LINK: A system for graph computation *

Jonathan W. Berry
Elon College
Department of Computing Sciences

Nathaniel Dean
Rice University
Department of Computing and Applied Mathemics

Mark K. Goldberg
Rensselaer Polytechnic Institute
Department of Computer Science

Gregory E. Shannon
Ascend Communications
Enterprise Networking Division

Steven Skiena
State University of New York, Stony Brook
Department of Computer Science †

April 14, 1999

1 Introduction

LINK is a freely-available, flexible environment for the creation, manipulation, and drawing of graphs and hypergraphs. This paper will describe the basic architecture of the system and illustrate its flexibility with several examples. These descriptions will be accompanied by commentary on the associated design decisions, but will certainly not be exhaustive. The *LINK* manual fills in

*We would like to acknowledge the support of DIMACS and NSF grant CCR-9214487. DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Laboratories, Lucent Technologies/Bell Laboratories Innovations, 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.

†This work was partially supported by NSF Grant CCR-9625669 and ONR award 431-0857A

the details. *LINK* is distinguished from existing software for discrete mathematics by its layered interface, including a graphical user interface tied into an object-oriented Scheme language interface which incorporates graphics, and an extensible underlying set of C++ libraries. This selection of interface first and foremost ensures that the interface is customizable, but also provides a large degree of platform independence. *LINK* currently runs on Unix and Microsoft Windows operating systems. The command language interface is much friendlier and more flexible than the "scripting" or "prototyping" facilities typical of other graph visualization software; indeed, it is an integral part of any *LINK* session. Unlike any existing system for discrete mathematics, *LINK* combines a computing environment comparable to those of the familiar symbolic mathematics packages with graph editing facilities.

2 Background

Over the past ten years, there have been many efforts to construct software systems for discrete mathematics, and in particular, for the manipulation of graphs. None, however, has resulted in a product with influence comparable to the familiar symbolic mathematics packages.

LINK grew out of the experiences of the authors of *Combinatorica* [15], an extension package for *Mathematica* due to Steven Skiena, *NETPAD* [12] due to Nathaniel Dean and others at Bellcore, *SetPlayer* [1], due to Mark Goldberg and his students at Rensselaer Polytechnic Institute, and Gregory Shannon et. al.'s *GraphLab* [14]. None of these systems has the potential to be a widely-useful environment for both graph manipulation and computation, though *Combinatorica* is widely successful as a tool for computing and displaying static pictures of discrete structures. The authors of these systems recognized that a more general tool would be useful and proposed the development of *LINK*. This was to be a freely available and portable software system for discrete mathematics which would combine the computational flexibility of *Combinatorica* with a customizable graphical user interface (GUI).

After several years of development, the system, which includes a 200 page on-line manual and an on-line tutorial, is now freely available from the *LINK* web site:

`http://dimacs.rutgers.edu/Projects/LINK.html`.

During the following discussion, please refer to the architecture diagram in Figure 1 for context.

LINK 's design philosophy placed flexibility as the highest priority, and this led to the selection of STk, Erick Gallesio's object-oriented Scheme language interface to John Ousterhout's portable, interpretive Tk graphics system. [13][6]. Tk enables involved graphics programming without any knowledge of underlying

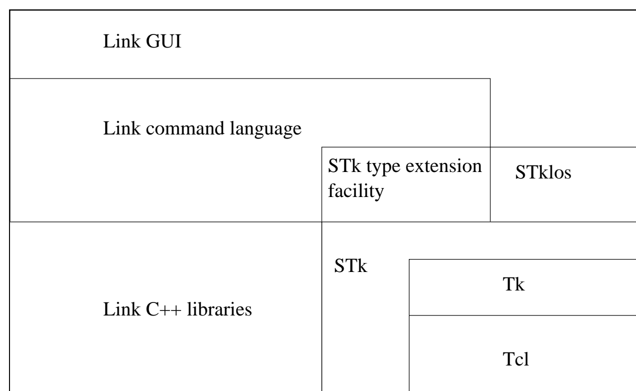

Figure 1: *LINK* architecture diagram

graphical systems such as X-windows, and offers the advantages of interpretation and portability at the cost of speed. This means that the system is not appropriate for viewing massive data sets. Graph views with a few thousand objects have been used, but these took several minutes to load on a Sparcstation 5.

STk provides its users with Tcl/Tk functionality under a very high level interface. Scheme is a well-known, small functional language, and is much friendlier than Tcl for the intended users of *LINK*: students and researchers who are mathematically, but not technically inclined. Furthermore, STk provides STklos, an elegant facility for building object-oriented extensions which effectively shields the user from the details of Tk. *LINK*'s command language is built into Scheme using a type extension mechanism provided by STk, and the *LINK* GUI is object-oriented, written using STklos. This gives the user great flexibility when manipulating objects such as graphs, vertices, and edges.

Several other graph manipulation systems have been designed using Tk, including Graphlet [1], which is built on top of the LEDA C++ library [11]. However, these systems rely on Tcl, a language which is too technical for students and faculty exploring basic discrete mathematical concepts.

The remainder of the paper is broken into sections describing *LINK*'s layered interface, its flexibility, and its roles in research and education.

