Search (BFS) algorithm. As a starting point, we will look at an adaptation of the textbook [4] algorithm, written in terms of the GGCL interface (Figure 2). This algorithm calculates the distance from a source vertex to all other reachable vertices in the graph. It also records the predecessor, or parent, of each vertex. The color decorator is used by the algorithm to keep track of which vertices have been visited (in case there are cycles). This algorithm is provided as a straw man — it is not the BFS that is actually provided by GGCL.

Expression	Return Type	Description
`X::vertex_type`		A model of Vertex
`X::edge_type`		A model of Edge
`X::vertices_type`		A ContainerRef of vertices
`X::edges_type`		A ContainerRef of edges
`X::direct_tag`		A tag of either directed or undirected
`g.vertices()`	`vertices_type`	The *vertex set* of graph `g`
`g.edges()`	`edges_type`	The *edge set* of graph `g`

Table 1: The specification of the Graph concept.

Expression	Return Type	Description
`X::edge_type`		A model of Edge
`X::vertexlist_type`		A ContainerRef of vertices
`X::edgelist_type`		A ContainerRef of edges
`u.adj()`	`vertexlist_type`	The adjacent vertices of vertex `u`
`u.out_edges()`	`edgelist_type`	The out edges of vertex `u`
`u.in_edges()`	`edgelist_type`	The in edges of vertex `u` (optional)

Table 2: The specification of the Vertex concept.

Expression	Return Type	Description
`return_type`		A type of object accessed by the decorator
`d[u]`	`return_type`	The decorating property `d` of Vertex `u`

Table 4: The specification of the Decorator concept.

Expression	Return Type	Description
`v.initialize(u)`	`void`	Invoked during initialization.
`v.start(s)`	`void`	Invoked at the beginning of algorithms.
`v.discover(u)`	`void`	Invoked when an undiscovered vertex is encountered.
`v.finish(u)`	`void`	Invoked when algorithms finish visiting a vertex.
`v.process(e)`	`bool`	Invoked when an edge is traversed.

Table 5: The specification of the Visitor concept.

```
template <class Graph, class Color, class Distance, class Predecessor>
void textbook_BFS(Graph& G, typename Graph::vertex_type s,
                  Color color, Distance d, Predecessor p)
{
  typedef typename Graph::vertex_type Vertex;
  typename Graph::vertices_type::iterator ui;
  typename Vertex::edgelist_type::iterator ei;

  //initialization
  for (ui = G.vertices().begin(); ui != G.vertices().end(); ++ui) {
    color[*ui] = WHITE;
    d[*ui] = INF;
  }

  //starting from vertex s
  color[s] = GRAY;
  d[s] = 0;
  std::queue<Vertex> Q;
  Q.push(s);

  //main algorithm
  while (! Q.empty()) {
    Vertex u = Q.front();
    for (ei = u.out_edges().begin(); ei != u.out_edges().end(); ++ei) {
      Vertex v = (*ei).target();
      if (color[v] == WHITE) {
        color[v] = GRAY;
        d[v] = d[u] + 1;
        p[v] = u;
      }
    }
    Q.pop();
    color[u] = BLACK;
  }
}
```

Figure 2: The textbook Breadth First Search algorithm.

As it stands, this algorithm is quite useful, but in many ways it is not sufficiently general. In GGCL we capture the essence of the Breadth First Search pattern in a generic generalized BFS algorithm, as shown in Figure 3. The `visitor` parameter provides flexibility in the kinds of actions performed during the BFS. There are several call-back points associated with the visitor, including `start()`, `discover()`, `process()`, and `finish()`. The `Q` parameter allows for different kinds of queues to be used. The `visited` functor was added for algorithms that would like to perform an action on subsequent encounters with a vertex after it is discovered. The initialization steps were moved to a separate function to accomodate the need for certain type-specific initializations (e.g., a graph consisting only of edge lists without explicit vertex storage).

In the `generalized_BFS()` algorithm we use the expression `u.out_edges()` to access the list of edges leaving vertex u. Iterators of this list are used to access each of the edges. The Visitor is used to parameterize the operations performed on each edge as it is discovered. The algorithm also inserts each discovered vertex onto `Q` or, if the vertex has already been visited, invokes the `visited` functor. Target vertices are accessed through `e.target()`.

The `generalized_BFS()` algorithm is ideal for reuse in other algorithms. Figure 4 gives an overview of the algorithms we have constructed so far using the `generalized_BFS`. A variation on the UML [10, 16] notation is used to represent the algorithms, visitor classes, and concepts. A solid box stands for an algorithm or a class. Dotted boxes are template arguments or concepts. The classes within a concept box are models of the concept. The notation `<<bind>>` indicates the binding of formal template arguments to concrete types. Unbound template arguments are marked with underscores, giving a notation for partial specialization.

In Figure 4 we can see how particular parameters are chosen in the creation the different algorithms. First, with regards to the queue type, the BFS algorithm in Figure 5 is constructed by using the STL `queue`, while Dijkstra's single-source shortest path and Prim's minimum spanning tree algorithms are constructed with a mutable priority queue (a priority queue with a decrease-key operation [4]). Implementation of the textbook BFS algorithm using `generalized_BFS()` is shown in Section 4.3. A customized queue is used with BFS in the Reverse Cuthill McKee sparse matrix ordering algorithm [8, 17].

