Generic Graph Algorithms for Sparse Matrix Ordering*

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,
{llee1,jsiek,lums}@lsc.nd.edu,
WWW home page: http://lsc.nd.edu/~lsc

Abstract. Fill-reducing sparse matrix orderings have been a topic of active research for many years. Although most such algorithms are developed and analyzed within a graph-theoretical framework, for reasons of performance, the corresponding implementations are typically realized with programming languages devoid of language features necessary to explicitly represent graph abstractions. Recently, generic programming has emerged as a programming paradigm capable of providing high levels of performance in the presence of programming abstractions. In this paper we present an implementation of the Minimum Degree ordering algorithm using the newly-developed Generic Graph Component Library. Experimental comparisons show that, despite our heavy use of abstractions, our implementation has performance indistinguishable from that of the Fortran implementation.

1 Introduction

Computations with symmetric positive definite sparse matrices are a common and important task in scientific computing. For efficient matrix factorization and linear system solution, the ordering of the equations plays an important role. Because Gaussian elimination (without numerical pivoting) of symmetric positive definite systems is stable, such systems can be ordered before factorization takes place based only on the structure of the sparse matrix. Unfortunately, determining the optimal ordering (in the sense of minimizing fill-in) is an NP-complete problem [1], so greedy heuristic algorithms are typically used instead.

The development of algorithms for sparse matrix ordering has been an active research topic for many years. The algorithms are typically developed in graph-theoretical terms, while the most widely used implementations are coded in Fortran 77. Since Fortran 77 supports no abstract data types other than arrays, the graph abstractions used to develop and describe the ordering algorithms must be discarded for the actual implementation. Although graph algorithms are well-developed and widely-implemented in higher-level languages such as C or C++, performance concerns (which are often paramount in scientific computing) have continued to restrict implementations of sparse matrix ordering algorithms to Fortran.

Efforts to develop sparse matrix orderings with modern programming techniques include [2] and [3]. These were based on an object-oriented, rather than generic, programming paradigm and although they were well programmed, the reported performance was still a factor of 4-5 slower than Fortran 77 implementations.

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

The recently introduced programming paradigm known as *generic programming* [4, 5] has demonstrated that abstraction and performance are not necessarily mutually exclusive. One example of a graph library that incorporates the generic programming paradigm is the recently developed Generic Graph Component Library (GGCL) [6]. In this paper we present an implementation of the minimum degree algorithm for sparse matrix ordering using the GGCL. Although the implementation uses powerful graph abstractions, its performance is indistinguishable from that of one of the most widely used Fortran 77 codes.

The rest of this paper is organized as follows. We provide a brief overview of generic programming and the Generic Graph Component Library in Sections 2 and 3. Algorithms for sparse matrix ordering are reviewed in Section 4 and our implementation of the Minimum Degree algorithm is given in Section 5 along with performance results in Section 6.

2 Generic Programming

Recently, generic programming has emerged as a powerful new paradigm for software development, particularly for the development of (and use of) component libraries. The most visible (and perhaps most important) popular example of generic programming is the celebrated Standard Template Library (STL) [7]. 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 STL the data structures are *containers* such as vectors and linked lists and *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 being accessed, but on the type of operation that is applied during the traversal of a container as well. For example, the `transform()` algorithm has a parameter for a `UnaryOperator` *function object* (aka "functor"). Finally, STL contains classes known as *adaptors* that are used to modify underlying class interfaces.

The generic programming approach to software development can provide tremendous benefits to such aspects of software quality as functionality, reliability, usability, maintainability, portability, and efficiency. The last point, efficiency, is of particular (and sometimes paramount) concern in scientific applications. Performance is often of such importance to scientific applications that other aspects of software quality may be deliberately sacrificed if nice programming abstractions and high performance cannot be simultaneously achieved. Until quite recently, the common wisdom has been that high levels of abstraction and high levels of performance were, *per se*, mutually exclusive. However, beginning with STL for general-purpose programming, and continuing with the Matrix Template Library (MTL) [5] for basic linear algebra, it has been clearly demonstrated that abstraction does not necessarily come at the expense of performance. In fact, MTL provides performance equivalent to that of highly-optimized vendor-tuned math libraries.

3 The Generic Graph Component Library

The Generic Graph Component Library (GGCL) is a collection of high-performance graph algorithms and data structures, written in C++ using the generic programming style. Although the domain of graphs and graph algorithms is a natural one for the application of generic programming, 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.

