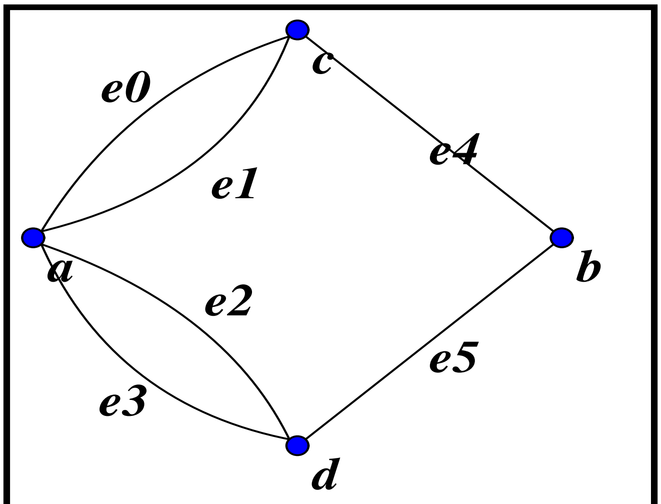
# An Informal Preview of the *Link* * System

Jonathan Berry
DIMACS Center†

April 23, 1996



***The Koenigsberg Bridges***

## 1 Introduction

This document gives a preview of *Link*, a software system for graph theory and combinatorics which will be useful in both research and education. It should be

*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. It has been partially funded by NSF grant CCR-9214487 and ONR award 400x116yip01. Many people have contributed to Link. This unpublished preview contains text from [2], which lists the contributers. *The figure above is postscript output from Link.*

†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

especially useful for algorithms and discrete math courses. A public release is planned for July 1996. Link will be freely available for teaching and research. If you wish to become an alpha tester before that date, please contact Jonathan Berry, the project coordinator at `berryj@dimacs.rutgers.edu`. The system currently runs on SunOs and linux, but a port to Windows is planned.

The foundation of Link is a set of templated C++ libraries which provide access to a rich variety of *Collection* and *Graph* objects. It is straightforward to use these libraries in new C++ applications, and there is a programmer's manual complete with class descriptions and coding examples, some of which are reproduced in Section 5.

But Link is much more than a set of class libraries. Link has a flexible interface based upon Eric Gallesio's *STk* (Scheme-Tk) package [1] [4]. STk supports a variant of the Common Lisp Object System, which allows object-oriented techniques to be used during interface design. Examples of Scheme code which accesses Link objects are included in Section 4. A user's manual explains the Scheme interface. Work on graphical user interface using Tk is under way at the time of this writing.

## 2 Background

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 [3, 11, 6].

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

[1] STk substitutes Scheme for Tcl as the command language for John Ousterhout's Tk graphics toolkit.[8]

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.

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 [9], Nate Dean was largely responsible for the package NETPAD at Bellcore [7] , Steve Skiena at State University at New York at Stony Brook had produced Combinatorica on top of Mathematica [10], and Mark Goldberg and his students at Rensselaer Polytechnic Institute had implemented SetPlayer [1]. 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.

Currently, the project is headed by Jonathan Berry, a postdoctoral fellow at The Center for Discrete Mathematics and Theoretical Computer Science at Rutgers University. The Tcl and Motif interfaces are being replaced with STk interface. which is easier to use and extend. All comments and suggestions should be sent to `berryj@dimacs.rutgers.edu`.

# 3 The Basic Objects

The basic objects of the Link system are instances of *Container*, *Collection*, and *Graph* classes. *Containers* are templated abstract data types such as *List*<int>, *Array*<char>, etc. The Link contains several useful *Containers* and documentation describing how new *Containers* can be added.

Graphs and hypergraphs are collections of sets, and three parameters determine a graph's type: whether or not every edge has cardinality 2, whether or not each set is directed (ordered), and whether or not multiple occurrences of the same edge are permitted. The kernel of the Link system is a hierarchy of *Collection* objects designed to facilitate these distinctions. Figure 3.1 depicts this hierarchy.

The three basic types of *Collection* are multiset (*MSet*), set (*Set*), and sequence (*Sequence*). The classes with names ending in *Base* are parameterized by implementation type as well as element type. For example, *MSetBase* objects might be implemented using lists, arrays, bit vectors, etc.