Looking at the `Visitor` parameter, we see that the normal BFS algorithm uses the `bfs_visitor` which keeps track of the vertex colors. Dijkstra's and Prim's algorithms both use the `weighted_edge_visitor`, the only difference between them being the operator that is bound to BinaryOp parameter. Dijkstra's algorithm is implemented using a `plus` functor, and Prim's is implemented using the `project2nd` functor, which is just a binary operator that returns the 2nd argument. Figure 6 shows the GGCL implementation of Prim's minimum spanning tree algorithm while Figure 7 shows the GGCL implementation of Dijkstra's single source shortest path algorithm. The algorithms consist simply of some setup declarations, initialization and a call to `generalized_BFS`. The only difference between the two algorithms is the function object used inside `weighted_edge_visitor`.

The `Visited` parameter is simply a null operation for the normal BFS algorithm, while in the Dijkstra's and Prim's algorithms it provides queue update by invoking the mutable priority queue' decrease-key operation.

Depth First Search The Depth First Search is another fundamental traversal pattern in graph algorithms, and is a second source for reuse. Figure 8 depicts some algorithms that can be either directly derived from DFS, or that make use of it. The code example in Figure 9 gives the implementation of the topological sort algorithm, a classic example DFS algorithm reuse. The `topo_sort_visitor` merely outputs the vertex to the OutputIterator inside the `finish(u)` callback.

The concise implementation of algorithms such as Prim's Minimum Spanning Tree and Topological Sort is enabled by the genericity of the GGCL algorithms, allowing us to exploit the reuse that is inherent in these graph algorithms in a concrete fashion.

Currently, the GGCL includes a basic set of algorithms: DFS, BFS, Dijksta's algorithm for the Shortest Path problem, Prim and Kruskal algorithms for Minimum Spanning Tree, topological sort, and connected components. In addition we have implemented several graph algorithms for sparse matrix ordering, including the Reverse Cuthill McKee and the Minimum Degree algorithms. GGCL is an ongoing project and a number of generic graph algorithms are in the process of being implemented.

4 GGCL IMPLEMENTATION

4.1 Graph Data Structure Implementation

The GGCL graph data structures are constructed in a layered manner to provide maximum flexibility and reuse. The layered architecture also provides several different points of customizability. At one end of the spectrum one can use the graphs provided by GGCL and make small modification with little effort. In the middle of the spectrum are graph types that can be pieced together from standard components such as lists and vectors. At the far end of the spectrum the user may already have their own data structure, and they just need to create a GGCL Graph compliant interface to his or her data structure.

Interfacing with external graph types To test the difficulty of creating a GGCL interface for non-GGCL graph types, we constructed a Graph interface for LEDA graphs. The interface code is 1 1/2 pages and took approximately 1 man-hour to develop.

Composing Graphs from standard containers The GGCL provides a framework for composing graphs out of standard containers such as STL `std::vector`, `std::list`, and matrices from the Matrix Template Library (MTL) [19], another generic component library we have developed. Of course, the composition mechanism will work for any STL Container compliant components, so this provides another avenue for extensibility by the user.

The set of graph configurations currently provided by GGCL are listed in Figure 10.

Below is an example of defining an adjacency-list graph type whose vertices have an associated color and whose edges have an associated weight.

```
template <class Vertex, class QType, class Visitor, class Visited>
void generalized_BFS(Vertex s, QType& Q, Visitor visitor, Visited visited)
{
  typedef typename Vertex::edgelist_type::value_type Edge;
  typename Vertex::edgelist_type::iterator ei;
  visitor.start(s);
  Q.push(s);
  while (! Q.empty()) {
    Vertex u = Q.front();
    Q.pop();
    visitor.discover(u);
    for (ei = u.out_edges().begin(); ei != u.out_edges().end(); ++ei) {
      Edge e = *ei;
      if (visitor.process(e))
        Q.push(e.target());
      else
        visited(visitor, Q, ei);
    }
    visitor.finish(u);
  }
}
```

Figure 3: The generalized Breadth First Search algorithm.

