DIMACS Technical Report 95-15
June 1995

LINK: A Combinatorics and Graph Theory Workbench for Applications and Research

by

Jon Berry[1], Nathaniel Dean[2], Patricia Fasel[3],
Mark Goldberg[4], Elizabeth Johnson[5], John MacCuish[6],
Gregory Shannon[7], and Steven Skiena[8]

[1]DIMACS (berryj@dimacs.rutgers.edu)
[2]AT&T Bell Laboratories
[3]Los Alamos National Laboratory
[4]Rensselaer Polytechnic Institute
[5]Indiana University
[6]Los Alamos National Laboratory
[7]InfoStructure Services & Technologies, Inc.
[8]State University of New York, Stony Brook

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

LINK is a set of C++ class libraries that supports applications in discrete mathematics. The libraries include a commandline interpreter and a graphical user interface that allow access to basic data structures such as *Sets* and *Lists*, and a graph hierarchy that includes undirected, directed, and "mixed" hypergraphs and graphs. A "mixed" graph may contain both directed and undirected edges. Many standard data structures including arrays, lists, heaps and binary search trees are within a *Container* hierarchy. *Sets* and *Sequences* are supported within a *Collection* hierarchy. The data structure hierarchies enable the user to experiment with competing data structure implementations, and with more complex and sophisticated data structures. If an algorithm has several possible choices of a data structure to be used, a single object can be created that is templated with the particular data structure desired. LINK also contains a set of graph generators, layout algorithms for hypergraphs and binary graphs, and numerous graph algorithms.

Interactive visualization of hypergraphs, graphs, and subgraphs is included in LINKGUI, the research tool application for combinatorics and graph theory built on top of the LINK libraries. LINKGUI is a collection of libraries that includes a Motif-based graphical user interface and Tcl-based command-line interface. The ability to select, contract and expand subgraphs and nested subgraphs in either hypergraphs or binary graphs is included in LINKGUI. LINKGUI enables a user to perform unary operations, such as complement, on a particular graph or to combine graphs via binary operations such as graph union or intersection. The resulting graphs can be displayed in multiple windows. Algorithms can be run on graphs where subgraphs have been contracted into single vertices. Other LINK applications can be developed by using various parts of the library code and adding code specific to the application. This new code can include an entirely different interface from that supplied by LINKGUI.

This report is also published as a technical report of The Los Alamos National Laboratory.

1 Introduction

Over the last 10 years there have been numerous attempts to develop software for discrete mathematics that would benefit pure research, application-driven research, and/or pedagogic concerns. This software has focused on combinatorics, graph theory, computational geometry, or some combination of these areas. However, each package has had shortcomings which has prevented it from becoming a generally useful tool. Some packages are too specialized, some are not robust, some are inefficient, and some are hard to learn. Nevertheless, the fact that these systems continue to be designed in spite of previous efforts indicates that there is a real need for a comprehensive and robust system which can serve a variety of users, from applications to research, to education. The magnitude of such a project exceeds the efforts made so far by the researchers and research groups responsible for the existing systems. A good discussion of software packages for discrete math is found in [11, 25, 19].

From the standpoint of a theoretician or an applications researcher, there are several features which characterize a useful software tool for their work. First, it is essential to have a set of libraries that are efficient, easy to use, robust, and extensible. Second, a sophisticated GUI and command-line interface are vital for interaction, visualization, testing, and experimentation. Third, having a large set of standard algorithms and data structures is important. Educators trying to produce the next generation of researchers would also find all of these features very useful. But even with the improvements in object-oriented design and GUI technology these goals present an enormous challenge. The LINK project is an attempt to balance the trade-offs inherent in these varied goals and produce a usable and reliable tool for theorists, applied researchers and educators.

This document is an overview of the history and structure of the LINK software system. It is intended to provide a general introduction to LINK and its applications. Specific details about the system are available in the LINK User Guide and Programmers Reference Manual (forthcoming). The remainder of this section discusses the history and motivation of the LINK project. In section 2, we discuss the organization of LINK. Section 3 presents the libraries that make up LINK. In section 4, the process of creating a LINK application is described along with several current applications. Finally, we discuss future work and present our conclusions in sections 5 and 6.

1.1 History and Motiviation

In July 1991, four researchers met at the SIAM conference for Applied Mathematics in Washington D.C. and discussed the possibility of creating a large software package for discrete mathematics. Each had already directed similar projects. Greg Shannon of Indiana University had organized the development of GraphLab [23], Nate Dean was largely responsible for the package NETPAD at Bellcore [20] , Steve Skiena at State University at New York at Stony Brook had produced Combinatorica on top of Mathematica [24], and Mark Goldberg and his students at Rensselaer Polytechnic Institute had implemented SetPlayer [5]. Though these packages have many useful features, some overlapping and some not, none was complete or without significant drawbacks depending on the user and the application. These researchers wanted a more comprehensive package that would add new features and retain the positive aspects of their previous efforts. They wanted a package that could easily be used for intensive research and applications, and yet have the capability to be used for pedagogic purposes. There would be numerous tradeoffs in trying to satisfy these somewhat competing goals. The hope was that they could find a satisfactory balance that would fulfill each of these goals in turn.

Later that year and into the next they, along with their students, did further research and wrote a proposal for a research grant to develop such a package. The grant stipulated that they would receive organizational support from DIMACS (Center for Discrete Mathematics and Theoretical Computer Science). The late Daniel Gorenstein joined the project as the principal investigator with co-PI's Nate Dean, Mark Goldberg, Greg Shannon and Steve Skiena.

Independently, at Los Alamos National Laboratory (LANL) there were several projects where such a package would be of considerable use, and researchers and programmers at LANL began implementing many components similar to software that the PI's had developed. Mark Goldberg was on sabbatical at LANL and was involved in one of these projects. He and the other PI's convinced Vance Faber (the group leader of CIC-3, the Computer Research and Applications Group of the Communications, Information, and Computing Division) that a more general tool would better fulfill the needs of these and similar projects at LANL. The grant was accepted by the National Science Foundation in the fall of 1992, and CIC-3 provided staff support and development resources in the form of support for the PIs' students to work on the project at LANL. In addition, Greg Shannon took a leave of absence from IU and worked on LINK at LANL as a contractor.

Initial work on LINK began with meetings in December 1992 between representatives from Indiana University, RPI, Stony Brook and LANL. Many months were spent in designing the graph hierarchy, and how that hierarchy would be represented to the user, and in the implementation of basic data structure classes. Algorithms were implemented using these designs in order to test the ease of programming and the efficiency of the design. Much time was spent dealing with the intricacies of C++ and its compiler immaturity.

In the summer of 1993, students from all the representative institutions worked together with staff from LANL at Los Alamos. During this time, the current *Graph* and *Collection* hierarchies were developed and much of the work on the Tcl interface was done. This command-line interface provided a means of testing the various generators, layouts and algorithms that had been implemented during the spring and summer months.

By January 1994, the LINKGUI interface had moved from XView code to Motif, and LINK was working with both the AT&T and GNU compilers. The latter was a special challenge since the two compilers process templates completely differently. Also at this time, the initial implementation of an attribute manager was completed. From this point, laboratory support for the project came through LINK's use in applications. By October 1994 LINK had its first two applications, presenting new challenges to its design. The applications were an interface to Mosaic for effective network navigation and a fraud detection project for the IRS (see Section 4.2 below).