Grouping *Collection* objects into a hierarchy allows flexible interaction between these objects. For example, graph methods such as *neighbors* are coded only once, yet work for graphs with any type of edge.

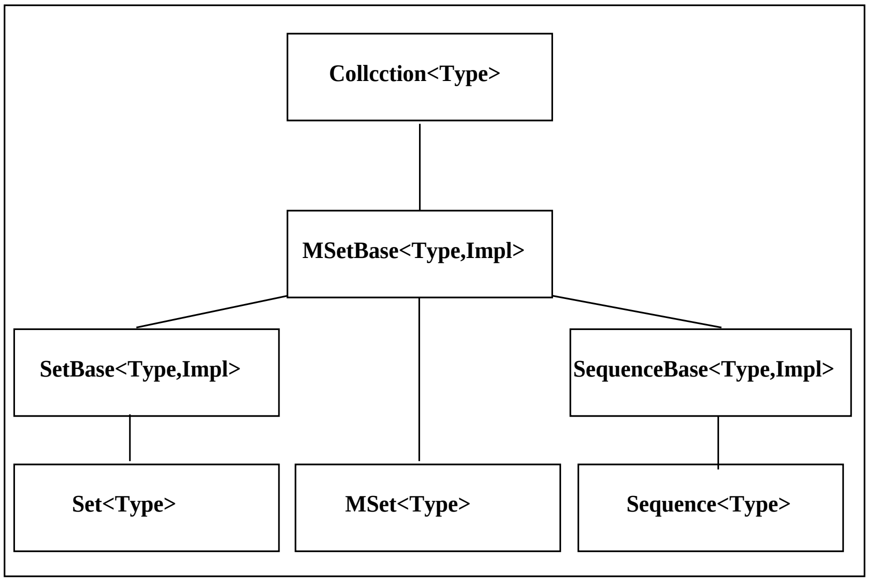



Figure 3.1: The LINK *Collection* Hierarchy

The hierarchy of *Graph* objects in Link is shown in Figure 3.2. The type of *Collection* used to implement each edge is shown in parentheses below each graph type. In reality, each class represented in the figure (other than *Graph*, which is abstract) represents two graph classes: one which allows multiple occurrences of the same edge, and one that does not. For example, instances of the class *MDHyperGraph* are graphs with edge sets which are multisets of sequences, where each sequence has cardinality at least one. On the other hand, the edge sets of *UBinGraph* objects are sets of sets, where each set has cardinality two.

This rich variety of graph types is designed to be a general-purpose tool for attacking a large number of graph applications.

