Link: Exploring Combinatorial Objects

Jonathan Berry*
DIMACS Center†
Psicataway, NJ

Nathaniel Dean
AT&T Research
600 Mountain Avenue
Murray Hill, NJ 07974 U.S.A.

June 26, 1996

Abstract

This paper introduces the *Link* system for exploring combinatorial objects. It provides graphical and functional language interfaces which are user friendly and flexible, and includes a detachable set of C++ libraries. *Link* allows us to interactively explore the structure of combinatorial objects such as collections, graphs, and hypergraphs. Graphs and Hypergraphs can be defined in which multiple edges are allowed or not, and in which edge sets can contain undirected edges, directed edges, or both types. Full support for default and user-defined attributes is included.

We examine the use of *Link* in the context of a clustering problem which is a natural application for the use of hypergraphs. Real-world problems calling for the manipulation of hypergraphs or heavily attributed graphs are often simplified to avoid overcomplicated implementations. *Link* allows the user to define, manipulate, and visualize such combinatorial objects at a very high level.

1 Introduction

For many years many data analysts have developed a variety of data exploration techniques centered mostly around statistical analyses, database tools, technological developments, and data visualization. Still, there are some other very promising developments that seem to have been neglected and could yield some significant advances in knowledge exploitation and data exploration approaches. These developments have been motivated by the study of combinatorial objects such as sets, graphs, directed graphs, hypergraphs, partial orders, and sequences.

In discrete mathematics, we study relationships between collections of objects, real or abstract. In fact, many real-world problems can be modeled this way and then solved or approximated using discrete mathematical techniques. Computer technology supports the human decision-making process by helping data analysts and other data explorers discover non-trivial patterns in the data. Some examples include, but certainly are not limited to the routing of objects ranging from snow plows to airplanes, the scheduling of meetings, and the matching of jobs to appropriate candidates. Furthermore, the models generated during the solution of such problems are often easily visualized. Manipulation of these models and visualization of the results can lead to deeper understanding of the problems themselves.

Many computer tools have been developed for the manipulation and analysis of combinatorial objects. Most of them have been directed at very narrowly defined domains; e.g., graphs [1], sets [2], DAGs [6, 8], function libraries [7] and [11], and so on. Because of the combinatorial similarity between these objects from an OO point of view it makes sense to treat all of these discrete objects systematically under one system called *Link*.

To introduce the *Link* system we selected a dataset based on the purchase of items in a supermarket. The problems associated with this dataset are interesting because they highlight *Link*'s functionality.

*supported by DIMACS and the *Link* grant: CCR-9214487

†DIMACS is a cooperative project of Rutgers University, Princeton University, 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.

Most graph tools are designed to handle only simple, undirected or directed graphs which lack attributes. Nodes are placed in the plane so that their relative positions have meaning. and through interactive visualization we identify interesting features, clusters and patterns to ultimately derive some useful insights. A discussion of this problem, the limitations of modeling it with a simple graph, and a solution using *Link*'s functionality is presented below in Section 3.

2 Description of the *Link* System

Link is a cooperative project of DIMACS, Los Alamos National Laboratory, Indiana University, Rensselaer Polytechnic Institute, and The State University of New York at Stony Brook. The project was conceived by Nathaniel Dean, Mark Goldberg, Gregory Shannon, Steven Skiena, and Daniel Gorenstein, the founding director of DIMACS. Many people have contributed to the design and development of the system [3]. More information on *Link* is available at `http://dimacs.rutgers.edu`.

The most distinguishing features of *Link*, when compared to other graph visualization software, are that it combines flexible algorithmic and visualization features with a powerful computing environment. Users can manipulate collections, graphs and hypergraphs using an extension of the Scheme language in addition to menus and other graphical features. This combination of computing power and convenience with a flexible graphical user interface sets the system apart from past work.

Link's graphical user interface is written using Erick Gallesio's Scheme-Tk [5] (STk), which provides access to John Ousterhout's Tk [10] graphics language from the Scheme language rather than Tcl. Manipulating combinatorial objects in a functional language like Scheme is more natural than in a systems-oriented language like Tcl. STk also provides a flexible object-oriented extension which *Link* uses to provide a consistent syntax for graph manipulation. The STk system is portable to many workstation platforms, and is being ported to PC environments. Link inherits this portability as well as the flexibility inherent in programming with a high-level, interpretive graphics language. We will present several examples of this flexibility below.