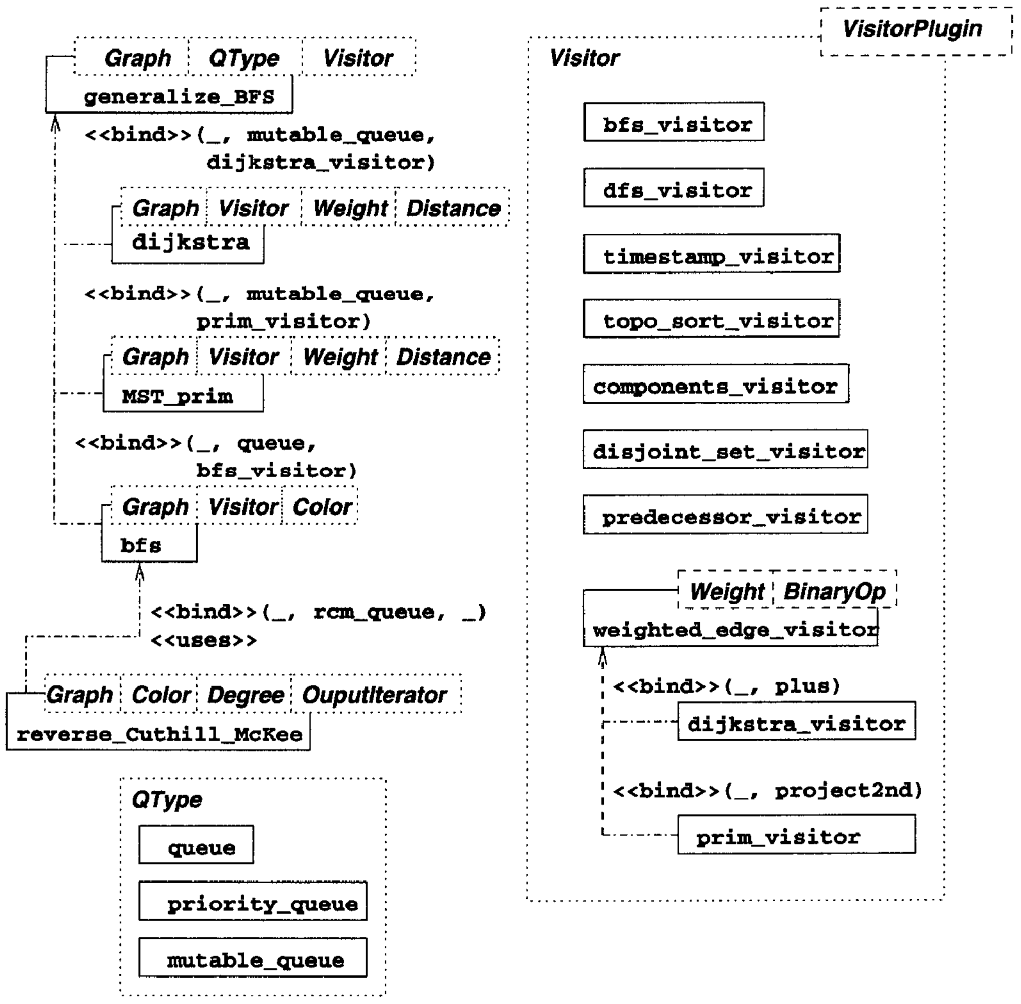

Figure 4: The BFS family of algorithms and the predefined set of visitors provided in GGCL.

```
template <class Graph, class Visitor, class ColorDecorator>
void bfs(Graph& G, Graph::vertex_type s, Visitor visit, ColorDecorator color)
{
  typedef typename Graph::vertex_type Vertex;
  std::queue<Vertex> Q;

  bfs_visitor<ColorDecorator, Visitor> visitor(color, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, null_operation());
}
```

Figure 5: The BFS algorithm in GGCL.

```
template <class Graph, class Visitor, class Distance, class Weight, class ID>
void prim(Graph& G, Graph::vertex_type s, Visitor visit, Distance d, Weight w, ID id )
{
  typedef typename Graph::vertex_type Vertex;
  typedef typename Distance::return_type D;
  typedef functor_less<Distance> Compare;

  Compare c(d);
  mutable_queue<Vertex, std::vector<Vertex>, Compare, ID > Q(G.num_vertices(), c, id);

  weighted_edge_visitor<Weight, Distance, Visitor, _project2nd<D,D> > visitor(w, d, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, queue_update());
}
```

Figure 6: The GGCL implementation of the Prim Minimum Spanning Tree algorithm as a call to `generalized_BFS()`. The Dijkstra's Single-Source Shortest Path algorithm can be realized in the same way simply by using a different function object in place of `_project2nd<D,D>`.

```
template <class Graph, class Visitor, class Distance, class Weight, class ID>
void dijkstra(Graph& G, Graph::vertex_type s, Visitor visit, Distance d, Weight w, ID id )
{
  typedef typename Graph::vertex_type Vertex;
  typedef typename Distance::return_type D;
  typedef functor_less<Distance> Compare;

  Compare c(d);
  mutable_queue<Vertex, std::vector<Vertex>, Compare, ID > Q(G.num_vertices(), c, id);

  weighted_edge_visitor<Weight, Distance, Visitor, plus<D> > visitor(w, d, visit);

  generalized_init(G, visitor);
  generalized_BFS(s, Q, visitor, queue_update());
}
```

Figure 7: The GGCL implementation of the Dijkstra's Single-Source Shortest Path algorithm as a call to `generalized_BFS()`.