2 Organization and Design

Much time was spent during the LINK project on design issues such as class hierarchies and library organization. Underlying everything is the goal to design classes and methods which can be used by researchers more familiar with combinatorics than programming. Another key idea is the building-block approach to libraries, which allows many types of applications to be built using the LINK libraries. The user interface was separated from the libraries in order to facilitate this variety of uses.

2.1 Language and Compiler Issues

It was decided during design meetings that the best object-oriented programming language for LINK development was C++. There were a number of tradeoffs in choosing C++. First, it is not as pure an object-oriented language as Smalltalk, for instance, nor as syntactically simple as Objective C, but it is well known and in widespread use. In addition, C++ is flexible and relatively efficient.

Initial work on LINK was done with the AT&T compiler version 3.0.1, which supported templates. In January 1994, an effort was made to move to the AT&T 3.0.2 version and to the GNU `g++` compiler since the latter compiler was more readily available to universities that would be using LINK. Currently, either the AT&T or GNU compiler can be used to compile LINK.

2.2 Components

LINK has evolved over several years into various distinct libraries. Applications can be built on top of some or all of the libraries as needed. The LINKGUI application (described in Section 4.1), for example, uses all of the LINK libraries. The libraries contain code for basic data structures, graph data structures, graph algorithms, graph generator algorithms, graph layout algorithms, and general interface structures.

The goal with this organization is a building-block approach to library usage. All of the libraries need the basic library, so the graph library is built on top of it, and the generator library on top of the graph library. The interface library is needed only if the application will have a GUI. If the interface library is present, then the layout library may be present. An application will include the LINK libraries that are needed and will supply a main program, specific interface code and specific application code.

The LINK basic library consists of classes which deal specifically with general purpose data structures such as lists, arrays and binary heaps. These basic classes are all templated and include classes that support generic *Sets* and *Sequences* and provide all of the standard operations on *Collections*.

The LINK graph library contains a hierarchy of graph classes which includes undirected, directed, and mixed versions of binary graphs and hypergraphs. A "mixed" graph is one which may contain both undirected and directed edges.

The LINK generator, layout and algorithm libraries include simple graph generators, graph layout algorithms, and standard graph algorithms respectively. The generators will return a graph of the requested type, while algorithms take a graph as an argument and operate on it. The layout library is used in conjunction with a GUI and provides functions to embed a graph in a particular way, such as in a circle or random layout.

The LINK interface library contains classes used to display graphs on the screen. LINK has general classes for drawing graphs which may work in conjunction with Motif code, or which may work with other graphical user interface code. The LINK Tcl library includes procedures defining a Tcl-based command-line interface. The Tcl interface presents users with the ability to run experiments with scripts. This is also very useful for testing.

3 LINK Libraries

3.1 Basic Library

The basic data structures are divided into two separate hierarchies. The first is the *Container* class hierarchy which consists of two sub-hierarchies: *SimpleContainer* and *Dictionary*. The former includes primitive data structure classes such as *List* and *Array*, while the latter is made up of classes such as *BinaryHeap*, where retrieval is by a separate key field. The *Container* class defines a standard class interface of data and methods, and may serve as the parent of any data structure implementation following this standard.

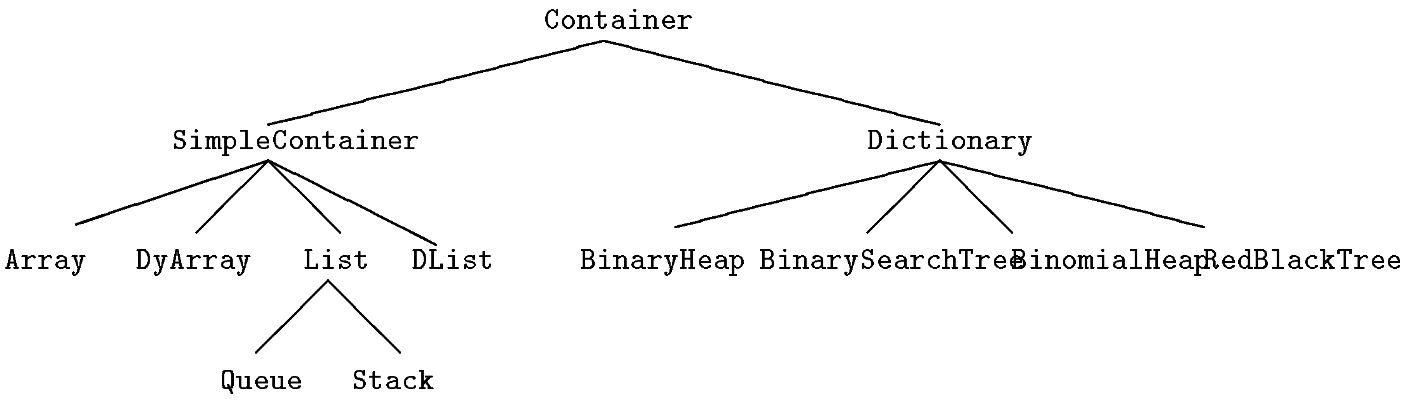
Like the *Container* hierarchy, the *Collection* hierarchy defines a standard interface of data and methods necessary to realize a *Set* or *Sequence* object. *Collection* objects such as *Sets* and *Sequences* are templated both by element type and implementation type, where the implementation is a member of the *Container* hierarchy.

There is a small set of methods which are available for all *Containers* and *Collections*. These include:

```
size()              returns the number of items stored
emptyQ()            returns TRUE if the Container is empty
fullQ()             returns TRUE if the Container is not empty
memberQ(item)       returns TRUE if the item belongs to the Container
first()             returns the first item in the Container
last()              returns the last item in the Container
insert(item)        inserts the given item into the Container
remove(item)        removes the given item from the Container
clear()             removes all items from the Container
iterate()           iterates through all items in the Container
display()           display to standard out the contents of the Container
```

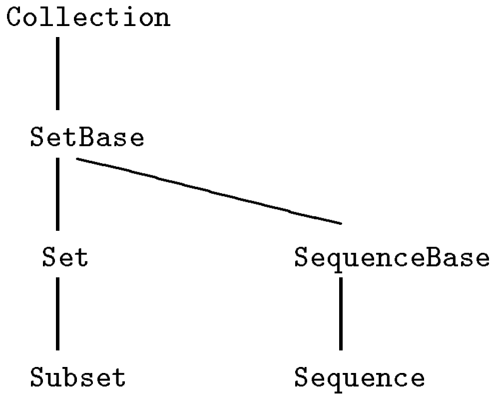
Containers and *Collections* may have other methods which apply to their means of storage, but any basic structure must minimally define the above methods.

3.1.1 ***Container*** **Hierarchy**



The *Container* class is itself an abstract class, so no objects of type *Container* can be instantiated. It includes as subclasses all data structures which can actually be instantiated. These data structures have a prescribed storage and access method. For instance, a *List* is composed of objects of type *ListNode* which contain the item being stored and point to the next *ListNode* . An *Array* is composed of a block of contiguous storage and is accessed using an index number. A *SimpleContainer* is templated by element type. For example, *List*<int> and *Array*< *Vertex**> are legal *Container* types. Dictionaries are templated both by element type and *key* type, where the latter is the data type of the key field stored with every entry. An object declared as BinaryHeap<int, *Vertex**>, for example, contains pointers to *Vertex* objects, but **int** keys are used for retrieval and determining position in the heap.

