The Generic Graph Component Library

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

The Standard Template Library [14] has established a solid foundation for the development of reusable algorithms and data structures in C++. It has provided us with both a way to *think* about designing reusable components (*generic programming*), and it has demonstrated the programming techniques necessary to build efficient implementations. However, there are many problem domains beyond those addressed by the STL; consequently, there is a vast amount of work yet to be done in the area of generic library development.

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 numbers 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 of 1999 we did a survey of existing graph libraries. Some of the libraries we looked at were LEDA [16], the Graph Template Library (GTL) [5], Combinatorica [22], and Stanford GraphBase [10]. We also looked at repositories of graph algorithms such as Netlib [2] and [23]. 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* [3] 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) [12, 13], drawing on previous experience with the development of the Matrix Template Library [20, 21]. The most important aspect of designing the library was to define an interface that allowed for the maximum flexibility. The following is a list of our initial the 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 (i.e., the way functors make STL algorithms extensible).

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

Figure 1: Pseudo-code for the depth-first search algorithm.

- 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, it is important to start by analyzing how the data structures are typically used in the algorithms [18]. Figures 1, 2, 3, and 4 show the pseudo-code for some simple algorithms that demonstrate the different ways that graph structures are typically accessed. [1] 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.[2]
- Access the edge (if it exists) connecting vertex u and v.

The pseudo-code 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();
  // ...
};
```

[1]The initialization step has been omitted from the algorithms. Also, the depth-first search algorithm does not show the operations that would be invoked to do some useful task during the search.

[2]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.

```
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;
  }
}
```

Figure 2: Pseudo-code for Dijkstra's shortest paths algorithm.

```
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;
}
```

Figure 3: Pseudo-code for Bellman-Ford shortest paths.

However, in the interests 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 requirements that **any** type must meet to be considered a graph. As with the iterators of the STL [3], the requirements are expressed as a set of valid expressions and associated types (which are accessed through a traits class [17]). 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 iterators of the STL, a concept can be broken up into several categories, such as InputIterator and RandomAccessIterator. These categories factor the requirements into smaller bundles so that an algorithm can succinctly state what its requirements are, without overstating the requirements. An example of the need for this is given in Figure 5.

We have factored the graph concepts into seven sub-concepts, which are shown in Figure 6. 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 shown in Table 2. The notation used in the requirements table is shown in Table 1. There are several references in the requirements table to concepts such as Assignable.

```
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];
      }
}
```

Figure 4: Pseudo-code for part of an all-pairs shortest paths.