3.1 Graph Concepts

The graph interface used by GGCL can be derived directly from the formal definition of a graph [8]. 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 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.

To describe the graph interface of GGCL we use generic programming terminology from the SGI STL [4]. 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*. The various classes that fulfill the requirements of a concept are said to be *models* of the concept. Concepts can extend other concepts, which is referred to as *refinement*. We use a bold sans serif font for all concept identifiers.

The three main concepts necessary to define our graph are Graph, Vertex, and Edge. The abstract iterator interface used by STL is not sufficiently rich to encompass the numerous ways that graph algorithms may compute with a graph. Instead, we formulate an abstract interface, based on Visitor and Decorator concepts, that serves the same purpose for graphs that iterators do for basic containers. These two concepts are similar in spirit to the “Gang of Four” [9] patterns of the same name, however the implementation techniques used are based on static polymorphism and mixins [10] instead of dynamic polymorphism. Fig. 1 depicts the analogy between the STL and the GGCL.

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. The only requirement is the *vertex set* be a model of Container and its `value_type` a model of Vertex. The *edge set* must be a model of Container and its `value_type` a model of Edge.

Vertex: The Vertex concept provides access to the adjacent vertices, the out-edges of the vertex and optionally the in-edges.

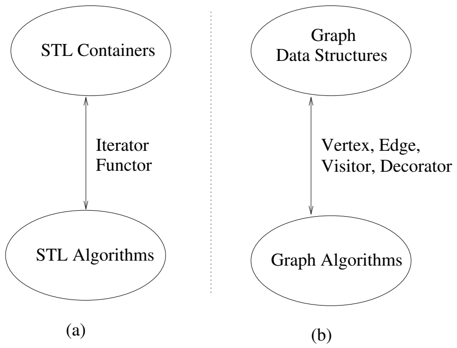

Fig. 1. The analogy between the STL and the GGCL.

Edge: An Edge is a pair of vertices, one is the *source* vertex and the other is the *target* vertex. In the unordered case it is just assumed that the position of the *source* and *target* vertices are interchangeable.

Decorator: As mentioned in the introduction, we would like to have a generic way to access vertex and edge properties, such as color and weight, from within an algorithm. The generic access method is necessary because there are many ways in which the properties can be stored, and ways in which access to that storage is implemented. We give the name Decorator to the concept for this generic access method. The implementation of graph Decorators is similar in spirit to the GoF decorator pattern [9]. A Decorator is very similar to a functor, or function object. We use the `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 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, since we are basically just using a template version of the well known visitor pattern [9]. 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.

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, one in which there is a default ColorDecorator. The default decorator accesses the color property directly from the graph vertex. This is analogous to the STL algorithms. For example, there are two overloaded versions of the `lower_bound()` algorithm. One uses `operator<` by default and the other takes a BinaryOperator functor argument.

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

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

3.2 Generic Graph Algorithms

With the abstract graph interface defined, generic graph algorithms can be written solely in terms of the graph interface. The algorithms do not make any assumptions about the actual underlying graph data structure.

The Breadth First Search (BFS) algorithm, as an example, is shown in Fig. 2. In this algorithm we use the expression `u.out_edges()` to access the Container of edges leaving vertex `u`. We can then use the iterators of this Container to access each of the edges. In this algorithm, the Visitor is used to abstract the kind of operation performed on each edge as it is discovered. The algorithm also inserts each discovered vertex onto `Q`. The vertex is accessed through `e.target_vertex()`.

```
template <class Graph, class QType, class Visitor>
void generalized_BFS(Graph& G, Graph::vertex_type s,
                     QType& Q, Visitor visitor)
{
  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.visit(e))
        Q.push(e.target_vertex());
    }
    visitor.finish(u);
  }
}
```

Fig. 2. The generalized Breadth First Search algorithm.

The concise implementation of algorithms 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.

4 Sparse Matrix Ordering

The process for solving a sparse symmetric positive definite linear system, $Ax = b$, can be divided into four stages as follows:

Ordering: Find a permutation P of matrix A,

Symbolic factorization: Set up a data structure for Cholesky factor L of PAP^T,
Numerical factorization: Decompose PAP^T into LL^T,
Triangular system solution: Solve $(LL^T)Px = Pb$ for x.