3.1.2 *Collection* Hierarchy



The *Collection* class is also an abstract class, and will appear only as a pointer to an object from one of its subclasses. The principal classes in the *Collection* hierarchy are *Set* and *Sequence*. A LINK *Set* is currently a Multiset (it may contain duplicate elements), but a separate Multiset class has been implemented and soon will be incorporated. The *Set* class will then be modified to conform to the traditional definition of set. Both the *Set* and *Sequence* classes rely on a standard set of functions which are implemented by any of the data structures belonging to the Container Hierarcy. Both *Set* and *Sequence* inherit from a class called the *SetBase*, which is templated both by element type and type of implementation. Namely, it is possible to have a *Set* of *Items* implemented as an *Array*<*Item*>, a *List*<*Item*>, or as any *Container* of *Items*. LINK provides a default *List* implementation for both *Set* and *Sequence*, but it is easy to use a variety of implementations should the user want to experiment. For example, *SetBase*<int,*Array*<int> > is a set of integers and is implemented as an *Array*, while *SetBase*<int,*List*<int> > is a set of integers stored in a *List*.

Collection objects such as *Set* and *Sequence* maintain a reference count to enable efficient copying of shared objects.

3.1.3 Symbolic Set Manipulation

Implicit manipulation of power sets is possible through a class *SymSet*, which for the moment is detached from the *Collection* Hierarchy. This class brings the functionality of *SetPlayer* to LINK. For a set A comprised of non-negative integers, the expression $pow(A)$ (resp. $pow_k(A)$) denotes the set of all subsets (resp. all subsets of cardinality k) of A. The data structures employed to represents and manipulate such expressions are called *difference trees*.

All standard set operations, such as the union, difference, intersection, membership predicate, and computing cardinality are performed[1] implicitly. Having implicit implementation

[1] We used the implementation of difference tree machinery in the system *SetPlayer*; the conversion to LINK was done by Darren Lim of Rensselaer Polytechnic Institute.

of power sets allows the use of sets with cardinality up to 2^{400} or larger, depending on the power of the multiprecision arithmetic package used.

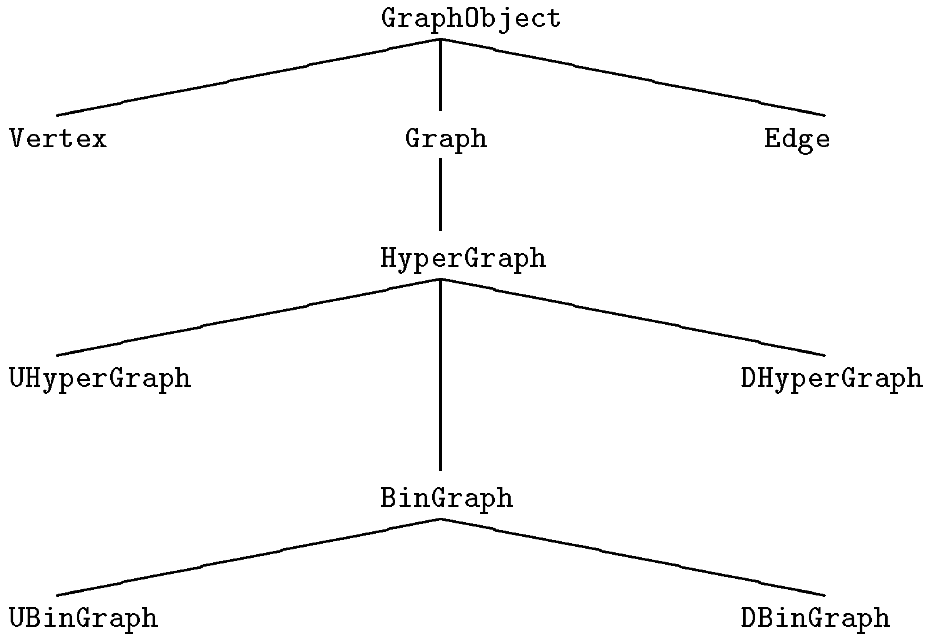
3.1.4 *Iterators*

All *Collection* and *Container* subclasses must provide a function called `iterate()` which takes as arguments an *Iterator* object and an *Item* object (where *Item* is the element type of the *Collection* or *Container*). The *Item* object will hold the next value in the iteration *Sequence*. One or more *Iterator* class objects may operate simultaneously on any given *Container*. The iterator class keeps a pointer to the data structure being iterated and a pointer to the next *Item* in the iteration *Sequence*. However, in the case of contiguous data structures such as arrays, the iteration is performed with respect to index, relieving the need for extra storage in the data elements. From programmer's point of view, iteration with *Iterator* objects is syntactically consistent regardless of data structure. The following is a typical instance of *Iterator* usage:

```
Set<Edge*> edges = vertex->incidentEdges();
Edge* edge;
Iterator<Edge*> get_edge(&edges);
while (get_edge(edge)) {
        operate(edge);
}
```

This code will retrieve all incident edges of the vertex in whatever order the implementation of *Set* defines. Since it is a *Set* and not a *Sequence*, no order is enforced, although many *Containers* may store elements in a default lexicographic order.

3.2 *Graph* Library



The *Graph* library includes a class hierarchy with the *Graph* class defined as an abstract base class for six types of graphs. *Graph* primitive operations such as *neighbors()* are implemented as virtual functions to return appropriate results depending on graph type. Many standard algorithms can operate on a *Graph* pointer (e.g. many of the search and path algorithms), although they can be written specifically for one or another of the six graph types. Since most algorithms use only a small set of graph primitive operations to manipulate graphs, the same code can often work correctly for any type of graph.

The actual graphs that can be instantiated are:

Hypergraph	edges of 0 or more vertices, ordered or unordered
Undirected Hypergraph	edges or 0 or more vertices, unordered
Directed Hypergraph	edges of 0 or more vertices, ordered
Binary Graph	edges of 2 vertices, ordered or unordered
Undirected Binary Graph	edges of 2 vertices, unordered
Directed Binary Graph	edges of 2 vertices, ordered

Although there are six distinct classes of graphs in the *Graph* hierarchy, most of the code resides in the *HyperGraph* class. The other graph classes are straightforward modifications that benefit heavily from inheritance and make only slight changes. (See [4, 3, 7, 15] for a discussion of graphs and hypergraphs.)