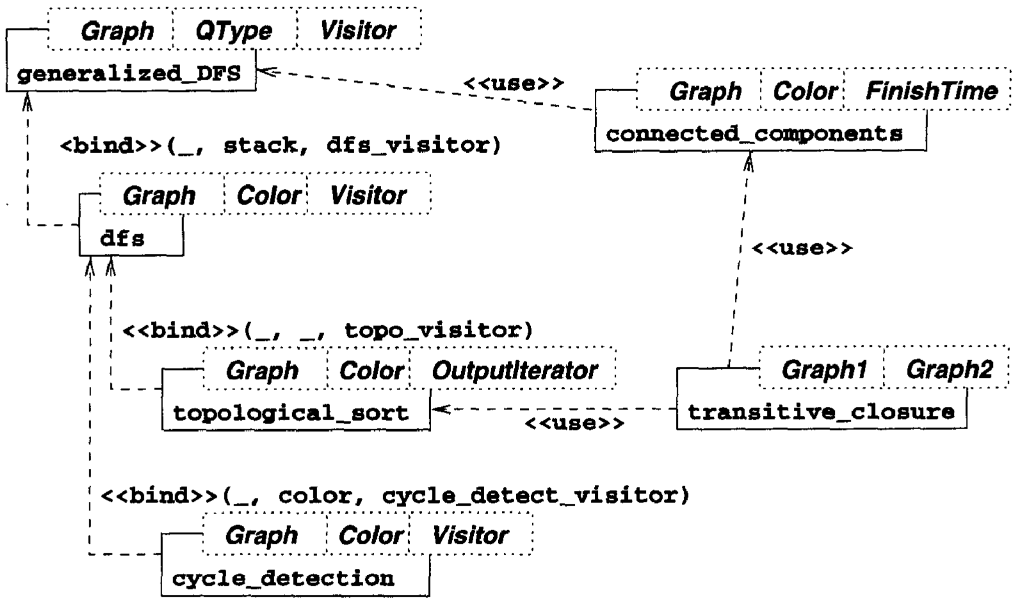

Figure 8: The family of DFS algorithms.

```
template <class Graph, class OutputIterator, class Visitor, class Color>
void topological_sort( Graph& G, OutputIterator result, Visitor visitor, Color color) {
  topo_sort_visitor<OutputIterator, Visitor> topo_visit(c, visitor);
  dfs(G, topo_visit, color);
}

template <class OutputIterator, class Super>
struct topo_sort_visitor : public Super {
  //constructors . . .

  template <class Vertex>
  void finish(Vertex u) {
    *result = u; ++result;
    Super::finish(u);
  }
  OutputIterator result;
};
```

Figure 9: The GGCL implementation of the topological sort algoritm using DFS.

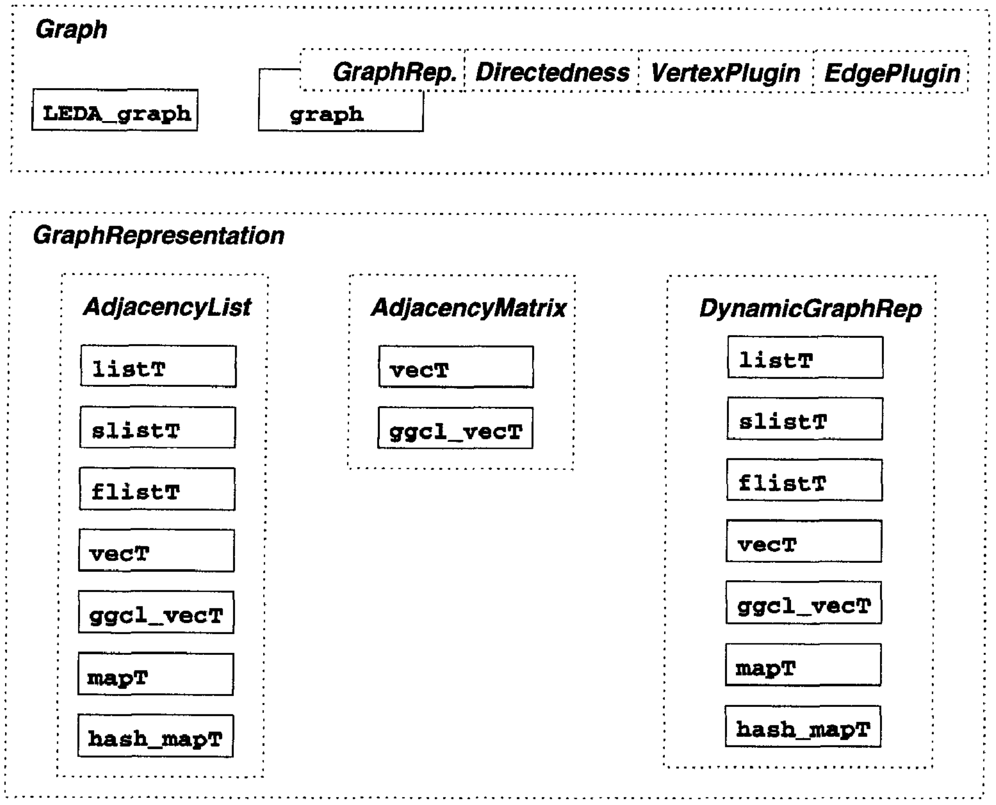

Figure 10: The Graph Components Provided By GGCL.

```
typedef graph<adjacency_list<vecT>, undirected,
          color_plugin<>, Weight<int> > myGraph;
```

Graph Representation The implementation framework centers around the main graph interface class and the GraphRepresentation concept. The graph interface class constructs the full graph interface based on the minimized interface exported by the GraphRepresentation concept. This allows full fledge GGCL Graphs to be constructed out of standard container components with very little work.

The GraphRepresentation concept is basically a 2D Container (a Container of Containers) coupled with four helper functions:

```
Iter2D get_target(Iter2D b, Iter1D i);
stored_edge& get_edge(Iter1D i);
bool add(EdgeList& elist, size_type vertex_num,
        const stored_edge& e);
void remove(EdgeList& elist, size_type vertex_num);
```

A 1D Container within a GraphRepresentation corresponds to the out-edge list for a particular vertex. In addition, there is a one-to-one correspondence between the 2D Iterator and the vertices of the graph.

The get_target() helper function is necessitated because the GGCL graph must be able to derive the target vertex from an edge object, through the information provided by the GraphRepresentation. The get_edge() function provides a generic access method to the extra edge properties stored within an edge list, and the add() and remove() methods provide a generic interface for adding and removing edges from a vertex.