Because the choice of permutation P will directly determine the number of fill-in elements (elements present in the non-zero structure of L that are not present in the non-zero structure of A), the ordering has a significant impact on the memory and computational requirements for the latter stages. However, finding the optimal ordering for A (in the sense of minimizing fill-in) has been proven to be NP-complete [1] requiring that heuristics be used for all but simple (or specially structured) cases.

Developing algorithms for high-quality orderings has been an active research topic for many years. Most ordering algorithms in wide use are based on a greedy approach such that the ordering is chosen to minimize some quantity at each step of a simulated n-step symmetric Gaussian elimination process. The algorithms using such an approach are typically distinguished by their greedy minimization criteria [11].

4.1 Graph Models

In 1961, Parter introduced the graph model of symmetric Gaussian elimination [12]. A sequence of elimination graphs represent a sequence of Gaussian elimination steps. The initial Elimination graph is the original graph for matrix A. The elimination graph of k's step is obtained by adding edges between adjacent vertices of the current eliminated vertex to form a clique, removing the eliminated vertex and its edges.

In graph terms, the basic ordering process used by most greedy algorithms is as follows:

1. *Start:* Construct undirected graph G^0 corresponding to matrix A
2. *Iterate:* For $k = 0, 1, \ldots$, until $G^k = \emptyset$ do:
   - Choose a vertex v^k from G^k according to some criterion
   - Eliminate v^k from G^k to form G^{k+1}

The resulting ordering is the sequence of vertices $\{v^0, v^1, \ldots\}$ selected by the algorithm.

One of the most important examples of such an algorithm is the *Minimum Degree* algorithm. At each step the minimum degree algorithm chooses the vertex with minimum degree in the corresponding graph as v^k. A number of enhancements to the basic minimum degree algorithm have been developed, such as the use of a quotient graph representation, mass elimination, incomplete degree update, multiple elimination, and external degree. See [13] for a historical survey of the minimum degree algorithm. Many of these enhancements, although initially proposed for the minimum degree algorithm, can be applied to other greedy approaches as well. Other greedy approaches differ from minimum degree by the choice of minimization criteria for choosing new vertices. For example, to accelerate one of the primary bottlenecks of the ordering process, the *Approximate Minimum Degree* (AMD) algorithm uses an estimate of the degree (or external degree) of a vertex [14]. The *Minimum Deficiency* class of algorithms instead choose the vertex that would create the minimum number of fill-in elements. A nice comparison of many of these different approaches can be found in [11].

5 Implementation

Our GGCL-based implementation of MMD closely follows the algorithmic descriptions of MMD given, e.g., in [15, 13]. The implementation presently includes the enhancements for mass elimination, incomplete degree update, multiple elimination, and external degree. In addition, we use a quotient-graph representation. Some particular details of our implementation are given below.

Prototype The prototype for our algorithm is

```
template<class Graph, class RandomAccessContainer,
         class Decorator>
void mmd(Graph& G, RandomAccessContainer& Permutation,
         RandomAccessContainer& InversePermutation,
         Decorator SuperNodeSize, int delta = 0)
```

The parameters are used in the following way.

`G` (input/output) is the graph representing the matrix A to be ordered on input. On output, `G` contains the results of the ordered elimination process. May be used subsequently by symbolic factorization.

`Permutation, InversePermutation` (output) respectively contain the permutation and inverse permutation produced by the algorithm.

`SuperNodeSize` (output) contains the size of supernodes or supernode representative node produced by the the algorithm. May be used subsequently by symbolic factorization.

`delta` (input) controls multiple elimination.

Abstract Graph Representation Our minimum degree algorithm is expressed only in terms of the GGCL abstract graph interface. Thus, any underlying concrete representation that models the GGCL Graph concept can be used. Not all concrete representations will provide the same levels of performance, however. A particular representation that offers high performance in our application is described below.

Concrete Graph Representation We use an adjacency list representation within the GGCL framework. In particular the graph is based on a templated “vector of vectors.” The vector container used is an adaptor class built on top the STL `vector` class. Particular characteristics of this adaptor class include the following:

- Erasing elements does not shrink the associated memory. Adding new elements after erasing will not need to allocate additional memory.
- Additional memory is allocated efficiently on demand when new elements are added (doubling the capacity every time it is increased). This property comes from STL vector.

