Graph Drawing and Manipulation with *LINK*

Jonathan Berry [*] Nathaniel Dean [†] Mark Goldberg [‡] Gregory Shannon [§] Steven Skiena [¶]

Abstract

This paper introduces the *LINK* system as a flexible tool for the creation, manipulation, and drawing of graphs and hypergraphs. We describe the basic architecture of the system and illustrate its flexibility with several examples. *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 with access to Tk, and an extensible underlying set of C++ libraries. We conclude by briefly discussing roles *LINK* has played in research and education.

1 Background

Over the past several years, there have been several 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.

Some notable existing systems for discrete mathematics are *Combinatorica* [14], Steven Skiena's extension package for *Mathematica*, *NETPAD* [11] 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* [13]. For various reasons, none of these systems has the potential to be a widely-useful environment for both graph manipulation and computation. The authors of these systems recognized this and proposed the development of *LINK*, which was to be a freely-available and portable software system for discrete mathematics overcoming the various shortcomings of existing systems. After three years of development, the system is now freely available from the *LINK* web site:

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

LINK comes with a 170 page manual and an online tutorial.

[*] Elon College (berryj@numen.elon.edu)

[†] Bell Laboratories Innovations (nate@research.bell-labs.com)

[‡] Rensselaer Polytechnic Institute (goldberg@cs.rpi.edu)

[§] Milkyway Technologies (shannon@milkyway.com)

[¶] SUNY Stony Brook (skiena@cs.sunysb.edu)

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. [12][6]. Tk enables involved graphics programming without any knowledge of the X-window system, 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.

Several other graph manipulation systems have been designed using Tk, including Graphlet [1], which is built on top of the LEDA C++ library [10]. However, these systems rely on Tcl, a language more similar to operating system shell scripting languages than high-level programming languages. *LINK*'s command interface, on the other hand, offers the mathematician or computer scientist a high-level Scheme language interface with full object-oriented access to Tk. Scheme is a compact, standardized dialect of Lisp, a functional language useful for symbol manipulation.

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

2 *LINK*'s Templated C++ Libraries

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

2.1 *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 and arrays and keyed dictionary structures such as binary heaps, red-black trees, etc. As in the Standard Template Library, the glue that binds all container and collection objects is the *Iterator*. *LINK*'s *Iterator* object can be used to retrieve the elements of any *Collection*