The GraphRepresentation is further refined into three sub concepts, the AdjacencyList, AdjacencyMatrix, and DynamicGraphRep.

The AdjacencyList concept corresponds to a "sparse" or "compressed" representation of a graph. As such, further requirements are added to the 2D Container of the GraphRepresentation. For a model of AdjacencyList the inner container must be a variable-sized Container whose value_type is the size_type for a vertex if the graph has no extra edge-associated data, or a std::-pair<size_type,stored_edge> where the stored_edge is the type of an object containing any extra edge-associated data such as weight.

The AdjacencyMatrix concept corresponds to a "dense" representation of a graph, with boolean values for all vertex pairs, to mark them as connected or not.

The DynamicGraphRep concept requires its models to have a head pointer and explicitly stored vertex objects. Through the stored vertex it is able to access adjacent vertices.

Custom Graph Representations As an example of constructing customized models of GraphRepresentation, we show how one can build an AdjacencyList using std::vector and std::-list. The various parts of the GraphRepresentation are injected into the GGCL graph class by constructing a graph representation class. Figure 11 lists the implementation. One merely has to compose a couple of container types and fill in a few short functions. The `add()` and `remove()` methods are not depicted, but they are each approximately 5 lines.

4.2 Decorator Implementation

In some situations the particular property of vertices or edges is strongly associated with the graph and exists for the lifetime of the graph. For instance, the distance property could fall into this category. In other situations the property is only needed for a particular algorithm. Typically one would want to store a color property externally, since it may only be needed for a particular algorithm invocation. Thus there are two categories of decorators, *interior decorators* and *exterior decorators*. For exterior decorators, the decorating properties are stored outside of the graph object (they are passed directly to the GGCL algorithm) and the decorator will access the externally stored data indexed by the vertex or edge ID. On the other hand, if the decorating properties are stored inside of the graph object, the decorator consults the vertex or the edge objects to obtain the decorating property. Figure 12 shows the predefined models Decorator in GGCL.

Internally Stored Properties: Vertex and Edge Plugins For internal properties, the graph class provides optional parameterized storage plugins for both vertices and edges. This allows the user to plug in storage for an arbitrary set of decorating properties. For example, a graph with internally stored edge weights and color and distance properties for vertices could be defined as follow:

```
typedef color_plugin<distance_plugin<> > VPlugins;
typedef graph<adjacency_list<>, undirected,
              VPlugins, Weight<int> > myGraph;
```

The mixin technique [18] of templated inheritance is used to implement the layering of vertex and edge plugins. Figure 12 shows the decorators that are provided in GGCL. We have also created a mechanism so that users can easily create new custom storage plugins for decorating properties with user-defined names.

4.3 Visitor Implementation

To implement a model of Visitor one defines a class conforming to the Visitor concept and fills in the call-back methods (`discover()`, `process()`, etc.). Figure 13 shows the model of Visitor used to create the normal BFS algorithm from the `generalized_BFS`. This class is reponsible for keeping track of the vertex colors.

As in the decorator plugins, the mixin technique [18] is used to make visitors more extensible. This is the reason for the `Base` template argument, which allows visitors to be layered through inheritance, giving an arbitrary number of visitors a chance to perform actions during the algorithm (each call-back method must invoke in inherited call-back in addition to performing its own actions). If one wished to recreate the textbook BFS algorithm shown previously, which calculates distances and predecessors, one would call `bfs` with a distance and predecessor visitor. The GGCL has helper functions defined for creating the standard visitors.

```
bfs(G, s, visit_distance(d, visit_predecessor(p)));
```

where `G` is a graph object, `s` the starting vertex, `d` an instance of distance decorator, and `p` an instance of predecessor decorator.

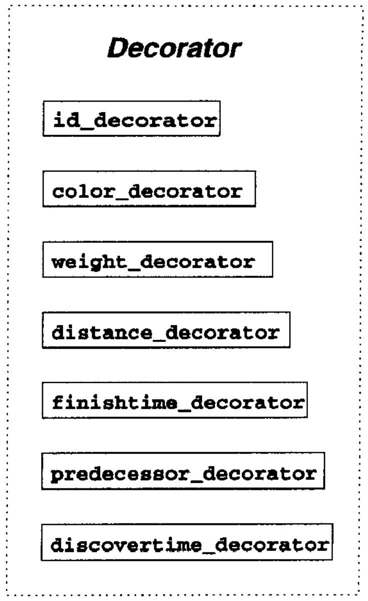

Figure 12: The predefined models of Decorator in GGCL.

5 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 and high-level performance optimization techniques (which themselves can be expressed in a generic fashion) 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 [19].

For many of the efficient graph data structures in GGCL, vertex and edge objects that model the GGCL interface concepts are not explicitly stored. Rather, only partial information is stored. The GGCL interface layer constructs full vertex and edge objects on the fly from this information. These objects are extremely lightweight, and have been designed so that a modern C++ compiler will optimize the small objects away altogether. [2]

Additionally, the flexibility within the GGCL is derived exclusively from static polymorphism, 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.