The methods defined on a *Graph* include:

```
addVertex(name)               create a Vertex and add to the graph
addVertex(vertex)             add the given vertex to the graph
addUndirectedEdge(v1, v2)     add edge of two vertices to the graph
addUndirectedEdge(Set<v>)     add edge containing the set of vertices
addDirectedEdge(v1, v2)       add edge of two ordered vertices
addDirectedEdge(Sequence<v>)add edge containing the sequence of vertices
addEdge(edge)                 add edge to the graph
deleteVertex(vertex)          remove and delete the vertex from the graph
deleteEdge(edge)              remove and delete the edge from the graph
removeVertex(vertex)          remove but don't delete the vertex
removeEdge(edge)              remove but don't delete the edge
degree(vertex)                how many edges are incident to this vertex
inDegree(vertex)              how many edges come into this vertex
outDegree(vertex)             how many edges come out of this vertex
findVertexByName(name)        return the named vertex
findEdgeByName(name)          return the named edge
```

The incident edge list maintenance is automatically handled within the methods used to add or remove vertices and edges. For instance, if a vertex with a given name is added to the graph, the *Vertex* object is created and a pointer to it is stored in the *Array*< *Vertex**> that the *Graph* maintains. If an undirected edge of two vertices is added to a *Graph*, the edge is created as a *Set*< *Vertex**> and each of the vertices has this newly created edge added to its respective incident edge lists. Similarly, when removing edges the incident edge sets of individual vertices will be modified.

The actual storage for any graph consists of an array of vertex pointers and an array of edge pointers. *Array* was chosen as the data structure rather than *List* or *Set* because iteration is more straightforward on an *Array* object. Since processing all vertices or edges is a very common operation in graph algorithms, methods are provided for iterating over them using a `for` loop rather than an *Iterator*. These methods and an example of their use are:

```
vertexStart()   returns a Vertex** to the beginning of storage of the array
edgeStart()     returns a Edge** to the beginning of storage of the array
vertexCount()   returns the number of vertices in the array
edgeCount()     returns the number of edges in the array

Vertex** vertices = graph->vertexStart();
for (int i = 0; i < graph->vertexCount(); i++)
    process(vertices[i]);
```

Because *Graph* is an abstract base class, graphs must be instantiated as an object of one of the subclasses. For example:

```
UBinGraph* graph = new UBinGraph();
```

In subsequent statements, the undirected binary graph may be referred to as a Graph* object. This allows it to be passed as an argument to the algorithm functions.

3.2.1 Hypergraphs

Hypergraphs have edges containing any number of vertices, including zero. A hypergraph is actually a mixed hypergraph where an edge is either a *Set* or *Sequence* of vertex pointers. Hypergraph has subclasses for directed hypergraph and undirected hypergraph. These further restrict the edge to being a *Sequence* or a *Set*, respectively.

3.2.2 Binary Graphs

Binary graphs are hypergraphs with an edge defined as a *Set* or *Sequence* of exactly two vertex pointers. A BinaryGraph object may have edges that are directed or undirected (mixed binary graph), meaning the edge may contain a *Sequence* or a *Set*. BinaryGraph has subclasses for directed binary graph and undirected binary graph which further restricts the edge to being a *Sequence* or a *Set*, respectively.

3.2.3 *Vertices* and *Edges*

The primary reason for the development of the *Collection* hierarchy was to support in a simple and consistent way the definition of edges under the various types of LINK graphs. A LINK *Edge* is simply a *Collection* of *Vertex* pointers. Graph types are distinguished by edge type, which in turn is determined by the type of *Collection* object used to store the vertices of an edge. For an undirected graph, the *Collection* is a *Set*<Vertex*> and for the directed graph it is a *Sequence*<Vertex*>. For a binary graph, the *Collection* is a *Set* or *Sequence* of size 2, and for a hypergraph it is a *Set* or *Sequence* of any size. The complexity of the *Collection* hierarchy allows the *Edge* class to be defined very simply. Methods on the *Edge* class consist of:

```
vertices()          returns a \Collection\ pointer to vertex contained
sourceVertex()      returns the first vertex, useful in binary graphs
sinkVertex()        returns the last vertex, useful in binary graphs
```

A LINK *Vertex* object contains a *Set* of pointers to the edges incident to that vertex. The user methods of the class *Vertex* are:

```
incidentEdges()
inNeighbors()
outNeighbors()
neighbors()
```

3.2.4 *Attributes* and *GraphObjects*

Since one goal of LINK is to provide a general purpose graph library, it must be possible to create graphs which model many different types of problems. In order to accomplish this, *Graph*, *Vertex*, and *Edge* objects often need to contain one or more types of attributes. Examples of useful attributes of vertices and edges are color, location, and weight. Furthermore, graph properties are stored as attributes of a *GraphObject*. A templated *Attribute* class supports this need. The *GraphObject* base class (which has *Graph*, *Vertex* and *Edge* as subclasses) contains a list of *Attributes*, so every graph-related object in the system can have *Attributes* associated with it.

In the graph attribute system, attributes with default values are stored in the graph's *GraphObject* portion. If an attribute has a different value within a vertex or an edge, a copy of the original attribute is made and the value is changed. The copied attribute is stored in the attribute list belonging to the vertex or edge. When the value of an attribute is requested for a particular vertex, the method first looks in the attribute list belonging to that vertex. If the attribute is not there, a pointer is followed to the graph which owns the vertex and the default value is returned from there. The templated functions to deal with attributes are:

```
newAttribute(Graph*, String name, const Item&);
setAttribute(GraphObject*, String name, const Item&);
getAttribute(GraphObject*, String name, Item&);
```

Additionally, there are non-templated functions to allow the deletion and display of attributes, and the resetting of attribute values to the default value. They are:

```
resetAttribute(Graph*, String name);
deleteAttribute(Graph*, String name);
deleteAttribute(GraphObject*, String name);
printAttribute(Graph*, String name, AttrCategory);
displayAttribute(Graph*, String name, AttrCategory);
```

An *Attribute* object may be instantiated as a permanent attribute by the call:

```
Attribute<int> index_attr = new Attribute<int>(graph, "index", 0);
```

which defines `index_attr`, an attribute of type **int** with name *index* and default value 0. This attribute will exist for the graph until a `delete` is called on `index_attr`.

A temporary attribute may also be created in any scope, global or local, by the call:

```
Attribute<int> index_attr(graph, "index", 0);
```

When the constructor for *Attribute* is called, the object is created and adds itself to the list of attributes for the given graph. When the `index_attr` goes out of scope, the destructor not only deletes the attribute, but causes the default attribute to be removed from the graph's attribute list and causes all copies of the attribute to be removed from the attributes lists of all vertices and edges. This means that a temporary attribute can be created by an algorithm and used in running the algorithm, but as soon as the algorithm has completed, the temporary attribute disappears. This helps prevent waste of memory storage on objects that need not persist. If an algorithm should have to return values to the calling program using an attribute, the calling program code must declare the attribute before the algorithm call. Values can then be retrieved by the calling program before the attribute goes out of scope and is deleted.

Attributes may be templated with virtually any type. For instance, the algorithm "LexBFS" (a special form of Breadth-First Search useful to human genome researchers) makes use of the following complex *Attribute*:

```
Attribute< DList<Vertex*>* > list_attr(graph, "list", (DList<Vertex*>*) 0);
```

Here, `list_attr` is an attribute defined as a doubly-linked list of vertex pointers. Because attributes can easily be assigned to vertices, edges, and graphs, a LINK application can be tailored to have the graph model many problems. If the application were a computer security network program, for example, useful vertex attributes could be *workstation_type*, *security_level*, *classification_level* and *building_location*.

3.2.5 Subgraphs

In large applications, it is sometimes necessary to consider contractions of large graphs, in which induced subgraphs are identified and replaced with single vertices. Problems thus can be presented as coarse graphs whose vertices can be expanded to provide more detail in areas interesting to the user.