3 *LINK*'s Templated C++ Libraries

Underlying the *LINK* system is a set of object-oriented C++ libraries designed to offer a rich and coherent set of graph and collection objects to support programming in the pursuit of discrete mathematics research.

[1] See http://fmi.uni-passau.de/himsolt/Graphlet.

3.1 *Collections* and *Containers*

The basis of the *LINK* system is a set of classes grouped into two hierarchies: *Collection* and *Container*. The *Collection* hierarchy consists of multisets (bags), sets, and sequences, while the *Container* hierarchy is subdivided into data structures such as lists, arrays, binary heaps, red-black trees, etc. As in the Standard Template Library (STL), the glue that binds all container and collection objects is the *Iterator*. STL was not mature enough at system design time to be used, and even had it been, we might not have employed it since our *Collection* hierarchy offers some flexibility that STL does not. For example, implementations of *Collection* objects can be interchanged, and lazy copying is supported. *LINK*'s *Iterator* object can be used to retrieve the elements of any *Collection* or *Container*, and has been used to create a set of constructors and assignment operators which allow the easy transfer of data between any two collection or container objects.

The hierarchies of *Collection* and *Container* classes give library programmers a common interface for performing simple operations such as copying, assignment, insertion, deletion, extraction, comparison, display of values, and the set primitive operations. Also available are standard queries such as membership and whether or not the structure is empty, sorted, a permutation, or a subset of another.

Collections are notable because they are arranged into a templated hierarchy, with the set and sequence classes inheriting from a multiset class. This organization provides the advantages of type safety, reference counting, and polymorphism, but also has required the construction of an automated explicit template instantiation tool to keep track of the hundreds of instantiations and explicit specializations required to compile the libraries.

The most fundamental design decision in the construction of the *Collection* hierarchy was to state that a set *is a* multiset (bag) in which redundant insertions are prohibited. This inheritance decision, as opposed to that declaring a multiset to be a set with frequency counts, contributes to the flexibility of the system at the cost of efficiency in certain queries.

A complete coding example showing *Collection* objects in use is shown in Figure 2

3.2 The Graph Hierarchy

The *Collection* hierarchy makes it possible to define a variety of graph objects. These are arranged into the hierarchy shown in Figure 3 to take advantage of polymorphism. As in the *Collection* hierarchy, objects of the *Graph* hierarchy can be copied and assigned gracefully. In Figure 3, the naming convention for graph classes is as follows: If the first letter is an *M*, then the graph allows multiple edges. If the next letter is *U*, then the graph is undirected, if it is *D*, then the graph is directed, and if it is anything else, the graph may contain both

```
#include<iostream.h>
#include<LINK/basic/MSet.h>
#include<LINK/basic/Set.h>

int main()
{
        MSet<int> mset1, mset2;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                mset1.insert(i);
                mset2.insert(2*i);
                mset2.insert(2*i);
        }

        cout << mset1 << " + " << mset2 <<" = " << (mset1 + mset2) << endl;
        cout << mset1 << " ^ " << mset2 <<" = " << (mset1 ^ mset2) << endl;
        cout << mset1 << " - " << mset2 <<" = " << mset1-mset2 << endl;
        Set<int> s;
        s  = mset1+mset2;
        cout << "returned into a Set: " << s << endl;
}
```

```
{0 0 1 1 2 2} + {0 0 2 2 4 4} = {0 0 0 0 1 1 2 2 2 2 4 4}
{0 0 1 1 2 2} ^ {0 0 2 2 4 4} = {0 0 2 2}
{0 0 1 1 2 2} - {0 0 2 2 4 4} = {1 1}
returned into a Set: {0 1 2 4}
```

Figure 2: A complete program using *Collection* objects

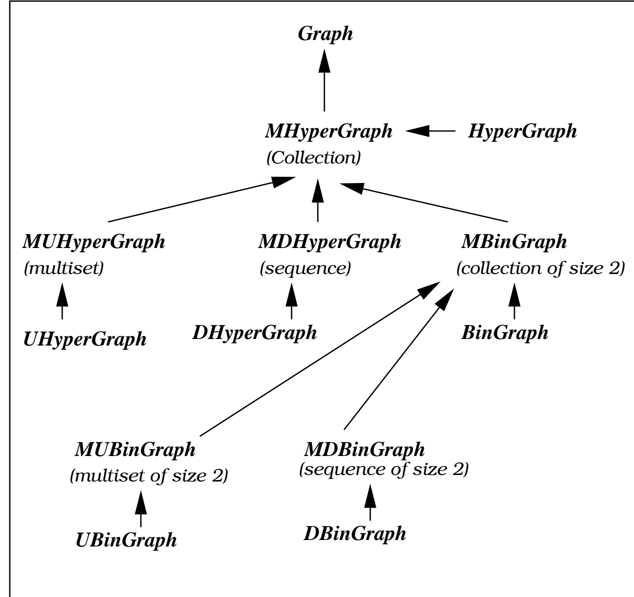

Figure 3: This figure shows *LINK*'s graph hierarcy. The parenthesized expressions below certain classes show the structure in which an edge stores its vertices in graphs of that type.

directed and undirected edges. The graph classes that begin with *M* are simple extensions of the corresponding multigraph classes. The only difference is that no multiple copies of edges can be inserted.