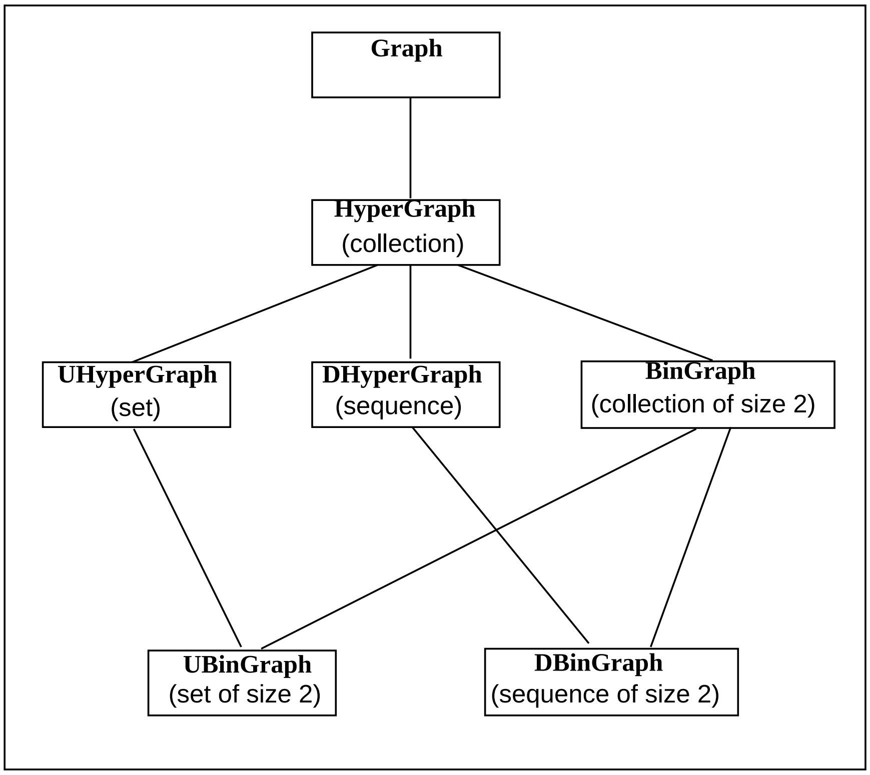



Figure 3.2: The LINK *Graph* Hierarchy. Each class name can be preceded by "M" to allow multigraphs of that graph type.

```
STk> (define g (graph))
#[undefined]

STk> (describe g)
#[<ubingraph*> #p5658cc] is an instance of class <ubingraph*>
Slots are:
     val = {[] {}}
#f
```

Figure 4.1: Creating an empty graph

# 4 The Scheme-Tk Interface

The basic objects of Link can now be created and manipulated in a Scheme-Tk environment, thanks to the STk package [4]. The examples in this section are the output of an automated process which feeds input files to Link's STk interpreter and formats the output in LaTeX. [2]. They are excerpts from the user's manual of Link.

Note the heavy influence of object-oriented techniques in these examples. For example, Figures 4.1, 4.3, and 4.4 show extensive overloading of the *graph* method to provide the user with a flexible graph-creation command. Another good example is found in Figure 4.5. The operators "+," "^ ," and "-" have been overloaded to accept *Collection* arguments. Furthermore, the *Collection* hierarchy has been partially duplicated in Scheme's object system, allowing, for example, multiple types of *Collection* to be passed as arguments to the same function call.

A key feature of Link will be the ease in which output from one method can be used as input to another. This will allowing functional computing in the style of Steven Skiena's *Combinatorica* package [10].

[2] This is accomplished using Don Libes' *expect* extension of Tcl [5]

```
STk> (define dg (digraph '(1 2 3 4) '((1 2 3) (2 3 4))))
#[undefined]

STk> (define n (neighbors (vertex-ref 1 dg)))
#[undefined]

STk> (describe (vertex-ref 1 dg))
#[<vertex*> #p55a234] is an instance of class <vertex*>
Slots are:
    val = 2
#f

STk> (describe n)
#[<set<vertex*>> #p564968] is an instance of class <set<vertex*>>
Slots are:
    val = {1 3 4}
#f

STk> (define in (in-neighbors (vertex-ref 1 dg)))
#[undefined]

STk> (describe in)
#[<set<vertex*>> #p55892c] is an instance of class <set<vertex*>>
Slots are:
    val = {1}
#f

STk> (define out (out-neighbors (vertex-ref 1 dg)))
#[undefined]

STk> (describe out)
#[<set<vertex*>> #p557324] is an instance of class <set<vertex*>>
Slots are:
    val = {3 4}
#f
```

Figure 4.2: Digraph creation and manipulation

```
STk> (define g (graph '(1 2 3) '((1 2) (1 3) (3 1))))
#[undefined]

STk> (map (lambda (x)
            (find-attribute 'vcolor x))
          (set-vertex->list (vertices g)))
(1 1 1)
```

Figure 4.3: Graph attributes: finding values

```
STk> (define g (graph '(1 2 3) '((1 2) (2 3) (3 1))))
#[undefined]

STk> (define h (graph '(1 2 3) '((1) (2 3) (1 2 3))))
#[undefined]

STk> (define h2 (graph (+ (vertices g) (vertices h))
                       (+ (edges g) (edges h))))
#[undefined]

STk> (describe h2)
#[<muhypergraph*> #p55fad8] is an instance of class <muhypergraph*>
Slots are:
    val = {[1 2 3] {{1} {1 2} {1 2 3} {1 3} {2 3} {2 3}}}
#f
```

Figure 4.4: Computing the sum of two graphs

```
STk> (define ms (mset-int 3 2 2 1))
#[undefined]

STk> (define s (set-int ms))
#[undefined]

STk> (define sq (sequence-int ms))
#[undefined]

STk> (define ms2 (+ ms s sq))
#[undefined]

STk> (define ms3 (^ ms s sq))
#[undefined]

STk> (define ms4 (- ms2 ms3))
#[undefined]

STk> (describe ms2)
#[<mset<int*>> #p558284] is an instance of class <mset<int*>>
Slots are:
    val = {1 1 1 2 2 2 2 2 3 3 3}
#f

STk> (describe ms3)
#[<mset<int*>> #p55619c] is an instance of class <mset<int*>>
Slots are:
    val = {1 2 3}
#f

STk> (describe ms4)
#[<mset<int*>> #p554924] is an instance of class <mset<int*>>
Slots are:
    val = {1 1 2 2 2 2 3 3}
#f
```