In LINK, this hierarchical idea is implemented by *subgraphs*. A subgraph is a *GraphObject* of the same type as its enclosing graph. This feature is implemented through the use of several attributes on graphs. The basic idea is that the vertices making up the subgraph are replaced within the graph by a *supervertex* (a *contraction* operation) that has a pointer to the subgraph as an attribute. Edges move down to the subgraph level only if they belong to the induced subgraph.

When an algorithm such as `AllPairsShortestPath()` is run on a graph containing *closed* subgraphs, i.e. supervertices, it treats each supervertex as a normal vertex. For example, if the original graph contained 100 vertices and a subgraph of 25 vertices is created and closed, the algorithm finds the pairwise shortest paths between 76 vertices (75 original vertices and one supervertex). If the subgraph is subsequently opened, the algorithm again operates on 100 vertices.

Subgraphs can be nested many levels. The only requirement is that every vertex which is put into a subgraph must have the same parent. While the owner graph is always the original graph, the parent graph of a vertex is the immediate graph or subgraph containing the vertex. Deletion of vertices in a subgraph is supported efficiently.

The large applications discussed in 4.2 will make heavy use of the subgraph feature. In particular, the navigation and network projects use several levels of subgraph nesting.

The bulk of the code to manage subgraphs can be found in the **HyperGraph** class. Methods available to the application programmer for managing subgraphs include:

```
addSubgraph(Graph* owner, List<Vertex*>* vertices);
dissolveSubgraph(Vertex* supervertex);
openSubgraph(Vertex* supervertex);
closeSubgraph(Vertex* supervertex);

subgraphAddVertex(Vertex* vertex);
subgraphRemoveVertex(Vertex* vertex);
subgraphAddEdge(Edge* edge);
subgraphRemoveEdge(Edge* edge);
```

3.3 Interface Library

One of the key ideas in the design of the LINK graph hierarchy was that one should be able to create a graph without creating the machinery necessary to view it. In this way, applications involving graphs that do not need to be rendered can be created without having the additional storage and inefficiency of the graphical information. In order to facilitate this, the *GraphView* and *Graphics* classes exist separately from the **Graph** class itself. When a graph is displayed, the geometric information stored in these classes is used to display the graph. When a vertex is rendered, it is actually a *VertexGraphic* which contains the display information. Methods are also included to access the *Graph* object given its *Graphic* and vice-versa. These methods are necessary to allow a GUI user to select vertices and edges through point-and-click operations and then perform graph functions such as deletion on the selected objects without corrupting the underlying graph.

3.3.1 *GraphWindow*

The *GraphWindow* class contains the following:

- a pointer to the graph that it is currently rendering,
- a pointer to the *GraphView* object that contains the *Graphics* of that graph,
- a pointer to the *GraphCanvas* object into which the rendering takes place,
- a *Selection* that is the set of *Graphics* currently highlighted or selected in that window.

When manipulating a graph in the user interface, *GraphWindow* methods are called. The methods of class *GraphWindow* support the creation of vertices and edges. The methods will not only cause the vertex or edge to be created, but will add it to the graph, create the *Graphics*, and add them to the view. The *GraphWindow* thus operates very closely with the underlying graph methods.

Graphics are not created automatically for every graph generated. If an application creates a graph by making new vertices and edges and adding them to the graph, the *Graphics* will not exist, even if the graph is later associated with a window. There are *GraphView* methods that will create either individual *Graphics* or all *Graphics* for a graph.

```
createVertexGraphic(v)    creates a single vertex graphic for a vertex
createEdgeGraphic(e)      creates a single edge graphic for an edge
createGraphicsForGraph()  creates all graphics needed for a graph
```

When these *Graphics* are created, the initial location is defined to be (0,0). After creating the *Graphics*, it is necessary to call a layout algorithm to assign locations before drawing in a *GraphWindow*. A typical scenario might be:

```
GraphWindow* window;
GraphView* view = window->view();
UBinGraph* g = GenUBinGraph(number_of_vertices, edge_probability);

window->graph(g);        // assigns the new graph to this window
view->createGraphicsForGraph();
CircleLayout(view, g);
window->drawGraph();
```

One of the more complicated things the *GraphWindow* must do is manage subgraphs. The user can select a group of vertices and cause them to be coalesced into a subgraph. The user may open up a subgraph, close a subgraph (create a supervertex) or dissolve a subgraph through the GUI as well as through graph methods. Some applications might allow the user access to these operations through the *GraphWindow*, while other applications will control all graph manipulation explicitly through code.

The *GraphWindow* methods for dealing with subgraphs include:

```
createSubgraph();
dissolveSubgraph();
openSubgraph();
closeSubgraph();
```

These take no arguments because they operate on the current *Selection* in the *GraphWindow*. If an application is using its own graphical user interface with subgraphs, the above code probably will have to be used. It is in this code that the *GraphView* is altered for visible and hidden *Graphics* (see below).

3.3.2 *GraphView*

A *GraphView* object contains a pointer to the *GraphWindow* object associated with it and a pointer to the *Graph* object which the view renders. The *GraphView* object maintains a list of visible *VertexGraphics* and a list of visible *EdgeGraphics*. Each time the *GraphWindow* object requests that a *GraphView* object be drawn, the view iterates through its lists of visible *Graphics* and has each one draw itself.

A complication with the *GraphView* object comes into play with the creation of subgraphs. When a subgraph is created and closed, the vertices within that subgraph are hidden from the window. If the subgraph is subsequently opened, then all the previously hidden *Graphics* in that subgraph must be displayed. This problem is solved by maintaining a list of hidden *VertexGraphics* and hidden *EdgeGraphics*. When a subgraph is closed, the *Graphics* for its vertices and edges are moved to the hidden *Graphics* lists. When the subgraph is opened, the opposite occurs.

3.3.3 *GraphCanvas* and *Canvas*

The *GraphCanvas* class has methods to draw vertices and edges at specific points. The *GraphCanvas* inherits from a base class called *Canvas* that contains the Xlib commands for drawing rectangles, lines and strings. When a *GraphCanvas* object is requested by a *VertexGraphic* object to draw a vertex at a particular point, the *GraphCanvas* object looks up the vertex attributes to determine the vertex color and location, then calls the low level routine to render the vertex. Currently, a vertex in a LINK application is rendered as a rectangle but the implementation of icon drawing is planned for the near future.

3.3.4 *Graphics*

For each vertex and edge in a graph, a *Graphic* object exists which is capable of drawing itself in a specified *GraphView*. There are separate graphics methods for binary edges and hyperedges. A hyperedge is represented by a small square that is connected to its *Set* (undirected) or *Sequence* (directed) of graph vertices. We call this small square a hyperedge *locus*.

Currently, hyperedges can be rendered in two ways. One layout, more appropriate for directed hypergraphs, looks like a polygon on which the vertices are laid out in a circular embedding. The sequence of vertices is connected to the hyperedge locus in a cycle, starting and ending in sequence order from the hyperedge locus. The other layout, again with vertices in a circular embedding, looks like a star with a single hyperedge locus with connections to each contained vertex. The lines that connect the vertices to the hyperedge locus are not edges in the graph, but are sometimes called *tentacles*. In a directed hyperedge, these edges are numbered to indicate order.