We note that this representation is similar to that used in Liu's implementation, with some important differences due to dynamic memory allocation. With the dynamic memory allocation we do not need to over-write portions of the graph that have been eliminated, allowing for a more efficient graph traversal. More importantly, information about the elimination graph is preserved allowing for trivial symbolic factorization. Since symbolic factorization can be an expensive part of the entire solution process, improving its performance can result in significant computational savings.

The overhead of dynamic memory allocation could conceivably compromise performance in some cases. However, in practice, memory allocation overhead does not contribute significantly to run-time for our MMD implementation. Finally, with our approach, somewhat more total memory may be required for graph representation. In the context of the entire sparse matrix solution process this is not an important issue because the memory used for the graph during ordering can be returned to the system for use in subsequent stages (which would use more memory than even the dynamically-allocated graph at any rate).

6 Experimental Results

6.1 Test Matrices

We tested the performance of our implementation using selected matrices from the Harwell-Boeing collection [16], the University of Florida's sparse matrix collection [17], as well as locally-generated matrices representing discretized Laplacians.

For our tests, we compare the execution time of our implementation against that of the equivalent SPARSPAK algorithm (GENMMD). The tests were run on a Sun SPARC Station U-30 having a 300MHz UltraSPARC-II processor, 256MB RAM, and Solaris 2.6. The GENMMD code was compiled with Solaris F77 4.2 with optimizing flags `-fast -xdepend -xtarget=ultra2 -xarch=v8plus -xO4 -stackvar -xsafe=mem`. The C++ code was compiled with Kuck and Associates KCC version 3.3e using aggressive optimization for the C++ front-end. The back-end compiler was Solaris cc version 4.2, using optimizations basically equivalent to those given above for the Fortran compiler.

Table 1 gives the performance results. 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.

7 Future Work

The work reported here only scratches the surface of what is possible using GGCL for sparse matrix orderings (or more generally, using generic programming for sparse matrix computations). The highly modular nature of generic programs makes the implementations of entire classes of algorithms possible. For instance, a generalized greedy ordering algorithm (currently being developed) will enable the immediate implementation of most (if not all) of the greedy algorithms related to MMD (e.g., minimum deficiency). We are also working to develop super-node based sparse matrices as part of the Matrix Template Library and in fact to develop all of the necessary infrastructure for a complete generic high-performance sparse matrix package. Future work will extend these approaches from the symmetric positive definite case to the general case.

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
LA2D32	1024	1984	0.00489657	0.006476
LA2D64	4096	8064	0.022337	0.028669
LA2D128	16384	32512	0.0916937	0.119037
LA3D16	4096	11520	0.0765908	0.077862
LA3D32	32768	95232	0.87223	0.882814
PWT	36519	144794	0.312136	0.383882
SHUTTLE_EDDY	10429	46585	0.0546211	0.066164
NASASRB	54870	1311227	1.34424	1.30256

Table 1. 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).

References

1. M Yannanakis. Computing the minimum fill-in is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, 1981.
2. Kaixiang Zhong. A sparse matrix package using the standard template library. Master's thesis, University of Notre Dame, 1996.
3. Gary Kumfert and Alex Pothen. An object-oriented collection of minimum degree algorithms. In *Computing in Object-Oriented Parallel Environments*, pages 95–106, 1998.
4. Matthew H. Austern. *Generic Programming and the STL*. Addison Wesley Longman, Inc, October 1998.
5. Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In Denis Carmel, Rodney R. Oldhhoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, pages 59–70, 1998.
6. Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *OOPSLA'99*, 1999. Accepted.
7. Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.
8. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addiaon Wesley Publishing Company, October 1994.
10. Yannis Samaragdakis and Don Batory. Implementing layered designs with mixin layers. In *The Europe Conference on Object-Oriented Programming*, 1998.
11. Esmond G. Ng amd Padma Raghavan. Performance of greedy ordering heuristics for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, to appear.
12. S Parter. The use of planar graph in gaussian elimination. *SIAM Review*, 3:364–369, 1961.
13. Alan George and Joseph W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, March 1989.
14. Patrick Amestoy, Timothy A. Davis, and Iain S. Duff. An approximation minimum degree ordering algorithm. *SIAM J. Matrix Analysis and Applications*, 17(4):886–905, 1996.
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. Roger G. Grimes, John G. Lewis, and Iain S. Duff. User's guide for the harwell-boeing sparse matrix collection. User's Manual Release 1, Boeing Computer Services, Seattle, WA, October 1992.
17. University of florida sparse matrix collection. http://www-pub.cise.ufl.edu/ davis/sparse/.