5.1 Comparison to General Purpose Libraries

Using a concise predefined implementation of adjacency list graph representation in GGCL following the concepts we described in Section 4, we compare the performance of `bfs`, `dfs`, and `dijkstra` algorithms with those in LEDA(version 3.8), a popular object-oriented graph library [14], and those in GTL [5]. We did not

[2]We call a light-weight object such as this a Mayfly because of its very short lifetime. We discuss the Mayfly as a design pattern for high performance computing in [20].

```
//Define a tag for the custom graph representation.
struct my_graphrep_tag { };

template < class stored_edge >
class graph_representation_gen< stored_edge, my_graphrep_tag > {
  typedef std::list<pair<size_t, stored_edge> > EdgeList;
  typedef EdgeList::iterator Iter1D;
  typedef std::vector<EdgeList>::iterator Iter2D;
public:
  typedef adjacency_list<my_graphrep_tag> rep_tag;
  typedef vector<EdgeList> graphrep_type;

  static Iter2D get_target(Iter2D b, Iter1D i)
    { return b + (*i).first; }
  static stored_edge* get_edge(Iter1D i)
    { return &((*i).second); }
  static bool add(EdgeList& elist, size_t vertex_num, const stored_edge& e);
  static void remove(EdgeList& elist, size_t vertex_num);
};

//Use the above representation to create a graph type.
typedef graph< adjacency_list< my_graphrep_tag > >  Graph;
```

Figure 11: An example of constructing a GGCL Graph out of STL vector components.

perform comparison between GGCL and Combinatorica [21] we mentioned previously because it is written in Mathematica.

Our experiments compare the performance of three algorithms: `bfs`, `dfs`, and `dijkstra`. The `bfs` algorithm calculates the distance and the predecessor for every reachable vertex from a starting vertex. The `dfs` algorithm calculates the discovery time and finishing time of vertices. The `dijkstra` algorithm calculates the distance and the predecessor of every vertex from a starting vertex.

Figure 14, Figure 15 and Figure 16 show the results for those algorithms applied to randomly generated graphs having a varying number of edges and a varying number of vertices. Because of lacking Dijkstra'a algorithm in GTL, it is not in Figure 16. 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.

5.2 Comparison to Special Purpose Library

In additionn, we demonstrate the performance of a GGCL-based implementation of the multiple mininum degree algorithm [13] using selected matrices from the Harwell-Boeing collection [9] and the University of Florida's sparse matrix collection [2]. Our tests compare the execution time of our implementation against that of the equivalent SPARSPAK Fortran algorithm (GENMMD) [7]. For each case, our implementation and GENMMD produced identical orderings. Note that the performance of our implementation is essentially equal to that of the Fortran implementation and even surpasses the Fortran implementation in a few cases.

5.3 Template Issues

There are several issues that often come up in libraries that make heavy use of C++ templates and advanced language features, such as code size, compile times, ease of debugging, and compiler portability. For template libraries such as GGCL, code size is very much

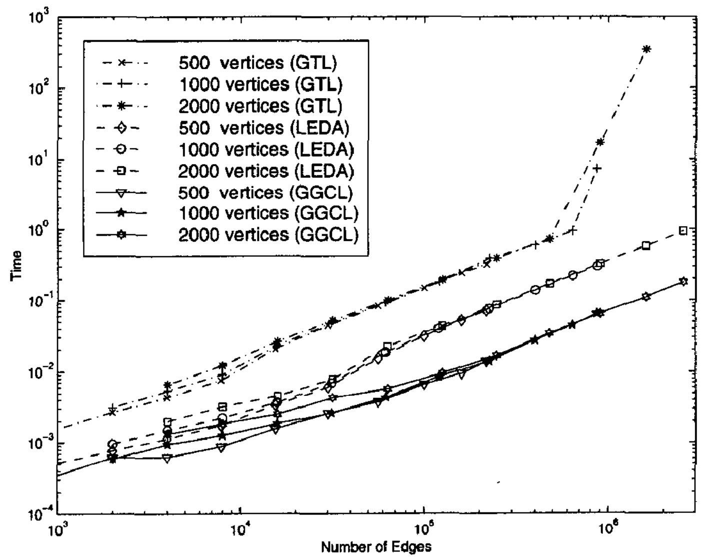

Figure 14: Performance comparison of the `bfs` algorithm in GGCL with that in LEDA and in GTL. Every curve represents a graph with fixed number of vertices and with varied number of edges.

```
template < class ColorDecorator, class Base = null_visitor >
class bfs_visitor : public Base {
  typedef typename ColorDecorator::return_type color_type;
public:
  // constructors . . .

  template <class Vertex>
  void initialize(Vertex u) {
    color[u] = color_traits<color_type>::white();
    Base::initialize(u);
  }

  template <class Vertex>
  void start(Vertex u) {
    color[u] = color_traits<color_type>::gray();
    Base::start(u);
  }

  template <class Vertex>
  void finish(Vertex u) {
    color[u] = color_traits<color_type>::black();
    Base::finish(u);
  }

  template <class Edge>
  bool process(Edge e) {
    typedef Edge::vertex_type Vertex;
    Vertex v = e.target();
    if ( is_undiscovered(v) ) {
      color[v] = color_traits<color_type>::gray();
      Base::process(e);
      return true;
    }
    return false;
  }

  template <class Vertex>
  bool is_undiscovered(Vertex u) {
    return (color[u] == color_traits<color_type>::white());
  }
protected:
  ColorDecorator color;
};
```

Figure 13: An example model of the Visitor concept.