Locations, colors and directions are attributes contained in the attribute lists of the individual vertices and edges. Locations are stored as floating point numbers between 0.0

and 1.0. Methods are provided on the *Canvas* for converting between these world coordinates (class Location) and screen coordinates (class Point).

3.3.5 *Selection*

Each *GraphWindow* has an instantiation of the *Selection* class in order to determine which objects are currently selected. This can be used by an application's GUI to allow users to select subsets of vertices and edges. Many GUI functions operate on a *Selection*, including the deletion of objects, moving and subgraph functions.

The *Selection* class stores the *Graphics* that are selected, but there are also methods for retrieving the actual *GraphObjects*. For instance, when a *Selection* is deleted through the GUI operations, the actual *GraphObject* must be removed as well as the *Graphic* from the *GraphView*.

3.4 Generator Library

The generator library contains both random and deterministic graph generators [6]. Arguments to the random binary graph generators include the graph type, the number of vertices, and an edge probability p which represents the probability that an edge exists between a pair of vertices. There are also several simple random hypergraph generators for both directed and undirected hypergraphs. The random generator creates a graph with a given number of vertices and hyperedges, using a probability that a particular vertex is within each hyperedge *Sequence* or *Set*. There are also *k-regular* and *k-uniform* hypergraph (defined below) generators. These also require the number of vertices and edges as parameters. However, instead of a probability, the third parameter is a *degree*. In k-regular graphs, the degree is the number of edges any particular vertex occurs in. Therefore, the degree can be no greater than the total number of edges. In k-uniform graphs, the degree is the number of vertices in each hyperedge's *Sequence* or *Set*.

The generators are useful in the testing of algorithms and layouts. Presently the set of generators is very small. However, more sophisticated random and deterministic generators can be implemented very easily in LINK.

3.5 Layout Library

Various binary and hypergraph layouts are provided in the layout library [2]. For binary graphs, the circle layout is probably the most useful for viewing a small graph, but there are several additional binary random graph layouts, including `Random` and `Poisson`. The Poisson layout embeds the vertices on the unit square using the poisson disk algorithm commonly used in sampling. The grid graph has its own layout and naturally embeds the graph on its own grid. For bipartite binary graphs there is a bipartite layout. There is also a layered embedding that is useful for showing the breadth-first-search tree structure of a binary graph. The Network Navigation application uses this layout extensively.

Effectively embedding hypergraphs in a meaningful way can be very difficult even with small graphs. LINK has several options that provide the user considerable flexibility. The “polygon” and “star” options are described above in Section 3.3. One of the most effective hypergraph layouts is the bipartite layout. It places the vertices in a uniform horizontal band at the top of the canvas and the hyperedge vertices uniformly in a horizontal band at the bottom of the canvas. The circular layout embeds the vertices in a circle and finds moments for the hyperedge loci from their respective vertex sets. In addition, functions exist to lay out only the hypergraph vertices or hyperedge loci. For example, the loci could be laid out in a horizontal band, while the vertices are laid out in a circle.

3.6 Algorithm Library

The algorithm library can be divided into several convenient groups:

- Predicates
- Graph Properties
- Connectivity
- Graph Traversal
- Path and Tree Computation
- Unary and Binary Graph Operations
- Other

Predicates test for certain graph properties such as connectivity and biconnectivity and return the boolean values TRUE or FALSE.

The Graph Properties algorithms currently implemented are

- Center
- Diameter
- Eccentricity
- Girth
- Radius
- Maximum Degree
- Minimum Degree
- Degree Sequence

Center returns a *Set* containing pointers to the vertices with eccentricity equal to the graph radius. Diameter returns the diameter of the graph as an integer. Eccentricity returns either a sequence of integers containing the eccentricity for each vertex in the graph or the eccentricity for a single vertex (if one vertex is specified). Girth, Radius, Maximum and Minimum Degree return the appropriate value as an integer. Degree Sequence returns a list of the degrees of each vertex, sorted by degree.

Within the Connectivity section, the following algorithms are included:

- Connected Components
- Biconnected Components
- Bridges
- Articulation Points
- Strongly Connected Components

Connected Components, Biconnected Components and Strongly Connected Components return a list of lists containing pointers to the vertices in each component. Bridges returns a list of lists containing the pointers to the vertices representing each bridge. Articulation Points returns a list of pointers to the vertices which are articulation points.

There are three types of search algorithms in the Graph Traversal section:

- Depth First Search
- Breadth First Search
- Lexigraphical Breadth First Search

They each return a sequence of pointers to vertices, in the appropriate order. The default root of each traversal is the first vertex in the graph's vertex list. The "Lexicographical Breadth First Search" is a specialized algorithm implemented for the Human Genome Project at LANL.

Within the Path and Tree Computation section are:

- Shortest Path
- All Pairs Shortest Paths
- Minimum Spanning Tree (An implementation of Kruskal's algorithm)
- Minimum Spanning Tree (An implementation of Prim's algorithm)

Shortest Path sets an attribute named `dist` on each vertex. This attribute represents the distance of that vertex from the source vertex. All Pairs Shortest Paths returns a matrix containing the distances between each pair of vertices. The two implementations of Minimum Spanning Tree set an attribute named `MSTedge` on each edge in the minimum spanning tree.

The unary operations included so far are the Copy, Divide, and LineGraph operations. Join, Union, Intersection and Product are the binary operations included in the Graph Manager. The unary operators return the new graph within the *GraphWindow* of the old graph. The binary operators on the other hand return a new graph in a new Graph Window, leaving the old graphs in their respective *GraphWindows*.

In the Other section are the push-relabel implementation of the maximum flow algorithm and Topological Sort. The maximum flow algorithm sets an attribute named `flowvalue` on each edge. This attribute represents the final flow value on that edge. Topological Sort returns a sequence of pointers to vertices, sorted in topological order.

A number of algorithms (for instance, the graph traversal algorithms) work for both undirected and directed binary graphs and hypergraphs. However, some algorithms currently work for only one particular type of graph. Maxflow, for example, only works on directed binary graphs presently. The algorithms contained in the Graph Properties section, however, run on both directed and undirected binary graphs.

Relatively little time was spent implementing these algorithms, most of which can be found in standard books on algorithm [9, 21, 22, 1]. The LINK libraries are built to facilitate rapid algorithm implementation. Even complex and sophisticated algorithms can be written without too much trouble. The Maxflow algorithm [14] was written with the idea of exercising many of the LINK library features, including the graph data structures as well as the *Container* class and its ability to vary implementation types. In a modest amount of code using combinations of heuristics and methods, 21 variations of the push-relabel algorithm are included. These variations are parameterized so that the user can experiment effectively. (For a review of experimental analysis see [18, 17].)

4 LINK Applications

A LINK application can be built by accessing the necessary LINK libraries and then adding application-specific user interface code and libraries. A prototype application, LINKGUI, was developed in tandem with LINK. Currently, several other applications are under development in the areas of fraud detection, network navigation, and cluster analysis.

4.1 LINKGUI

The LINKGUI application is LINK's general graph research tool which includes a GUI as well as a command-line interface.