The functionality of *Link* is provided by a set of C++ libraries. These are similar to those of LEDA, though they are more focused on combinatorial computing. There are three basic hierarchies of *Link* objects: containers, collections, and graphs. Comparison, display, and iteration over container objects such as lists, arrays, and binary search trees, and collection objects such as multisets, sets, and sequences have a consistent syntax. Furthermore, collections are templated by both element type and implementation type for convenient experimentation with data structures and algorithms.

Link's graph hierarchy consists of twelve different varieties of graph and hypergraph. Differences in the edge sets and edges themselves form these distinctions. Graph objects may be multigraphs or simple graphs, binary graphs or hypergraphs, and undirected, directed, or "mixed" graphs. The latter may contain both undirected and directed edges simultaneously. Edges of directed hypergraphs in *Link* are defined simply as sequences of vertices. This definition is more general than that in [9], since not only is the sink known, but the ordering of the other vertices of the edge is available if needed.

The *Link* system also includes generation, layout, and and algorithms libraries. Algorithms may be viewed using an animation controller which provides many of the features of a graphical debugger, such as forward and backward stepping and the definition of breakpoints. Multiple views of the same graph are supported, and each view may launch animations.

3 Description of the Problem

Consider a set of transactions at a local supermarket. Customers purchase sets of items, and receipts are generated and stored for each purchase. It could be beneficial for the store manager to use this data in order to better understand shoppers buying habits. This process is known in the marketing community as *market basket analysis*. A system designed to explore the data should allow the manager to visualize the data, formulate questions and conjectures, extract pertinent subsets of data, and then complete the circle by visualizing the results and formulating new questions. For example, it might be useful to distinguish those receipts in which store-brand cereal was purchased instead of name-brand, and see if this provides a good classification of customers. The dataset could be broken into these classes, and each one could be examined for responsiveness to changes in price or location of items.

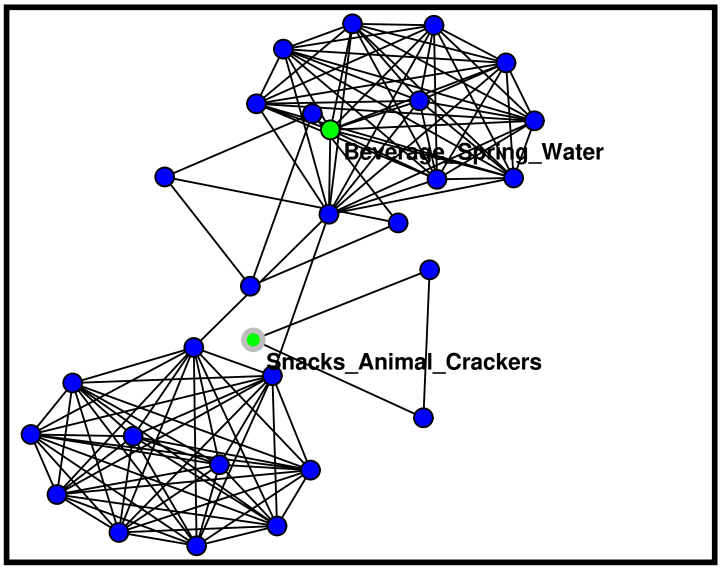

Figure 1: Simple graph model of transactions with at least one beverage and one snack