Matrix	n	nnz	GENMMD	GGCL
BCSPWR09	1723	2394	0.00728841	0.007807
BCSPWR10	5300	8271	0.0306503	0.033222
BCSSTK15	3948	56934	0.13866	0.142741
BCSSTK18	11948	68571	0.251257	0.258589
BCSSTK21	3600	11500	0.0339959	0.039638
BCSSTK23	3134	21022	0.150273	0.146198
BCSSTK24	3562	78174	0.0305037	0.031361
BCSSTK26	1922	14207	0.0262676	0.026178
BCSSTK27	1224	27451	0.00987525	0.010078
BCSSTK28	4410	107307	0.0435296	0.044423
BCSSTK29	13992	302748	0.344164	0.352947
BCSSTK31	35588	572914	0.842505	0.884734
BCSSTK35	30237	709963	0.532725	0.580499
BCSSTK36	23052	560044	0.302156	0.333226
BCSSTK37	25503	557737	0.347472	0.369738
CRYSTK02	13965	477309	0.239564	0.250633
CRYSTK03	24696	863241	0.455818	0.480006
CRYSTM03	24696	279537	0.293619	0.366581
CT20STIF	52329	1323067	1.59866	1.59809
PWT	36519	144794	0.312136	0.383882
SHUTTLE_EDDY	10429	46585	0.0546211	0.066164
NASASRB	54870	1311227	1.34424	1.30256

Table 6: Test matrices and ordering time in seconds, for GENMMD (Fortran) and GGCL (C++) implementations of minimum degree ordering. Also shown are the matrix order (n) and the number of off-diagonal non-zero elements (nnz).

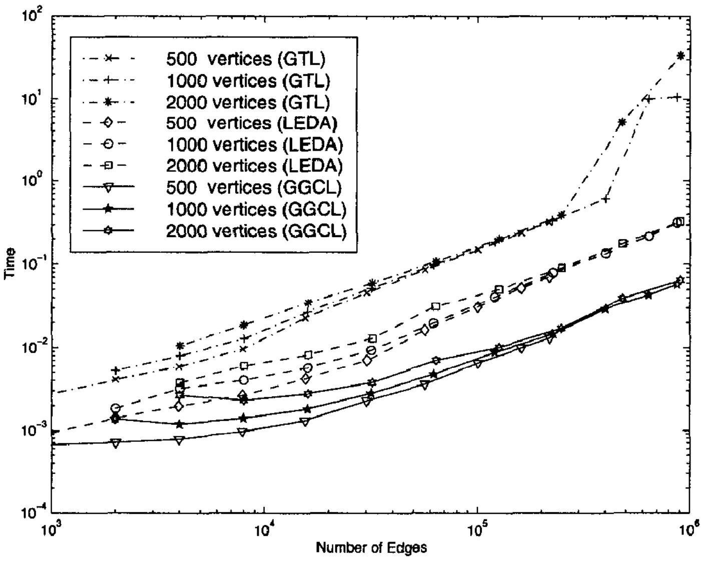

Figure 15: Performance comparison of the `dfs` algorithm in GGCL with that in LEDA and in GTL. Every curve represents a graph with fixed number of vertices and with varied number of edges.

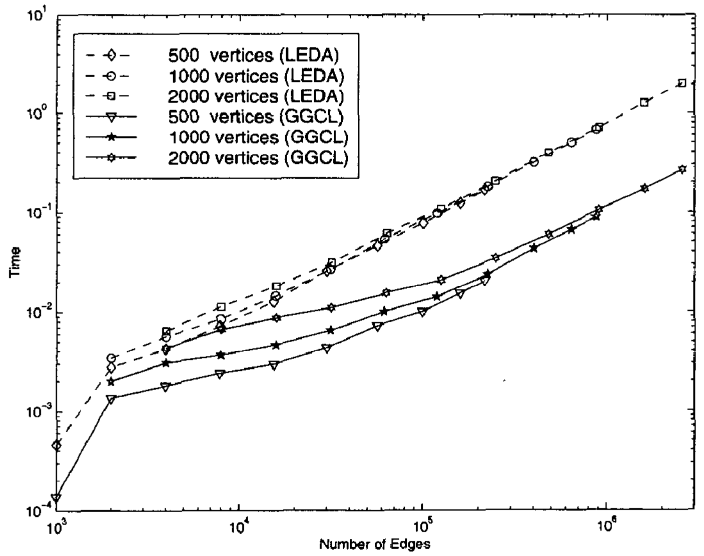

Figure 16: Performance comparison of the `dijkstra` algorithm in GGCL with that in LEDA. Every curve represents a a graph with fixed number of vertices and with varied number of edges.

dependent on how the library is used. If a particular code only uses a few GGCL algorithms and graph types, then the executable size will actually be much smaller than it would be using typical libraries. With a template library, only the functions that are actually used are included. On the other hand, with a traditional library, the whole object module will be linked in even though only one function in the module may be used. To demonstrate these effects, we compare the size of sample executables of `bfs`, `dfs`, and `dijkstra` algorithms in GTL, LEDA, and GGCL in Table 7. All are compiled by egcs1.1.2 using the same compilation options. (Similar results are obtained for other compilers and architectures.) Of course, with a template library like GGCL it is very easy to instantiate redundant functionality which may unnecessarily increase the executable size, so users with large projects should be cognizant of this issue. There are techniques one can use to reduce this effect by explicity instantiating template functions in object files that can be shared.