```
// 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++;
}
```

Figure 5: Example of overspecified requirements.

Their definitions can be found in the on-line documentation for the SGI STL implementation [1] or in the book by Austern [3]. 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 `it1 == it2` and `it1` is dereferenceable then `++it1 == ++it2`. 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 access various properties associated with the objects of a graph to achieve their task. 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 which are computed when needed like the distance between two vertices in a graph with Euclidian distances. To relieve generic algorithms from the

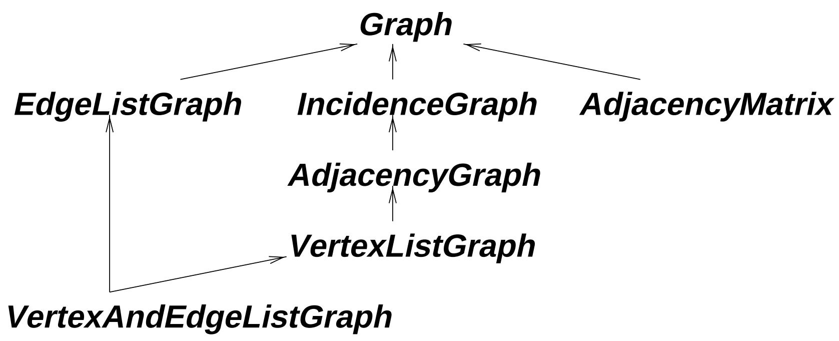

Figure 6: The Graph Concept Refinement Hierarchy.

`G`	is a Graph type.
`E`	is the edge descriptor type for `G`.
`V`	is the vertex descriptor type for `G`.
`AdjIt`	is the adjacency iterator type for `G`.
`IncIt`	is the incidence iterator type for `G`.
`VIt`	is the vertex iterator type for `G`.
`EIt`	is the edge iterator type for `G`.
`g`	is an object of type `G`.
`u,v`	are objects of type `V`.
`e`	is an object of type `E`.

Table 1: Notation for Graph Requirements.

Graph		
`graph_traits<G>::vertex_descriptor`		A vertex descriptor corresponds to a unique vertex in an abstract graph instance. The VertexDescriptor concept refines DefaultConstructible, Assignable, and EqualityComparable.
`graph_traits<G>::edge_descriptor`		An edge descriptor corresponds to a unique edge (u,v) in a graph. The EdgeDescriptor concept refines DefaultConstructible, Assignable, and EqualityComparable
IncidenceGraph refines Graph		
`graph_traits<G>::incidence_iterator`		The value type of an incidence iterator is the edge descriptor type of its graph.An incidence iterator must meet the requirements of MultiPassInputIterator.
`target(e,g)`	`V`	Returns the vertex descriptor for v of the edge (u,v) represented by `e`.
`out_edges(v,g)`	`std::pair<IncIt,IncIt>`	Returns an iterator-range providing access to the out-edges of vertex `v` in graph `g`.
AdjacencyGraph refines IncidenceGraph		
`graph_traits<G>::adjacency_iterator`		The value type of an adjacency iterator is the vertex descriptor type of its graph. An adjacency iterator must meet the requirements of MultiPassInputIterator.
`adj(v,g)`	`std::pair<AdjIt,AdjIt>`	Returns an iterator-range providing access to the vertices adjacent to vertex `v` in graph `g`. [3]
VertexListGraph refines AdjacencyGraph		
`graph_traits<G>::vertex_iterator`		A vertex iterator type must meet the requirements of InputIterator. The value type of the vertex iterator must be the vertex descriptor of the graph.
`vertices(g)`	`std::pair<VIt,VIt>`	Returns an iterator-range providing access to all the vertices in the graph`g`.
EdgeListGraph refines Graph		
`graph_traits<G>::edge_iterator`		An edge iterator type must meet the requirements of an InputIterator.The value type of the edge iterator must be the same as the edge descriptor of the graph.
`source(e,g)`	`V`	Returns the vertex descriptor for u of the edge (u,v) represented by `e`.
`target(e,g)`	`V`	Returns the vertex descriptor for v of the edge (u,v) represented by `e`.
`edges(g)`	`std::pair<EIt,EIt>`	Returns an iterator-range providing access to all the edges in the graph`g`.
AdjacencyMatrix refines Graph		
`edge(u,v,g)`	`std::pair<bool,E>`	Returns a pair consisting of a flag saying whether there exists an edge between `u` and `v` in graph `g`, and consisting of the edge descriptor if the edge was found.

Table 2: Requirements for the Graph Traversal Concepts.

details of the underlying property representation, the *property accessor*[4] abstraction is introduced.

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 bread-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 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.

The requirements for the respective categories of property accessors are listed in Table 4. The notation used in the property accessor table is listed in Table 3. 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 which 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 the utmost 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.

`PA`	is the type of a property accessor.
`T`	is the value type of the property accessed with PA objects.
`pa`	is a property accessor object of type `PA`.
`it`	is a suitable object identifier for property accessors of type `PA`.
`val`	is an object of type `T`.

Table 3: Notation for Property Accessor Requirements.

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. [5] We use the name GraphVisitor 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 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 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 GraphVisitor concept plays an important role in the GGCL algorithms, which are discussed in the next paragraph.

[4] In previous papers describing GGCL, the property accessor concept was named Decorator and used `operator[]` instead of `set()` and `get()`. In Dietmar's Master's thesis [11], property accessors are called data accessors.

[5] As of the time of writing the GraphVisitor concept has not been considered for part of the Boost graph interface.

ReadablePropertyAccessor		
`property_traits<PA>::value_type`		the type of the property.
`property_traits<PA>::category`		the category of the property accessor: a type convertible to `read_only_property_accessor_tag`.
`get(pa,it)`	T	read the property of the object identified by `it`.
WriteablePropertyAccessor		
`property_traits<PA>::value_type`		the type of the property.
`property_traits<PA>::category`		the category of the property: a type convertible to `write_only_property_accessor_tag`
`set(pa, it, val)`	`void`	write the property of the object identified by it
ReadWritePropertyAccessor refines the Readable and Writeable property accessors.		
LvaluePropertyAccessor refines ReadWritePropertyAccessor		
`property_traits<PA>::category`		the category of the property: a type convertible to `lvalue_property_accessor_tag`
`at(pa, it)`	`T&`	obtain a reference to the property

Table 4: Requirements for the PropertyAccessor Concepts.

GraphVisitor		
`v.initialize(u)`	`void`	Invoked during initialization.
`v.start(s)`	`void`	Invoked at the beginning of algorithms.
`v.discover(u, bad)`	`void`	Invoked when an undiscovered vertex is encountered.
`v.finish(u)`	`void`	Invoked when algorithms finish visiting a vertex.
`v.process(e, g, bag)`	`bool`	Invoked when an edge is traversed.

Table 5: Requirements for the GraphVisitor concept.

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 is the Self Avoiding Walk (SAW) used in adaptive mesh refinement algorithms [8]. The list is given in Figure 6.

Algorithm	**Graph Category**
Generalized Graph Search	IncidenceGraph
Breadth-First Search	IncidenceGraph
Depth-First Search	VertexListGraph
Dijkstra's Single-Source Shortest Paths	IncidenceGraph
Minimum Spanning Tree (Prim's and Kruskal's)	IncidenceGraph
Topological sort	VertexListGraph
Connected Components	VertexListGraph or EdgeListGraph
Strongly Connected Components	VertexListGraph
Reverse Cuthill-McKee	VertexListGraph
Modified Minimum Degree	AdjacencyGraph
Self Avoiding Walks	IncidenceGraph

Table 6: The GGCL Algorithms

Perhaps one of the most exciting aspects of the GGCL implementation (at least from the implementors 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 Figure 7. [6]

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 Figure 8, and the `coloring_visitor` in Figure 9. The template argument `Super` in the `coloring_visitor` may appear somewhat mystifying at first. We are using the mixin [19] 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 textbooks [4], a topological sort can be accomplished with depth-first search, where each vertex is output as it is "finished". The GGCL algorithm is coded exactly in this way, as shown in Figure 10.

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>` (where `EdgeList` is a template argument that must be some STL-conformant Container with vertex ID's as the `value_type`[7]) into a graph type without defining a "wrapper class".

[6]The `tie()` function used in `graph_search()` is a nice utility function created by Jaakko Järvi [9] that allows the values of a pair (or n-tuples) to be assigned to variables in a way reminiscent of the language ML.

[7]The requirement for the `value_type` to be a vertex ID is not really required as long as the data structure implementor provides a `get_target()` function. This makes it possible to store additional edge properties in the `EdgeList` container.