Each graph type is accompanied by the type of object used to store the vertices within an edge. For example, each edge in a *HyperGraph* or *MHyperGraph* may be either a multiset or a sequence, while each edge in an undirected binary graph (*UBinGraph*) must be a multiset of size 2.

3.2.1 Graph Types

The central goals of *LINK*'s design philosophy are that many different graph types should be available to the user and that algorithms need only be written once to work on many types of graph. The *Collection* hierarchy has been used extensively to meet both goals, with the result that *LINK* users may now select between 12 types of graph by specifying the edge type. An edge is a collection of vertices, and the *Collection* hierarchy enables us to offer graph types classified by the following pertinent questions:

- **Multigraph or not?** Multiple edges between the same vertices can be allowed or not. The edge set of a multigraph can contain duplicate elements,

while that of a simple graph cannot.

- **Binary Graph or Hypergraph?** Edges in a binary graph are defined as groups of exactly two vertices, while edges in a hypergraph are defined as groups of arbitrary numbers of vertices.
- **Directed, Undirected, or Mixed Graph?** Each edge is either a multiset or a sequence of vertices. All edges of a directed graph must be sequences, while all edges of an undirected graph must be multisets. A single mixed graph, however, can contain *both* kinds of edge simultaneously.

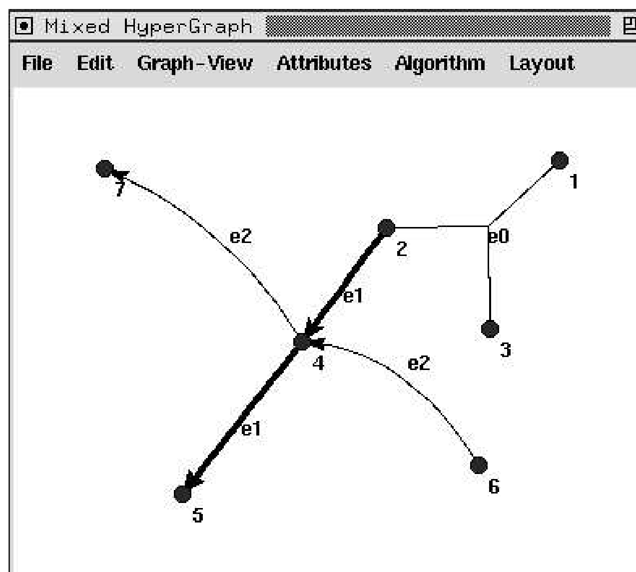

Figure 4: A "mixed" hypergraph

A *directed* hyperedge has been defined to be a set of vertices in which one vertex is specified to be a "sink," and the other vertices are assumed to precede that vertex [9]. We give a more general definition: a directed hyperedge is simply a sequence of vertices. A "mixed" hypergraph is shown in Figure 4, and a "mixed" binary graph with multiple edges is shown in Figure 5. Figure 4 is particularly interesting since it illustrates the two different modes for displaying hyperedges. Edge *e1* has been thickened for clarity, and edge *e0* is drawn using *star draw* mode, in which edge segments radiate from a central, draggable edge label. The other two edges are displayed using *path draw* mode, in which identically-labeled edge segments connect the vertices of the hyperedge. The complete order of vertices within a *directed* hyperedge, however, will only be

visible if the path draw display mode is used. The graphical user interface described below in Section 4.2 allows the user to change the edge display mode for the whole graph or some selected subset of the edges.

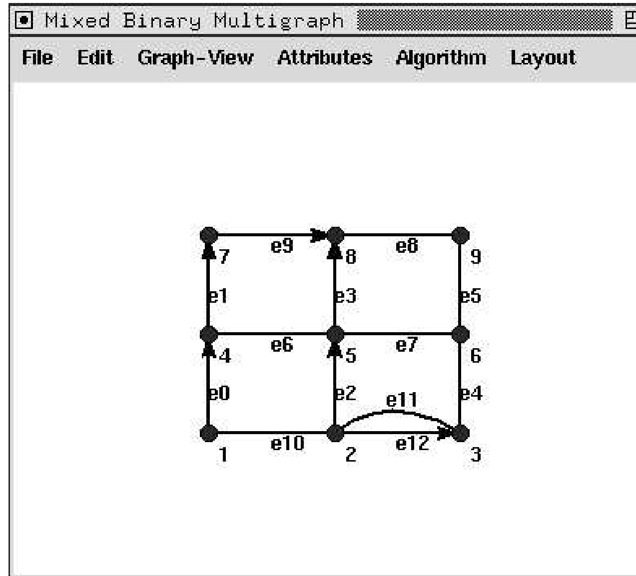

Figure 5: A “mixed” binary multigraph

Figure 6 shows a complete *LINK* program which illustrates interactions between graph types. An multigraph with directed edges (MDBinGraph) is built, then several other graphs of different types receive copies of the original graph subject to their own rules. Note that the simple graphs automatically remove multiple occurrences of edges, and that undirected edges correspond to directed edges during the copy. The call to `DepthFirstSearch` works for all of the graph types since the *Graph* hierarchy's primitive methods, to be described below, are all polymorphic.