	Executable Size (KBytes)		
Package Name	`bfs`	`dfs`	`dijkstra`
GTL	151	151	/
LEDA	842	841	857
GGCL	33	30	30

Table 7: Comparison of executable sizes for `bfs`, `dfs`, and `dijkstra` implemented with GTL, LEDA and GGCL.

Long compilation times are often cited as a drawback to template libraries, especially those that use expression templates [23]. Since GGCL does not use expression templates, and the overall code size of GGCL is moderate, we have not experienced severe problems in this regard. In addition, many compilers provice precompiled header mechanisms to improve compile times for template libraries such as GGCL.

Another concern for users of template libraries are the almost impenetrable error messages that occur when the library is misused (e.g., when a template parameter type does not model the appropriate concept). We haved recently addressed this problem with some template techniques that cause the arguments to a library call to be checked up front with regards to the type requirements. With this mechanism the resulting error messages are much more informative.

Lastly, compiler portability is currently an issue for libraries that use the more advanced features of C++. GGCL currently compiles with egcs, Metrowerks CodeWarrior, Intel C++, SGI MIPSpro, KAI C++, and other Edison Design Group based compilers. We foresee some difficulty porting to Visual C++ because of its lack of standards conformance. Since the C++ standard has been finalized, we fully expect that language conformance problems will cease to be a significant issue in the near future.

6 CONCLUSION

In this paper, we applied the emerging paradigm of generic programming to the important problem domain of graphs and graph algorithms. Our resulting framework, the Generic Graph Component Library, is a collection of generic algorithms and data structures that interoperate through the abstract graph interface comprised of Vertex, Edge, Visitor, and Decorator concepts. The generic GGCL algorithms allow basic algorithm patterns to be applied in different ways to build up more complicated graph algorithms, resulting in significant code reuse. Similarly, since GGCL algorithms are independent of the underlying graph representation, custom graph representation implementation can be mixed and matched with GGCL graph algorithms. Since our C++ implementation of the generic programming paradigm makes heavy use of static (compile-time) polymorphism, there is no run-time overhead associated with the powerful abstractions provided by GGCL. Experimental results demonstrate that the GGCL executes significantly faster than LEDA, a well-known object-oriented graph library, and can even compete with high performance Fortran codes.

Current work with the GGCL focuses on the implementation and inclusion into GGCL of other important (classical) algorithms. In addition, we are extending the GGCL based on application-specific needs. For instance, one of the motivations behind the development of GGCL was the need for highly-efficient graph-based algorithms for sparse matrix orderings in the Matrix Template Library.

7 AVAILABILITY

The source code and complete documentation for the GGCL can be downloaded from the GGCL home page at

`http://lsc.nd.edu/research/ggcl/`

ACKNOWLEDGMENTS

This work was supported by NSF grants ASC94-22380 and CCR95-02710.

REFERENCES

[1] Netlib repository. http://www.netlib.org/.

[2] University of Florida sparse matrix collection. http://www-pub.cise.ufl.edu/~davis/sparse/.

[3] Austern, M. H. *Generic Programming and the STL*. Addison Wesley Longman, Inc, October 1998.

[4] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. The MIT Press, 1990.

[5] Forster, M., Pick, A., and Raitner, M. *Graph Template Library*. http://www.fmi.uni-passau.de/Graphlet/GTL/.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addiaon Wesley Publishing Company, October 1994.

[7] George, A., and Liu, J. W. H. User's guide for SPARSPAK: Waterloo sparse linear equations packages. Tech. rep., Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1980.

[8] George, A., and Liu, J. W.-H. *Computer Solution of Large Sparse Positive Definite Systems*. Computational Mathematics. Prentice-Hall, 1981.

[9] Grimes, R. G., Lewis, J. G., and Duff, I. S. User's guide for the harwell-boeing sparse matrix collection. User's Manual Release 1, Boeing Computer Services, Seattle, WA, October 1992.

[10] Jacobson, I., Booch, G., and Rumbaugh, J. *Unified Software Development Process*. Addison-Wesley, 1999.

[11] Knuth, D. E. *Stanford GraphBase: a platform for combinatorial computing*. ACM Press, 1994.

[12] Lee, M., and Stepanov, A. The standard template library. Tech. rep., HP Laboratories, February 1995.

[13] Liu, J. W. H. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transaction on Mathematical Software 11*, 2 (1985), 141–153.

[14] Mehlhorn, K., and Naeher, S. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[15] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[16] Object Management Group. *UML Notation Guide*, version 1.1 ed., September 1997. http://www.rational.com/uml/.

[17] Saad, Y. *Iterative Methods for Sparse Minear System*. PWS Publishing Company, 1996.

[18] Samaragdakis, Y., and Batory, D. Implementing layered designs with mixin layers. In *The Europe Conference on Object-Oriented Programming* (1998).

[19] Siek, J. G., and Lumsdaine, A. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments* (1998).

[20] Siek, J. G., and Lumsdaine, A. Mayfly: A pattern for light-weight generic interfaces. In *PLOP99* (1999). Accepted.

[21] Skiena, S. *Implementing Discrete mathematics*. Addion-Wesley, 1990.

[22] Skiena, S. S. *The Algorithm Design Manual*. Springer-Verlag New York, Inc, 1998.

[23] Veldhuizen, T. L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.