4.1.1 Motif Graphical User Interface

The LINKGUI GUI consists of one ManagerShell and one GraphManager and allows many *GraphShell* and *GraphWindow* combinations. This allows the creation of several different graphs in different windows or several views of the same graph in different windows. The GraphManager contains a list of all active *GraphWindows* and all views. This allows the user to perform unary operations, such as complement, on a particular graph or to combine graphs via binary operations such as graph union or intersection

The Motif interface code is generated automatically using XDesigner. XDesigner has a feature that allows top level shells to be classes in their own right. Using this feature we can create multiple LINK *GraphWindows* so that many graphs can be viewed at one time. The *GraphShell* has a virtual window space controlled by scroll bars and also a font option where the size of vertices and labels can be changed. These are somewhat helpful when viewing small to moderate sized graphs, but there are no effective methods for viewing very large graphs in LINK at this time.

The LINK interface library provides general X library based classes and methods for rendering graphs which can be used in conjunction with any of the X interface toolkits. However, the actual LINK graph libraries are independent of any representation requirements and can be used with any graphical user interface system. The Network Navigation application described below will use Galaxy as a GUI, for example.

4.1.2 Tcl Command-Line Interface

There may be times when a LINK application user does not require or want a graphical display of the graphs under study. Examples of this type of use are large-scale experimental studies and testing library functions such as algorithms. For this reason, a prototype command-line interface for LINKGUI was designed using Tcl, an interpreted command language developed by John Ousterhout. The handle table facilities of Extended Tcl, an extension to the Tcl language developed by Karl Lehenbauer and Mark Diekhans, were also incorporated. With this interface, graphs and other LINK objects may be created and used, but no *Graphics* are created for the objects.

LINK objects are represented in the Tcl interface using handle tables. These tables contain a handle name by which to access each user-created LINK object and a pointer to that object in memory. There is a separate table for each major class of objects. Object member functions are accessible via Tcl commands. The C++ code to create these commands was originally hand-coded, but students at SUNY-Stony Brook have recently developed a system, `meta`, which automatically generates this code given a LINK object description and its corresponding header file. Basically, the C++ code to create a Tcl command for a member function consists of code to access a LINK object via the appropriate handle table, call the member function, and then return the result. In some cases, a new object is created so code is included to create a Tcl handle for the object and enter the information into the handle table. In the current system, the following LINK objects are accessible via the Tcl interface:

- *Graphs*

- *Containers*
- *Collections*
- *Sequences*
- *Sets*
- *SubSets*
- *Lists*
- *Queues*
- *Arrays*
- *SortedArrays*
- *Permutations*
- *Matrices*
- *BinarySearchTrees*
- *RedBlackTrees*

Generator and algorithm library functions are also available via Tcl commands. Code for algorithm function calls is very similar to the member function command code described above. Code to create generator function Tcl commands must initially call the generator to create the LINK object, and then create a Tcl handle and store the handle and object pointer in the table. This approach of creating Tcl commands by wrapping object-access code around calls to the LINK libraries means that the GUI and Tcl interface are both calling the same functions. This facilitates use of the Tcl interface for testing of new and modified library functions without having to use the LINKGUI GUI.

4.2 Other Applications

Work is now beginning on the third non-prototype LINK application. The best way to to develop a LINK application is to take one of the existing applications and use it as a model for building the new application. Using this approach and a good interface builder allows an application to be started in under a week.

4.2.1 Organized Fraud Detection

The first major LINK application is an organized fraud detection project. The idea behind this project is to find connected components in a graph, where components might indicate some form of organized fraud. The main processing involves calculating similarity measures between items of the data set based on a combination of measures. An example of a similarity measure is the edit distance between corresponding textual fields within two records of information. If the measure between two items is above the requested threshold, then both items are added automatically to a graph as vertices and an edge is added between them with the calculated measure as the weight on the edge. After each pair is examined, the final graph is run through the connected components algorithm that reports lists of vertices that are connected. Each of the distinct components, which indicate a high similarity between its vertices, are then made into a subgraph. The main graph now consists of these contracted subgraph components.

When the graph is initially created, only vertices and edges are put into the new undirected binary graph. When this process is complete, the view is asked to create *Graphics* for the graph and to draw it in the window. Other operations can be executed on this graph. For instance, it is possible to raise the threshold on the combination of the measures for the edges. This involves dissolving each subgraph back into the main graph, looking at each edge and deleting it if the threshold is no longer met, looking at each vertex and deleting it if it no longer has any incident edges, and then recalculating connected components and redoing the subgraphs. Because *GraphObjects* exist separately from the *Graphics* that can be found in the *GraphView*, every time an edge or vertex is deleted, the corresponding *Graphic* must be found in the *GraphView* and deleted as well.

One interesting twist to this application is that the user interface requires a selection mechanism for building a measure descriptor from various fields of information, which is used to decide how to calculate a combination measure and obtain a single value. This is handled by creating a graph with no edges that has as vertices each of the fields which can be selected. These vertices can be selected through the GUI and dragged to a particular point, causing them to be added to the measure descriptor. So the fraud detection application makes use of two kinds of graphs, one of them simply in the role of an enhanced user interface widget.

In little over a month from the start of this project, we were able to demonstrate it, complete with user interface. Part of this project involved extensive use of the Sybase database system, which in turn made extensive use of templated arrays for holding information.

4.2.2 Network and Database Navigation

Another prototype project that makes use of LINK is an interface to Mosaic. The idea is to have the navigation of a network or database reflected in a graph so that the users can see where they had been and can get back to that place easily. It is not always clear when navigating information systems whether or not a loop has been traversed. The LINK interface helps the user to maintain the big picture and avoid such pitfalls. Implementing the

LINK interface to Mosaic involved modifications to Mosaic to cause it to write HTML names along a socket to the Navigator application so that a graph could be built. The Navigator also had to be able to talk along the socket to Mosaic so that when a user clicked on a vertex in the Navigator window s/he could see that window of information under Mosaic.

Work on this prototype took under a week with the bulk of the effort involved in getting the sockets to work. The actual creation of the graph was very simple, and the standard LINKGUI application interface for drawing was used as is. A simple layered embedding algorithm lays out the graph as it is created. Each time a new set of vertices and edges is added to the graph, the layout algorithm is called and re-draws the graph. The HTML names are attached to each vertex as an attribute and are displayed as the label on the vertex.

This project is formally under way now, and will be using an graphical user interface built with Galaxy, an authoring software tool. Discussions are taking place as to whether to make use of the LINK graph library only, or to also use the general LINK interface library, supplying new hooks for drawing in a Galaxy environment rather than in Motif.

4.2.3 Cluster Analysis

Several exploratory projects involving the clustering of data [16] are currently being developed using LINK. One uses the layout features of LINK to visualize multi-dimensional data. Points in a multi-dimensional space are related to one another by a distance metric over that space. Typically, the number of dimensions far exceeds 2 or 3. In order to visualize the relationship among these points, a graph is created with points represented by vertices and distances between points represented by edges weighted by a distance metric value. A force-directed placement embedding of a graph is then used to lay out these points on a 2-dimensional surface [12, 13, 8]. The graph need not be complete. For instance, distances above or below a threshold may be the only relationships between points that are important. In this case, edges with metric values below or above the threshold are not included. This approach has been used with very promising results on a distance matrix of image signatures. Examples of this include LandSat image signatures and medical image signatures. The signatures showed very nice grouping, consistent with the types of images. Other similarity or dissimilarity measures could easily be used with other data. We hope to experiment soon with document signatures [10].