It is fair to ask why algorithms in *LINK* are defined as standalone functions rather than *Graph* methods. Defining the algorithms as member functions might might seem reasonable at first glance, but the system would be less flexible were it organized this way. As it is, algorithms can be added to the library without modifying the interface of the *Graph* hierarchy. Furthermore, the latter is flexible enough to provide for algorithms to be written once, yet run on many graph types. This strategy is consistent with that of the C++ Standard Template Library.

```
#include<iostream.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/graph/UBinGraph.h>
#include<LINK/algorithm/Algorithms.h>

void main()
{
        MDBinGraph g1;

        g1.addVertices(3);
        Vertex** vertices = g1.vertexStart();
        int num_vertices = g1.order();

        g1.addEdge(vertices[0], vertices[2]);
        g1.addEdge(vertices[0], vertices[2]);
        g1.addEdge(vertices[2], vertices[1]);
        g1.addEdge(vertices[1], vertices[0]);

        DBinGraph g2(g1);
        MUBinGraph g3(g1);
        UBinGraph g4(g1);
        cout << g1 << "\tDFS:" << DepthFirstSearch(&g1) << endl;
        cout << g2 << "\t\tDFS:" << DepthFirstSearch(&g2) << endl;
        cout << g3 << "\tDFS:" << DepthFirstSearch(&g3) << endl;
        cout << g4 << "\t\tDFS:" << DepthFirstSearch(&g4) << endl;
}
```

```
{[0 1 2]  {<0 2> <0 2> <1 0> <2 1>}}    DFS:<0 2 1>
{[0 1 2]  {<0 2> <1 0> <2 1>}}          DFS:<0 2 1>
{[0 1 2]  {{0 1} {0 2} {0 2} {1 2}}}    DFS:<0 1 2>
{[0 1 2]  {{0 1} {0 2} {1 2}}}          DFS:<0 1 2>
```

Figure 6: Building graphs, copying graphs, and running algorithms

3.2.2 Graph Methods

All graph objects have the same core functionality, and the member functions implementing this functionality can be broken down roughly as follows:

- Predicates indicating whether the graph is mixed, directed, or undirected, whether multiple edges are allowed, and whether hyperedges are allowed.
- Vertex and edge manipulation routines such as the insertion and deletion of vertices and edges, either as individuals or in groups.

- Queries to determine properties of the graph such as order and size, whether or not two vertices are adjacent, and whether or not the graph is isomorphic to another. The latter test uses *nauty*, Brendan McKay's well-known and practical isomorphism testing tool. [10].

- Routines to find groups of vertices and edges such as the neighbors and incident edges of a given vertex. These routines use *Collection* functionality to return appropriate answers for undirected, directed, and mixed graphs. Also included are routines to return the sequence of edge objects associated with a path of vertices and vice versa.

- Input and output operations for graphs, vertices, and edges, both to and from files and the screen. Note: *LINK*'s graph language, fully specified in the manual, is similar in flavor to, but less extensive than, Himsolt's GML [7]. *LINK* explicitly stores a graph's attributes along with the graph rather than separating them into maps to be applied later. GML will be supported in future versions.

- Conversion and construction operations which take sets of vertices and edges and produce graph objects. This set of routines also can convert into adjacency or incidence matrix representation.

- Edge comparison routines – two edges are equal if they contain the vertices of the same name in the same order. One edge is "less than" another if it comes first in a lexicographic ordering.

Multigraphs also feature methods to collapse and extract subgraphs. These are useful when studying properties which are defined in terms of graph contractions, such as the chromatic polynomial.

3.3 Attributes

LINK provides a mechanism for creating and manipulating attributes of graphs, vertices, and edges. The default attributes for all graph objects (including vertices and edges) are currently *name, direction, width, size, weight, x, y, color, label, mark, type, starttime, finishtime, back, low, distance, pred*, and *forefather*. To save space, the attribute mechanism stores a single copy of each attribute for the entire graph until individual attributes are changed (at which point an individual copy is made for the affected object). Some of the default attributes are used by the graphical user interface, and some are used by fundamental graph algorithms. If different attributes are desired, however, defining new attributes is simple, both for the library programmer and the interface user.

4 The STk Interface

Recall that STk is an extensible environment which combines a high level command interpreter with powerful object-oriented graphics authoring tools. *LINK*'s command line and graphical interfaces are extensions of STk. In addition to a standard R^4RS Scheme interpreter, STk provides an object-oriented extension based on the Common Lisp Object System called STklos, as well as operating system shell access, regular expression processing, HTML awareness, and Unix socket handling. The STklos extension enables the scheme programmer to define classes and generic functions, and *LINK*'s interface takes full advantage of this power. All of the basic *LINK* objects have been "wrapped" into the STk interpreter so that users can create, manipulate, and destroy them, and *LINK*'s graphical user interface consists exclusively of new STklos classes so that users may take advantage of high-level, object-oriented functionality to manipulate their data.

```
STk>   (describe (graph (current-graph-view)))
#[<dbingraph*> #p63f5cc] is an instance of class <dbingraph*>
Slots are:
     val = {[1 2 3 4 5 6]  {<1 2> <1 6> <2 5>  <3 4> <5 3> <5 6>}}
#f
STk>   (map color (vertices (current-graph-view)))
("black" "black" "black" "black" "black" "black")
STk>   (define vg(car(vertices(current-graph-view))))
#[undefined]
STk>   (set! (color vg) "green")
#[undefined]
STk>   (map color (vertices (current-graph-view)))
("green" "black" "black" "black" "black" "black")
```

Figure 7: This Scheme code segment retrieves and manipulates the attributes of the graph of the current *graph-view* (window) after the user has constructed a graph in it.