Figure 4.5: The set primitive operations

```
STk> (define g (graph '(1 2 3) '((1 2) (1 3) (3 1))))
#[undefined]

STk> (set-attribute! 'vcolor  3 (vector-ref (vertex-vector g) 1))
#[undefined]

STk> (map (lambda (x) (find-attribute 'vcolor x)) (set-vertex->list (vertices g)))
(1 3 1)
```

Figure 4.6: Manipulating attributes

```
STk> (define g (graph '(1 2 3 4) '((1 2) (1 3) (1 4))))
#[undefined]

STk> (define (max-vert l)
        (cond ((null? l) 0)
              (#t (max (degree (car l) g) (max-vert (cdr l))))))
#[undefined]

STk> (define-generic delta)
#[undefined]

STk> (define-method delta ((g <graph*>))
        (let ((v (set-vertex->list (vertices g))))
                (max-vert v)))
delta

STk> (delta g)
3
```

Figure 4.7: Example definition of a graph function: $\Delta(g)$

```
STk> (load "graph-view.stklos")
defined#[undefined]

STk> (define konigsberg (graph '(a b c d)
                              '((a c) (a c) (a d) (a d) (b c) (b d))))
#[undefined]

STk> (describe konigsberg)
#[<mubingraph*> #p5cb904] is an instance of class <mubingraph*>
Slots are:
    val = {[a b c d] {{a c} {a c} {a d} {a d} {b c} {b d}}}
#f

STk> (define kview (make <graph-view> :graph konigsberg))
#[undefined]
```

Figure 4.8: The Bridges of Königsberg Problem

```
STk> (load "graph-view.stklos")
defined#[undefined]

STk> (define g (graph '(1 2 3) '((1 2) (2 3) (1 3))))
#[undefined]

STk> (define gv (make <graph-view> :graph g))
#[undefined]

STk> (define tree-edges (kruskal g))
#[undefined]

STk> (define t (graph (vertices g) tree-edges))
#[undefined]

STk> (define tv (make <graph-view> :graph t))
#[undefined]
```

Figure 4.9: Algorithms Example: Finding and viewing a minimum spanning tree

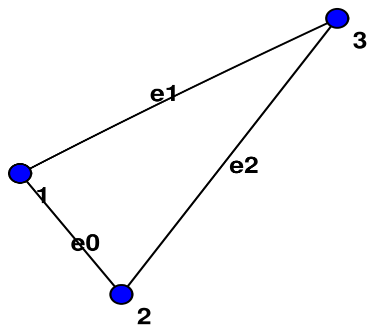



Figure 4.10: Minimum spanning tree example: original graph

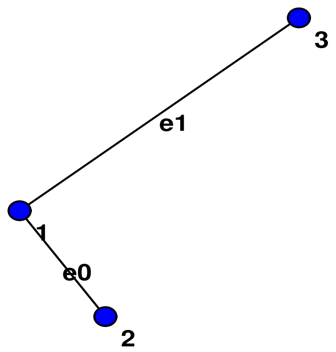



Figure 4.11: Minimum spanning tree example: the spanning tree

```
STk> (load "graph-view.stklos")
defined#[undefined]

STk> (define dhg (digraph '(1 2 3 4) '((1 3 2) (4 1) (4 1))))
#[undefined]

STk> (describe dhg)
#[<mdhypergraph*> #p5f6e18] is an instance of class <mdhypergraph*>
Slots are:
     val = {[1 2 3 4] {<1 3 2> <4 1> <4 1>}}
#f

STk> (define dhgv (make <graph-view> :graph dhg))
#[undefined]

STk> (define egs (set-edge->list (edges dhg)))
#[undefined]

STk> (set-attribute! 'weight 10 (caddr egs))
#[undefined]

STk> (define tree-edges (kruskal dhg))
#[undefined]

STk> (map (lambda (x)
        (let* ((etab (slot-ref dhgv 'edge-table))
               (eg (hash-table-get etab x)))
            (slot-set! eg 'width 4))) (set-edge->list tree-edges))
(#[undefined] #[undefined])
```

Figure 4.12: Directed hypergraphs and minimum spanning trees. Note that the same spanning tree algorithm works for various graph types.