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

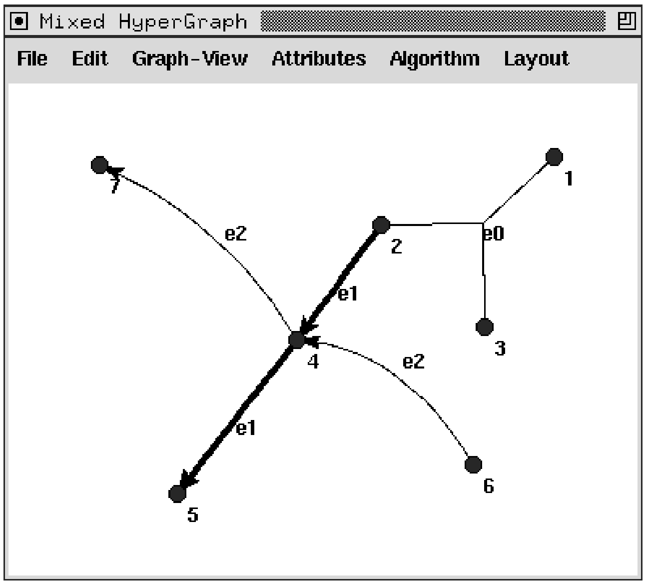

Figure 1: **A "mixed" hypergraph**

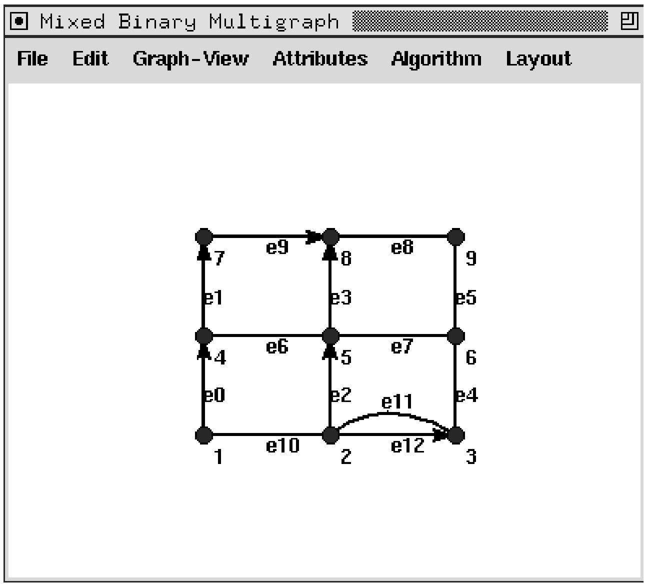

Figure 2: **A "mixed" binary multigraph**

or *Container*, and has been used to create a robust 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.

Collection objects are templated by both element type and implementation structure (a *Container* object) to allow programmers to experiment with different set and sequence implementations. This close relationship between *Collections* and *Containers* supports a reference counting scheme which allows *Collection* objects to be passed around efficiently without the unnecessary copying of elements.

Extension classes of the *Collection* hierarchy offer functionality including the manipulation of power sets, combinations, cartesian products, and permutations.

2.2 The *Graph* Hierarchy

The *Collection* hierarchy allows the definition of a rich variety of graph objects. These are also arranged into a hierarchy so that objects from different graph classes can interact. As in the *Collection* hierarchy, objects of the *Graph* hierarchy can be copied and assigned gracefully. The classes of the *Graph* hierarchy will be introduced below.

2.2.1 *Graph* Types

Central to *LINK*'s design philosophy are the goals 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 the 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.
- **Binary Graph or Hypergraph?** Edges can be defined as groups of two vertices or not. Relaxing this restriction results in hypergraph objects.
- **Directed, Undirected, or Mixed Graph?** Each edge is either a multiset or a sequence of vertices. Graphs can either limit their edge sets to undirected edges or directed edges, or they can make no such restriction. The result of the latter is a set of graph types in which a single graph object instance might contain both directed and undirected edges.

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 [8]. We give a more general definition: a directed hyperedge is simply a sequence of vertices. A "mixed" hypergraph is shown in Figure 1, and a "mixed" binary graph with multiple edges is shown in Figure 2. Figure 1 is particularly interesting since it illustrates the two different modes of 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 3.2 allows the user to change the edge display mode for the whole graph or some selected subset of the edges.

2.2.2 *Graph* Methods

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

- Vertex and edge manipulation routines such as the insertion and deletion of vertices and edges, either as individuals or in groups.
- Vertex and edge access routines which return specified individual or groups of vertices and edges.
- Queries to determine the order and size of the graph, whether or not it is directed, binary, simple, etc., 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. [9].
- Routines to find the neighbors and incident edges of vertices. 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.
- 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 comparable if they contain the vertices of the same name in the same order. Inequalities are resolved using lexicographic ordering.

Multigraphs also feature subgraph collapsing and extracting methods which are supported by the graphical user interface. These are useful when studying properties such as the chromatic polynomial, which are defined in terms of graph contractions.

2.3 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.

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

3 The STk Interface

STk is a complete programming environment in itself, and *LINK* inherits all of its functionality. 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, 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>   (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 4: **Fundamental attribute operations**

```
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 5: **Scheme code to call an algorithm and examine the results.**

3.1 *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 below. Figure 3 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 on the screen) and a reference to an underlying *LINK* 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 2.2.1: *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. 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 *oval* graphics object and a *LINK vertex* object.
- *edge-item*: a class containing (potentially many) STklos *line* graphics objects and a *LINK edge* object.

Figure 4 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* 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 4, changes to the edge-item's weight attribute are reflected in the edge's weight attribute and vice versa.

Figure 5 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

```
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 7: **An example which uses *Collection* methods**

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).

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 7 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 7 is a simple graph sum computation using the set primitive operations.

3.2 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 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 5.