4.1 *LINK*'s STklos Objects

The *LINK* interface user has access to graph objects at both the graphical user interface level and the Scheme command language level. *LINK*'s manual contains dozens of Scheme programming examples, and we will include some herein. Figure 7 contains Scheme code to retrieve the colors of the vertices that a user has created using the graphical interface. The example subsequently changes the color of the first vertex, a change reflected graphically on the screen. Note that *vg* is an STklos object which contains both a graphics field (displayed

```
STk> (define gv (show-graph (graph '(1 2 3) '((1 2) (2 3) (1 3)))))
#[undefined]
STk> (define g (graph gv))
#[undefined]
STk> (define eg (car (edges gv)))
#[undefined]
STk> (define e (edge eg))
#[undefined]
STk> eg
#[<edge-item> #p64ba68]
STk> e
#[<edge*> #p64c7a0]
STk> (set! (weight eg) 3.21)
#[undefined]
STk> (find-double-attribute 'weight e)
3.21
STk> (set-double-attribute! 'weight 2.1 e)
#[undefined]
STk> (weight eg)
2.1
```

Figure 8: Fundamental attribute operations

on the screen) and a reference to an underlying C++ vertex object. STklos gives users the considerable convenience of setting fields (or "slots") using the *set!* primitive. The expression `(color vg)` is a shorthand way of extracting the "color" field from the vertex.

The fundamental STklos graph objects of the *LINK* system correspond to the graph types described in Section 3.2.1: *vertex, edge, graph, bingraph, dbingraph, ubingraph, hypergraph, uhypergraph, dhypergraph mbingraph, mdbingraph, mubingraph, mhypergraph, muhypergraph, and mdhypergraph.* These and their most important methods are available to the *LINK* interface user, and detailed in the manual.

The *LINK* objects mentioned above are mirrored by special STklos graphics objects. The most important correspondences are those between classes representing the fundamental graph objects. Therefore, the three most important STklos classes in *LINK* are:

- *graph-view*: a window which contains a *LINK graph* object.
- *vertex-item*: a class containing an STklos graphics object and a *LINK vertex* object.
- *edge-item*: a class containing (potentially many) STklos *line* graphics objects and a *LINK edge* object.

Each *graph-view* object contains a copy of a *graph* object along with various graphics information. Is is possible to look at multiple views of copies of a given

```
STk>   (define gv (show-graph (graph '(1 2 3)
                     '((1 2) (2 3) (1 3)))))
#[undefined]
STk>   (map weight (edges gv))
(1.0 1.0 1.0)
STk>   (random-edge-weights (graph gv))
#[<ubingraph*> #p63c0c0]
STk>   (map weight (edges gv))
(38.0 58.0 13.0)
STk>   (define mst (kruskal (graph gv)))
#[undefined]
STk>   (describe mst)
#[<set<edge*>> #p6f1c98] is an instance of class <set<edge*>>
Slots are:
     val = {{1 2} {2 3}}
#f
STk>   (map (lambda (x) (find-double-attribute 'weight x))
            (set-edge- >list mst))
(38.0 13.0)
```

Figure 9: Scheme code to call an algorithm and examine the results.

graph, and the command interface can be used to simulate the visual effects that would accompany a true multiple viewing of a single graph instance.

Figure 8 illustrates the difference between STklos graphics objects and *LINK* objects. In this example, a graph is defined and displayed in a *graph-view* window called *gv*. The method *(graph gv)* returns the *LINK* graph object associated with this graph-view. The example then extracts the first *edge-item* from the graph-view and extracts the *edge* (a reference to a C++ object) associated with that edge-item. STklos supports "virtual" data within a class, and the *LINK* interface takes advantage of this by inseparably linking the attributes of the edge to those of the edge-item. When the user evaluates or modifies an attribute of the edge-item, such as its weight, that request is translated into an evaluation or assignment to the corresponding attribute of the underlying edge. For example, in Figure 8, changes to the edge-item's *weight* attribute are reflected in the edge's *weight* attribute and vice versa.