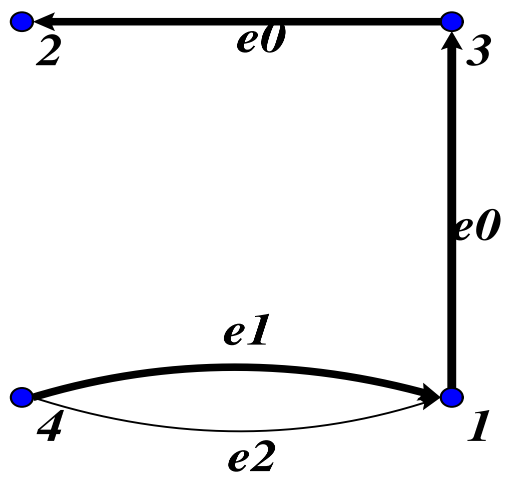



Figure 4.13: The graph from Figure 4.12 with the minimum spanning tree highlighted.

```
STk> (load "graph-view.stklos")
defined#[undefined]

STk> (define konigsberg (graph '(a b c d)
                          '((a c) (a c) (a d) (a d) (b c) (b d))))
#[undefined]

STk> (define vxcoords (list->vector '(10 250 130 130)))
#[undefined]

STk> (define vycoords (list->vector '(120 120 10 230)))
#[undefined]

STk> (define kview (make <graph-view> :graph konigsberg
  :xcoords vxcoords
  :ycoords vycoords))
#[undefined]

STk> (define (set-font x)
       (set! (font x) "-*-times-bold-i-normal--18-240-*-*-*-*-*-*"))
#[undefined]

STk> (map set-font (vertices kview))
(#[undefined] #[undefined] #[undefined] #[undefined])

STk> (map set-font (edges kview))
(#[undefined] #[undefined] #[undefined] #[undefined] #[undefined] #[unde-
fined])

STk> (define title (make <text-item> :text "The Koenigsberg Bridges"
  :parent (slot-ref kview 'graph-canvas)
  :coords '(150 280)
:font "-*-times-bold-i-normal--24-240-*-*-*-*-*-*"))
#[undefined]

STk> (define bound (make <rectangle>
                :parent (slot-ref kview 'graph-canvas)
                :coords '(3 3 297 297)
                :width 3
                :fill "white"))
#[undefined]

STk> (lower bound)
()
```

Figure 4.14: The code to produce the title page figure

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

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

        for (int i=0; i<3; i++)
                mset1.append(i);

        mset1.insert(0);
        mset2 = mset1;          // reference count increment
        mset2.append(0);        // copy

        cout << mset1 << endl;
        cout << mset2 << endl;
}
```

```
{0 0 1 2}
{0 0 0 1 2}
```

Figure 5.1: Reference counting and modifications in *Collections*

# 5 The C++ Libraries

If performance is important enough to rule out the use of Scheme as an implementation language, the programmer (upper-level undergraduate, graduate, or experimental researcher) can implement algorithms in C++. These new algorithms can then be "wrapped" to make them accessible from the STk interface. The wrapping process, which will be largely automated, will be described in the programmer's manual.

The Link objects are available via libraries, which may be included in user applications. This section provides some examples which were taken from the programmer's manual. These examples were generated automatically by a script which compiles and runs the program, then formats the output.

Link was designed with efficiency in mind as well as flexibility. For example, Figure 5.1 illustrates the fact that Link *Collections* use reference counting to obviate the need to make copies whenever possible. The example is commented to show an efficient assignment of one *MSet* object with another via reference counting. After the assignment, both objects share the same elements. Only when one of the multisets is changed will an actual copy occur. More general examples are described in the programmer's manual.

The remaining examples give some of the flavor of Link programming. *Col-*

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

main()
{
        MSet<int> list_set1;

        for (int i=0; i<3; i++)
                list_set1.append(i);

        //*************reference count is incremented*********
        MSet<int> list_set2(list_set1);

        //*************complete copy is made******************
        MSetBase<int, SortedArray<int> > array_set(list_set1);

        cout << list_set1 << endl;
        cout << list_set2 << endl;
        cout << array_set << endl;
}
```

```
{0 1 2}
{0 1 2}
{0 1 2}
```

Figure 5.2: Using alternative set implementations

*lections* are easily created and manipulated, as are *Graph* objects. Not shown are examples of Link's libraries of graph algorithms and generators. These will be detailed in the programmer's manual.