```
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();
    }
  }
}
```

Figure 7: The GGCL generalized graph search.

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. Figure 11 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 [20].

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 [16], and those in GTL [5].

Figure 12 show the results of the BFS applied to randomly generated graphs having a varying number

```
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));
}
```

Figure 8: The breadth and depth-first GGCL implementations.

of edges and a varying number of vertices. Note that 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 [15] 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) [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.

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

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

The graph interface described in this article can be found at the Boost home page at

`http://www.boost.org`

References

[1] SGI STL Web site. http://www.sgi.com/Technology/STL.

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

Matrix	n	nnz	GENMMD	GGCL
BCSSTK18	11948	68571	0.251257	0.258589
BCSSTK23	3134	21022	0.150273	0.146198
BCSSTK29	13992	302748	0.344164	0.352947
BCSSTK31	35588	572914	0.842505	0.884734
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
NASASRB	54870	1311227	1.34424	1.30256

Table 7: 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).

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

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

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

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addiaon Wesley Publishing Company, October 1994.

[7] Alan George 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, Canada, 1980.

[8] G. Heber, R. Biswas, and G.R. Gao. Self-avoiding walks over adaptive unstructured grids. In *Parallel and Distributed Processing*, number 1586 in LNCS, pages 968–977. Spriger-Verlag, 1999.

[9] Jaakko Järvi. Ml-style tuple assignment in standard C++ - extending the multiple return value formalism. Technical Report 267, TUCS, April 1999.

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

[11] Dietmar Kühl. Design patterns for the implementation of graph algorithms. Master's thesis, Technische Universität Berlin, July 1996.

[12] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE '99*, 1999.

[13] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *OOPSLA '99*, November 1999.

[14] Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.

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

[16] Kurt Mehlhorn and Stefan Naeher. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[17] Nathan Myers. A new and useful template technique: “Traits”. *C++ Report*, 7(5):32–35, June 1995.

[18] Graziano Lo Russo. An interview with alexander stepanov. *Edizioni Infomedia srl*, 1997.

[19] Yannis Samaragdakis and Don Batory. Implementing layered designs with mixin layers. In *The Europe Conference on Object-Oriented Programming*, 1998.

[20] Jeremy G. Siek and Andrew Lumsdaine. 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.

[21] Jeremy G. Siek and Andrew Lumsdaine. *Modern Software Tools in Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Birkhauser, 1999.

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

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

```
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);
}
```

Figure 9: The coloring visitor for BFS and DFS.

```
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;
};
```

Figure 10: The GGCL topological sort implementation.

```
#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()));
```

Figure 11: An example of an STL-based (and GGCL augmented) adjacency list.

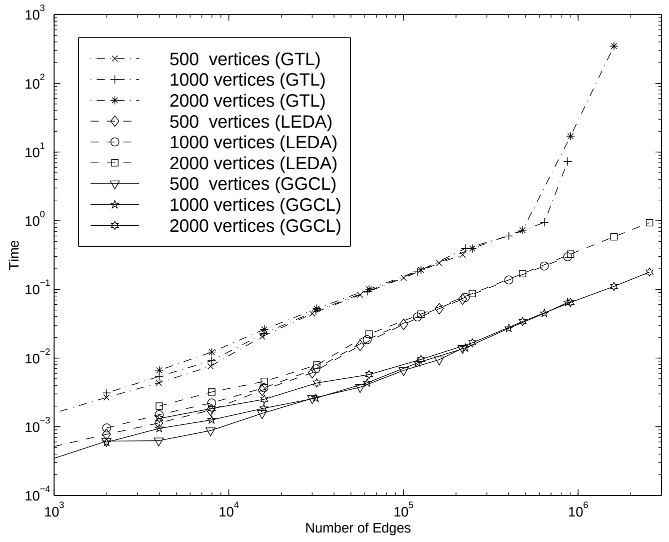

Figure 12: Performance comparison of the `bfs` algorithm in GGCL with that in LEDA and in GTL. Every curve represents a graph with fixed number of vertices and with varied number of edges.