Figure 9 shows a more detailed example in which a graph is created and displayed, its edges are assigned random weights, and a minimum spanning tree is computed, extracted, and manipulated. This example illustrates two different levels of attribute retrieval: directly from a graphics object, and via a *LINK* object. The former method is used in the command *(map weight (edges gv))*, which extracts the *weight* slot from each STklos edge object in the graph window called *gv*. Note that *mst*, the variable used to store the result of the spanning tree algorithm, is a *LINK set* object. This is converted into a Scheme list using the *set-edge→list* method (provided with all collection objects).

```
(define (forefather-binding graph-view)
 (strongly-connected-components (graph graph-view))
 (bind (slot-ref graph-view 'graph-toplevel) "<KeyPress-f>"
      (lambda (x y)
           (flash (vertex-item
                 (find-vertex-attribute 'forefather
                    (slot-ref (car *link:selected-vertex-items*) 'vertex)) graph-view)))))
```

Figure 10: This complete code segment binds the "f" key so that the forefather of a selected vertex is flashed. The strongly-connected-components algorithm sets an attribute called *forefather* that is subsequently retrieved to decide which vertex to flash.

It is legitimate to ask why *LINK*'s *Collection* objects are at all useful in a list-based Scheme interpreter (why not just have all algorithms return Scheme lists?). Figure 11 provides an answer. Many of *Collection*'s methods, including the copying, insertion, deletion, query, and set primitive operations are available from the STklos interface. The example in Figure 11 is a simple graph sum computation using the set primitive operations.

```
STk>   (define g (graph '(1 2 3) '((1 2) (2 3) (3 1))))
#[undefined]
STk>   (define h (graph '(2 3 4) '((2) (2 3 4) (3 4))))
#[undefined]
STk>   (define ng (graph (+ (vertices g) (vertices h))
                         (+ (edges g) (edges h))))
#[undefined]
STk>   (describe ng)
#[<uhypergraph*> #p63e848] is an instance of class <uhypergraph*>
Slots are:
    val = {[1 2 3 4]  {{1 2} {1 3} {2}  {2 3} {2 3 4} {3 4}}}
#f
```

Figure 11: An example which uses *Collection* methods

4.2 Graphical User Interface

STklos provides a core set of graphics classes (windows, labels, buttons, etc.) which make interface customization a high-level operation, and *LINK*'s graphical interface consists of a set of classes which inherit from these. The result is that standard graph operations such as graph creation, insertion and removal of vertices and edges, execution of algorithms, and animation viewing are both point and click features and high-level STklos operations. Multiple graph windows can be viewed at once, and multiple algorithm animations can be stepped

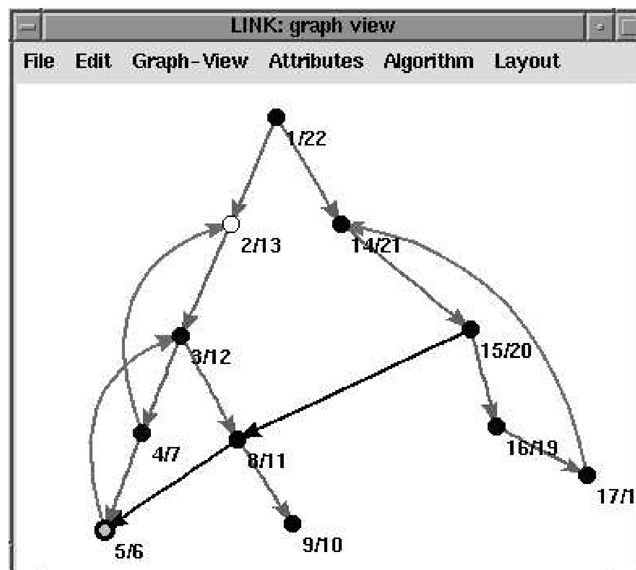

Figure 12: Forefather finding with finishing times depicted

through side-by-side for comparison. Graphics attributes such as world coordinates, size, arrow shape, label, color, text-color, stipple, outline width, outline color, and font can be evaluated and changed easily.

It is important to note that any command in the graphical user interface corresponds to a STklos command that could have been typed into the command line prompt. This makes *LINK* a powerful environment for systematically constructing, executing, viewing, modifying, and rerunning experiments. This interface has been used in several research projects, and two will be abstracted in Section 6.

4.3 Flexibility

Consider the following example of the interface's flexibility. When describing the strongly connected components of a directed graph, it is important to relate the concept of the *forefather* of a vertex. Simply stated, the forefather $\phi(v)$ of a vertex v with respect to a depth-first search is the vertex reachable from v which has the maximum finishing time in the depth-first search. *LINK*'s interface can easily be tailored to illustrate the concept intuitively. Figure 10 shows, in its entirety, the STKlos code necessary to "bind" the f key on the keyboard to a function which will flash the forefather of a vertex selected with the mouse. Figure 12 shows a graph-view in which a depth-first search has been run and the discovery and finishing times of the vertices are displayed. Selecting a vertex,

then pressing the *f* key highlights a vertex's forefather by flashing it several times.

4.4 Animations

When *LINK* algorithms are added to the C++ libraries, they can be augmented with special animation commands which modify the attributes of the graph's vertices and edges. These commands are executed if the algorithm is run from the interface (as opposed to being run from a standalone C++ program using *LINK*'s libraries). Algorithms selected from a graph-view bring up an animation controller/debugger which allows the user to step through the algorithm forwards and backwards, set breakpoints, continue, and restart. An example animation of a depth-first search is shown in Figure 13.