Another project involves two groups of data intermingled in the same dimensional space. The goal is to determine if there are clusters of one group separable from the other or at least large differences in density of one group compared to another in a given cluster. We have implemented a hierarchical clustering algorithm [26] which uses k-connected components to cluster the groups. The percentages of each group within a particular component can easily be computed from the decomposition tree created by the hierarchical clustering algorithm. This method has a number of applications including organized fraud detection.

5 Future Work

Though we are currently working on the beta version of LINK there is still extensive work to be done. This work can be classified into three different levels: the system level, the algorithms level, and the applications level. The systems level holds the most immediate challenges, as LINK is currently neither robust nor easily portable. There will undoubtedly be an inexhaustible demand for new algorithms and applications once a robust and easily extensible system is available.

To enumerate some systems level issues, we must consider the ease of extensibility, robustness, and portability of the system. The user's view of the textual and graphical interfaces needs to be better specified, as well as the system's error handling features. For example, some graph algorithms will only run on binary graphs, but no checking is done within the algorithm to determine whether the correct type of graph is being passed as an argument. Although the library functions have been exercised in creating LINKGUI, extensive and systematic testing of all library contents has not yet been done. The algorithm, layout, and generator libraries also need to be extended to include more functionality.

Another area of future work is in loading and saving graphs. All graph types can now be written to and read from an ASCII file. However, if the current GNU `g++` compiler is used, a user-created *Attribute* object cannot be saved when the graph is saved to an ASCII file. The problem is that the attribute must specify its type within the file so that when the graph is reloaded, the attribute can be created correctly. The straightforward way to implement this would be to overload the attribute save method. There should be a general `Attribute<Item>::saveToFile()` method which would work for most things, but we would like to overload this for `Attribute<int>`, `Attribute<float>` and `Attribute<String>`. The AT&T compiler can handle this, but the current GNU compiler cannot. An additional problem with saving attributes is that the value will not always even be meaningful, independent of the compiler used. A saved pointer to an object that existed during one run of a LINK application will make no sense when the graph is reloaded. Design decisions must be made to handle this type of issue in a consistant manner.

Except for the work on the automatic code generator, the Tcl interface has essentially remained unchanged since the summer of 1993. Consequently, there are objects and algorithms in the LINK libraries that have no corresponding Tcl object or command. These need to be added and the existing code needs to be examined in light of the evolution of LINK. In particular, a design philosophy for creating a Tcl interface for various LINK applications needs to be developed (similar to the current GUI application building mechanisms). A mechanism to use both a Tcl interface and GUI in tandem would also be useful. For example, the user may choose to put repetitive operations into a Tcl script and automatically display the results using the GUI. The user interface for the original LINKGUI application needs to be redone with an emphasis on providing more user-friendly research and application environments for LINKGUI.

We are currently exploring the use of C++ communication libraries (developed at LANL) in conjunction with the LINK libraries to enable the user to run LINK in an SPMP data parallel model. This would allow us to develop algorithms in LINK to be run on massively parallel machines and distributed heterogeneous architectures.

One of the main goals of the LINK-project is to stimulate research in experimental computer science. To achieve it, LINK provides both the expressiveness of the language and the efficiency of the underlying data structures and algorithms. An important part of the future development of LINK is the integration of explicit and implicit (symbolic) data representation. At a more advanced stage, the system will be able to select the most efficient data representation by default and transform the data into implicit from explicit, or reverse, form. This ability is crucial when the problem at hand is of the type that frequently occurs in the combinatorial design theory, such as the packing design or the constant weight code problem.

One enhancement to LINK will be direct support for experimental research. A set of coordinated tools which support such functions as the design and control of experiments, the collection and storing of data, and support for external applications (such as algorithm animation and statistical analysis) will make LINK an unparalleled environment for experimental research in combinatorics and graph theory. An important area which requires this type of support is that of algorithmic learning. The development and testing of algorithms that attempt to solve combinatorial problems by learning from experience requires a flexible environment which facilitates the design and execution of experiments, a well-organized database of results, and an interface to statistical analysis software. Learning algorithms will become a substantial part of the future LINK's library.

Work must proceed in making LINK more organized, robust, and efficient. We also need to provide documentation so that other developers and users can more easily use LINK. Jonathan Berry, originally from RPI and who worked on the LINK project during its first summer at LANL, is scheduled to take over a coordination role at DIMACS in June 1995. This central coordination will enable significant progress to be made in readying LINK for general release.

6 Conclusion

While much work is needed in refining LINK for general release, it has already proven to be useful in development of applications for diverse projects. Its generality, obtained through careful design of class hierarchies, provides the researcher with a rich and natural set of tools for exploring and using graph theory and combinatorics.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: theory, algorithms, and applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] G. Di Battista, P. Eades, and R. Tamassia. Algorithms for drawing graphs: an annotated bibliography. Technical report, Department of Computer Science, Brown University, Providence, RI, 1993.

[3] C. Berge. *Hypergraphs.* North-Holland, 1989.

[4] C. Berge. *Graphs.* North-Holland, 1991.

[5] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The setplayer system for symbolic computation on power sets. *Journal of Symbolic Computation*, 14:645–662, 1992.

[6] B. Bollobas. *Random Graphs.* Academic Press, 1985.

[7] J. Bondy and U. Murty. *Graph Theory with Applications.* North-Holland, 1976.

[8] J. Cohen. Drawing graphs to convey proximity: An incremental arrangement method. Technical Report R53-07-94 S-241,681, Department of Defense, 1994.

[9] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[10] M. Damanshek. Gauging similarity via n-grams: Language-independent categorization of text. *Science*, 267, 1995.

[11] N. Dean and G. E. Shannon, editors. *Computational Support for Discrete Mathematics.* American Mathematical Society, 1993. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 15.

[12] P. Eades and K. Sugiyama. How to draw a graph. *J. of Information Processing*, 13(4):424–437, 1990.

[13] T. Fruchterman and E. Reingold. Graph drawing by force-directed placement. Technical Report UIUCDCS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1990.

[14] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[15] F. Harary. *Graph Theory.* Addison-Wesley, 1969.

[16] A. Jain and R. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1988.

[17] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge.* AMS, 1993.

[18] C. McGeoch. *Experimental Analysis of Algorithms.* PhD thesis, Carnegie Mellon University, August 1986.

[19] K. Mehlhorn and S. Nähger. Leda: A platform for combinatorial and geometric computing. *CACM*, 38(1):96–102, Jan 1995.

[20] M. Mevenkamp, N. Dean, and C. Monma. NETPAD user's guide and reference guide, 1990.

[21] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP: Volume 1, Design and Efficiency.* The Benjamin/Cummings Publishing Company, 1991.

[22] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[23] G. Shannon, L. Meeden, and D. Friedman. SchemeGraphs: An object-oriented environment for manipulating graphs, 1990. Software and documentation.

[24] S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica.* Addison-Wesley, 1990.

[25] R. Tamassia and I. G. Tollis, editors. *Proceedings of the DIMACS International Workshop on Graph Drawing*, 1994. Lecture Notes in Computer Science, 894.

[26] R. Tarjan. An improved algorithm for hierarchical clustering using strong components. *Information Processing Letters*, 17:37–41, July 1983.