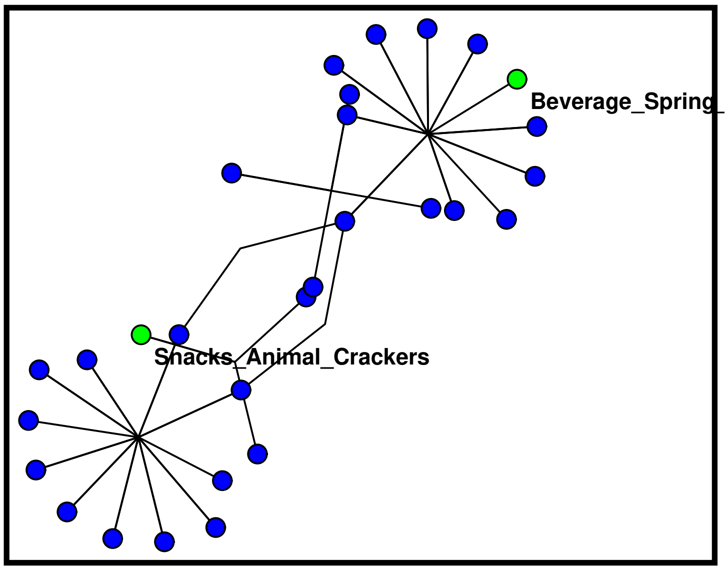

Figure 2: Hypergraph model of transactions with at least one beverage and one snack

One solution to the general problem of modeling the situation would be to use a simple graph with vertex and edge weights. A vertex represents an item purchased at the supermarket, while an edge represent the fact that two items were purchased in the same transaction. The weight of an edge might represent the percentage of receipts in which these two items appeared. Vertices could be assigned weights reflecting the number of times they were purchased. In fact, this solution has been implemented [4]. Changes in color and size distinguished vertices, and the user could filter out uninteresting vertices and edges, and zoom in to view interesting clusters of vertices.

Consider the implications of such a solution. Each transaction is modeled by a clique in the simple graph, which is very space-inefficient and makes the model needlessly dense. Furthermore, the fact that the items of a given receipt were all purchased in the same transaction is lost since the graph only models binary relationships. Clearly, the problem could be more effectively and efficiently modeled using a hypergraph.

Let us define a hypergraph $H = (V_H, E_H)$, where the vertices represent items purchased and the edges represent transactions. The degree of a vertex therefore represents the number of times the corresponding item was purchased while the size of an edge indicates the number of items purchased in the corresponding transaction.

Figure 1 depicts a simple graph model of an interesting substructure of sample dataset, while Figure 2 shows a hypergraph representation of the same substructure. [1] The substructure of the dataset was obtained using the query depicted in Figure 5 and briefly explained on Page 4 The embeddings show are spring embeddings, where the spring embedding of a hypergraph H is determined by embedding the underlying bipartite representation of H. We define the latter to be the graph $G_H = (Z, F)$, where $Z = V_H \cup Y, Y = \{y_e : e \in E_H\}$, and F consists of those pairs (x, y_e) where $x \in V_H$, $y_e \in Y$, and $e \in E_H$. If w is a weight function on E_H, then edge $(x, y) \in F$ has weight $\frac{w(e_H)}{2}$, where e_H is the hyperedge which corresponds to y. This weighting is intuitive if the weight of a hyperedge e_H is considered to be the cost of traveling from $v_i \in e_H$ to $v_j \in e_H$. The same movement and cost are achieved in G_H by moving from v_i to the vertex representing e_H, then to v_j.

```
(define b (make <button> :text "double vertex size"
       :expand #t :fill 'x                      ; resizable
       :parent (slot-ref hv 'graph-toplevel)    ; STk window
       :command '(map (lambda (x) (set! (size x) (* (size x) 2)))
                      *link:selected-vertices*)))
```

Figure 3: Defining a Button with Callback Function

In addition to being far less dense than the simple graph model, the hypergraph model contains more information. We use the latter to demonstrate *Link*'s flexibility and power. Assume that the hypergraph is named h and the graph window displaying h is named hv. Simple commands such as

```
(map (lambda(x)
   (set! (size x) (quotient (degree x) 3)))
     (vertices hv))
```

Figure 4: Setting Vertex Attributes (*map* applies its first argument (a function) to each of the elements in its second argument (a list))

those resizing vertices are easily incorporated into the GUI using STklos. For example, the command in Figure 3 creates a button which will double the size of any vertices selected by the user. The details inherent in this example are explained in the STk and *Link* documentation, but it is more instructive simply to note that it is possible to modify the user interface by typing a few lines of Scheme-Tk code.