5 Algorithms, Generators, and Layouts

LINK's libraries include several fundamental algorithms for manipulating, generating, and drawing graphs, including depth and breadth-first search, Kruskal's and Prim's minimum spanning tree algorithms, Goldberg & Tarjan's maximum flow algorithm, strongly connected components, generators for random graphs, cycles, complete graphs and grid graphs, and circular, random, grid, spring, and component-wise layout algorithms. This algorithms library will certainly grow as the system develops and new versions are distributed.

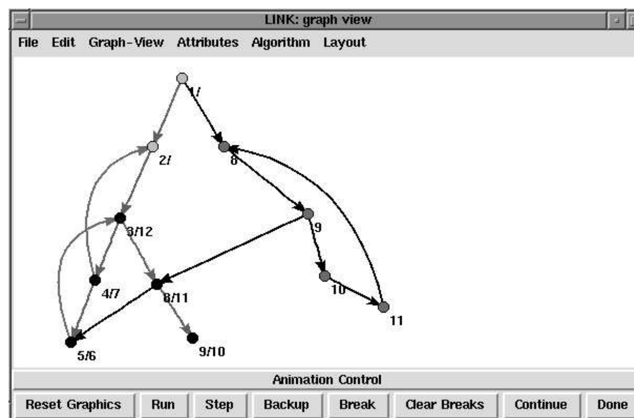

Figure 13: The animation controller

6 Research Examples

LINK has already demonstrated its usefulness in research, and its role in two recent projects will be summarized below.

6.1 Latka Tournaments

In the first project, Brenda Latka, while visiting DIMACS from Lafayette College, used *LINK* to assist in her study of infinite antichains of tournaments (complete directed graphs). An antichain of tournaments is a set of tournaments in which it is impossible to embed any tournament in the set within any other. Latka is interested in the construction of infinite antichains of tournaments. [5, 8]. Arguments to prove that a given set of tournaments is an antichain typically show that a specific sub-tournament of any given tournament cannot be mapped to any sub-tournament of any other tournament in that class. A crucial element of these arguments is the use of a non-trivial edge attribute: the number of directed 3-cycles in which an edge participates. Sub-tournaments and these edge attributes are easily visualized using *LINK*'s features: the for-

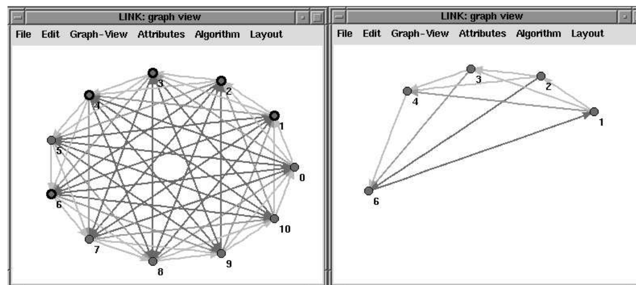

Figure 14: A Latka tournament and an induced subgraph extracted with *LINK*'s graphical user interface

mer can be extracted by clicking on vertices and selecting a menu option, while the latter can be computed (and the edges colored) by small programs written in *LINK*'s command language. Figure 14 shows an instance of a special class of tournaments defined by Latka (see [5, 8] for details), and an induced sub-tournament extracted by pointing and clicking (see the *LINK* web page for a color image).

Once *LINK* functionality had been used to generate Latka Tournaments, compute the edge attributes, and color their edges accordingly, Latka and Jonathan Berry used *LINK* to visualize dozens of these tournaments and variations upon them. Soon patterns began to emerge, leading Latka to conjecture that adding an extra parameter to her initial tournament construction would reveal the first known *infinite* set of infinite antichains of tournaments. The

conjecture, still unproven, will be detailed in a future paper.

During this time, Jonathan Berry was also serving as a mentor for Chris Burrows, a participant in the the *NSF Research Experience for Undergraduates (REU)* at DIMACS. He used *LINK* to study isomorphisms of certain subtournaments by implementing some special invariants described by Latka and using *LINK*'s point and click access to *nauty*. During this process, we observed that one of the Latka tournaments also happened to be a Paley tournament. [2] Burrows then used *LINK* to find two additional Latka-Paley tournaments, then develop the conjecture that no other Latka tournaments are Paley tournaments.

6.2 Market Basket Analysis

In another project, a set of supermarket data compiled and studied previously at Bell Laboratories was analyzed in a more meaningful way with *LINK*. Considering each type of item in a shopper's "basket" (e.g., bananas, 2% milk, skim milk) to be a vertex, "market-basket analysis" attempts to identify customer buying patterns by examining receipts. Correlations between purchases identified by the analysis can be used, for example, to arrange products more advantageously on the shelves or manipulate prices.

When a shopper purchases a set of items at once, we must represent this grouping somehow. An obvious approach to the problem is to place an edge between each pair of vertices in a basket, thus producing a graph where each purchase is represented by a clique. Given such a graph, however, the original "baskets" cannot by reconstructed. Nathaniel Dean and Jonathan Berry used *LINK* to re-model the problem using hypergraphs (graphs where each edge might contain more or fewer than two vertices) where each hyperedge represents a single shopper's basket. In addition to the considerable space savings inherent in this solution, more real-world information is preserved. Furthermore, the STk command language used by *LINK* makes it possible to pose interactive queries such as: "find all purchases in which both a snack food item and a beverage were purchased." A paper describing this work in more detail is available at the *LINK* web site [3].