3.3 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 6 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 8 shows a graph-view in which a depth-first search has been run and the discovery and

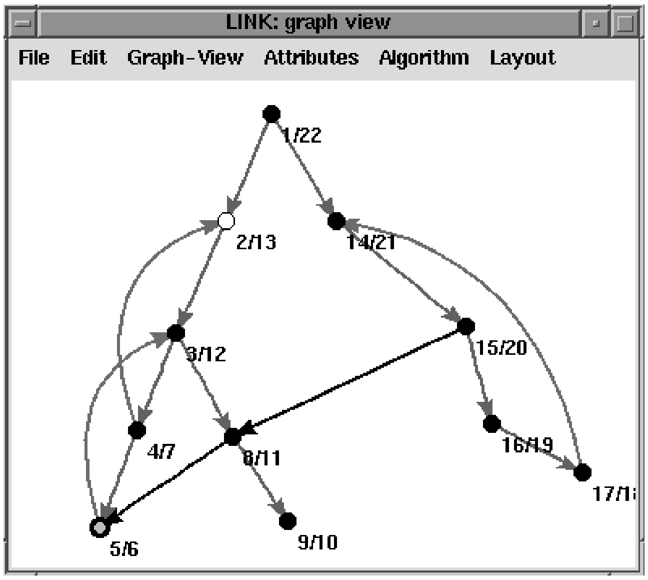

Figure 8: **Forefather finding with finishing times depicted**

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.

3.4 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). 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 9.

```
(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 6: **Binding the "f" key so that the forefather of a selected vertex is flashed.**

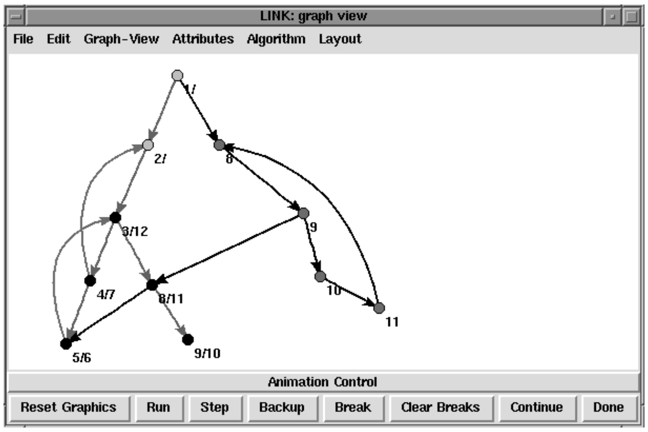

Figure 9: **The animation controller**

4 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. If the reader is interested in contributing to this effort, please contact Jonathan Berry at `berryj@numen.elon.edu`.

5 Research Examples

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

5.1 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 for 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, 7]. 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 former 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 10 shows an instance of a special class of tournaments defined by Latka (see [5, 7] 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 sub-tournaments 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.

5.2 Market Basket Analysis

In another project, a set of supermarket data compiled and studied previously at Bell Laboratories was analyzed in a more meaningful

[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.

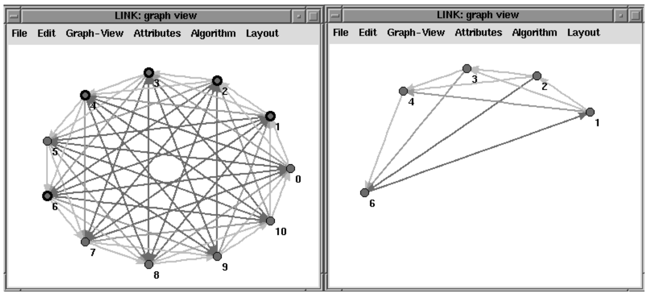

Figure 10: **A Latka tournament and an induced subgraph extracted with *LINK*'s graphical user interface**

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].

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

7 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.

Currently, *LINK* runs only on Unix systems (including Linux), but there is no major obstacle preventing a port to Windows, since the graphics system upon which *LINK* relies, STk [6], has already been ported. *LINK* for Windows is anticipated before the end of 1997.

With further development, *LINK* can become a formidible tool for prototyping, teaching, and experimentation. Development directions in the near future include improving the documentation, extending the algorithms library, and improving the STklos interface.

8 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.

We 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 we acknowledge 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*, 1997.

[3] J. Berry and N. Dean. Market basket analysis with LINK. http://dimacs.rutgers.edu/Projects/LINK.html, 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] B. Latka. Finitely constrained classes of homogeneous directed graphs. *The Journal of Symbolic Logic*, 59(1):124–139, March 1994.

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

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

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

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

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

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

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