To continue with our supermarket example, it is straightforward to resize the vertices according to their degrees. The maximum degree of h is 61, and let us say that the maximum size will be 20 screen pixels. The simple code in Figure 4 performs the task.

The store manager, upon viewing the graph, may select groups of vertices using the mouse and bring up windows containing induced subgraphs of h, where the vertices of an induced edge e' are those contained in the intersection of the original edge e and the set of vertices defining the subgraph. Substructures may also be specified in more interesting ways. For example, suppose that we want to identify the minimal set of vertices involved in transactions in which at least one snack item and one beverage item were purchased. The query in Figure 5 will compute and display the subgraph induced by this set of vertices.

[1] The data are authentic. The simple graph model contains 582 vertices and 2668 edges. Unfortunately, however, the original receipt information was not available. The hypergraph was created by repeatedly extracting maximal cliques from the simple graph, so although it is still useful as a testcase to illustrate *Link*'s power, it does not reflect actual transactions. The hypergraph contains 582 vertices and 649 hyperedges. The average size of a hyperedge is 2.79, and the maximum size is 12.

```
(define verts (set-vertex))
(define bev-test (string->regexp "Beverage"))  ;STk regular expression
(define snack-test (string->regexp "Snack"))

(define (has-snack vlist)        ;was a snack purchased?
        (cond ((null? vlist) #f)
              (#t (or (snack-test (vertex-label (car vlist)))
                      (has-snack (cdr vlist))))))

(define (has-bev vlist)          ;was a beverage purchased?
        (cond ((null? vlist) #f)
               (#t (or (bev-test (vertex-label (car vlist)))
                       (has-bev (cdr vlist))))))

(map (lambda (e)                 ;add items of transaction if
                                 ;≥ 1snack and ≥ 1beverage
               (if (and (has-bev   (set-vertex->list (vertices e)))
                        (has-snack (set-vertex->list (vertices e))))
                     (set! verts (+ verts (vertices e)))))
     (set-edge->list (edges h)))

        ;create induced subgraph
        ;1 → copy attributes
(define has-snack-or-bev (induced-subgraph h verts 1))
(define hsv (make <graph-view> :graph has-snack-or-bev
                              :layout 'spring))
```

Figure 5: Identification and viewing of an interesting substructure: The Subgraph induced by all vertices which belong to transactions in which at least one beverage and one snack were purchased.

4 Final Remarks

The *Link* system provides very high-level access to many combinatorial objects that have interesting properties and applications, but rarely appear in the literature or in practice due to the complexity of implementation. The system has been designed to provide a flexible set of tools not only to visualize graph properties and algorithms, but also to support series of computations.

The most crucial design decision of the *Link* system was to favor flexibility over speed. Instead of opting for compiled, low-level interface such as Motif, designers have based *Link*'s interface on the interpretive STk package. Applications written in Motif usually consist of thousands of lines of low-level function calls, requiring a competent C/XWindows programmer for any modifications. In contrast, Figure 3 demonstrates a five line run-time STk command resulting in a new button which calls a user-defined function. For a broad range of applications, flexibility like this is worth the trade-off in speed. As time goes on, machines become faster, and software is is improved, *Link* will become faster, while inflexible software will remain inflexible.

Link will have many applications, ranging from education at many levels, to interactive experimentation aiding researchers, to prototyping for industry. It is available at no charge for teaching and research. See

`http://dimacs.rutgers.edu`

for more information.

References

[1] G. Di Battista, P. Eades, and R. Tamassia. Algorithms for drawing graphs: an annotated bibli-

ography. Technical report, Department of Computer Science, Brown University, Providence, RI, 1993.

[2] 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.

[3] 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.

[4] S. .G. Eick and G. J. Wills. Navigating large networks with hierarchies. In *IEEE Visualization*, pages 204–210, 1993.

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

[6] E. Gansner, S. North, and K. Vo. Dag – a program that draws directed graphs. *Software – Practice and Experience*, 18(11):1047–1062, 1988.

[7] D. Knuth. *The Stanford Graphbase: a platform for combinatorial computing.* Addison-Wesley, Reading, Massachusetts, 1993.

[8] E. Koutsofios and S. C. North. Applications of graph visualization. Manuscript.

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

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

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