7 *LINK* as an educational tool

LINK's flexible interface makes it an valuable educational tool, both in the classroom and as a vehicle for interesting assignments. The key binding example discussed above (see Figure 10) enables the instructor to present the forefather concept as a puzzle to engage students. This was done recently with encouraging success in an algorithms course at Elon College. The instructor also made extensive use of *LINK*'s graphical user interface and interactive algorithm

[2] The well-studied Paley tournaments consist of p vertices, where p is prime and congruent to 3 mod 4. Arc (i, j) exists iff $j - i$ is a quadratic residue mod p.

animations in class. A discussion of *LINK*'s role in computer science education is found in [2].

8 Lessons

The proposal for the *LINK* system enumerated a myriad of algorithms and structures to be implemented, but did not give an implementation-level architectural plan of the system itself. Designers and implementers devoted significant effort in this regard, but we had more expertise in algorithms than in the design of large C++ software systems. As a result, several early design decisions favored theoretical flexibility over design simplicity. These decisions later served to complicate the system significantly, and returned questionable benefits.

For example, it was considered very important by the project leader that we be able to swap implementations in our set and sequence classes so that we could experiment with algorithms. This led to an early decision not to wait for STL to mature, since its classes are not templated by implementation type. Like LEDA, we embarked on a path which led to a flexible, but non-standard library. In retrospect, and with our experience in designing large systems, the proper priorities would have been (and our future priorities will be) to favor simplicity and proximity to standard, validated libraries over the flexibility to facilitate enticing algorithms experiments.

A continuing problem for our design is the necessity of allowing our templated container classes to contain either pointer or object elements. Our elaborate hierarchy of templated classes had to be specialized systematically for pointer instantiations, as we didn't discover the method suggested by Stroustrup until later. Personal communication with one of the chief designers of STL has not revealed a "canonical" way to handle the problem of container classes that may contain pointers or objects.

We used template classes with static members and functions to wrap the *LINK* type and function extensions to the STk interpreter, and developed an automated system to instantiate these classes and provide initializations for the static data. The system also generates run-time commands to define interpreter-level classes and generic functions systematically. This facility works well, but it is useful only to developers. A front-end is needed to allow users to use it to associate new C++ types with STk.

9 Conclusion

The early development and primary designers and developers of *LINK* are detailed and acknowledged, respectively, in [4], while the current system is described in the manual available from the web site. Jonathan Berry took over

the direction of the project in June, 1995, and spent a year at DIMACS preparing the public release.

With further development, *LINK* can become a formidible tool for prototyping, teaching, and experimentation. Current directions include improving the documentation, incorporating STL into the basic library, extending the algorithms library, and improving the STklos interface.

10 Acknowledgments

We would like to acknowledge the support of DIMACS and the *LINK* grant: CCR-9214487. DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Laboratories, Lucent Technologies/Bell Laboratories Innovations, 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. The original primary investigator of the *LINK* project was Daniel Gorenstein, the founding director of DIMACS, and the four co-primary investigators were Nathaniel Dean, Mark Goldberg, Gregory Shannon, and Steven Skiena.

I would also like to acknowledge the contributions of Patricia K. Fasel of Los Alamos National Laboratory, who was the original project leader and who helped design the graph hierarchy and implemented many system fundamentals. Many students have helped with the *LINK* effort as well, and I acknowledge and appreciate their effort.

References

[1] 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.

[2] J. Berry. Improving discrete mathematics and algorithms curricula with LINK. In *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, pages 14–20, 1997.

[3] J. Berry and N. Dean. Market basket analysis with LINK. submitted to Congressus Numerantium, 1996.

[4] J. Berry, N. Dean, P. Fasel, M. Goldberg, E. Johnson, J. MacCuish, G. Shannon, and S. Skiena. LINK: A combinatorics and graph theory workbench for applications and research. Technical Report 95-15, Center for Discrete Mathematics and Theoretical Computer Science (see also: http://dimacs.rutgers.edu), Piscataway, NJ, 1995.

[5] G. Cherlin and B. Latka. A decision problem involving tournaments. Technical Report 96-11, Center for Discrete Mathematics and Theoretical Computer Science, 1996.

[6] E. Gallesio. The stk reference manual. Technical Report RT 95-31a, I3S CNRS, Université de Nice - Sophia Antipolis, France, 1995.

[7] M. Himsolt. The graphlet system. In Stephen North, editor, Graph Drawing*, Lecture Notes in Computer Science*, volume 1190, pages 233–240, 1996.

[8] B. Latka. Finitely constrained classes of homogeneous directed graphs. *The Journal of Symbolic Logic*, 59(1):124–139, March 1994.

[9] E. Mäkinen. How to draw a hypergraph. *International Journal of Computer Mathematics*, 34:177–185, 1990.

[10] B. McKay. Nauty user's guide. Technical Report TR-CS-90-02, Australian National University, 1990.

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

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

[13] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

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

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