The Link system will be a valuable aid not only to education, but to practicioners as well. It is scheduled to be released to the public (for teaching and research) in July, 1996. By that time, there will be a web page and a mailing list set up to help organize a Link community. There will be an extensible GUI written in Tk, documentation in the form of user's and programmer's manuals, and a protocol for contributing extensions and obtaining new versions.

# 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, N. Dean, P. Fasel, M. Goldberg, E. Johnson, J. MacCuish, G. Shannon, and S. Skiena. Link: A combinatorics adn graph theory workbench for applications and research. Technical Report 95-15, Center for Discrete Mathematics and Theoretical Computer Science, Piscataway, NJ, 1995.

[3] 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.

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

[5] D. Libes. *Exploring Expect*. O'Reilly & Associates, Inc., 1995.

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

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

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

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

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

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

main()
{
        MSet<int> mset1;
        Set<int> set;
        Sequence<int> seq1;

        for (int i=0; i<3; i++) {
                mset1.insert(i);
                mset1.insert(i);
                seq1.insert(2*i);
                seq1.insert(2*i);
        }
        mset1 ^seq1;
        cout << "MSet receives return value" << endl;
        cout << mset1 << " + " <<seq1 <<" = " << mset1+seq1 << endl;
        cout <<seq1  << " + " << mset1<<" = " <<seq1+mset1 << endl;
        cout << mset1 << " ^ " <<seq1 <<" = " << (mset1^seq1) << endl;
        cout << seq1 << " ^ " << mset1<<" = " << (seq1^mset1) << endl;
        cout << mset1 << " - " <<seq1 <<" = " << mset1-seq1 << endl;
        cout << seq1 << " - " << mset1<<" = " <<seq1-mset1 << endl;
        cout << "Set receives return value" << endl;
        cout << mset1 << " + " <<seq1 <<" = " << (set=mset1+seq1) << endl;
        cout << mset1 << " ^ " <<seq1 <<" = " << (set=(mset1^seq1)) << endl;
        cout << mset1 << " - " <<seq1 <<" = " << (set=mset1-seq1) << endl;
}
```

```
MSet receives return value
{0 0 1 1 2 2} + <4 4 2 2 0 0> = {0 0 0 0 1 1 2 2 2 2 4 4}
<4 4 2 2 0 0> + {0 0 1 1 2 2} = {0 0 0 0 1 1 2 2 2 2 4 4}
{0 0 1 1 2 2} ^ <4 4 2 2 0 0> = {0 0 2 2}
<4 4 2 2 0 0> ^ {0 0 1 1 2 2} = {0 0 2 2}
{0 0 1 1 2 2} - <4 4 2 2 0 0> = {1 1}
<4 4 2 2 0 0> - {0 0 1 1 2 2} = {4 4}
Set receives return value
{0 0 1 1 2 2} + <4 4 2 2 0 0> = {0 1 2 4}
{0 0 1 1 2 2} ^ <4 4 2 2 0 0> = {0 2}
{0 0 1 1 2 2} - <4 4 2 2 0 0> = {1}
```

Figure 5.3: The set primitive operations with set and sequence operands

```
#include<iostream.h>
#include<LINK/graph/BinGraph.h>
#include<LINK/basic/Sequence.h>


main()
{
        BinGraph g;
        Sequence<Vertex*> sq;
        Set<Vertex*>      s;
        Vertex *v1, *v2;

        v1 = g.addVertex("v1");
        v2 = g.addVertex("v2");
        s.append( v2); s.append( v1);   // automatically sorted
        sq.append(v2); sq.append(v1);   // programmer determines order
        g.addEdge(s);
        g.addEdge(sq);
        cout << g << endl;
}
```

```
{[v1 v2]  {{v1 v2} <v2 v1>}}
```

Figure 5.4: "Mixed" Binary Graph

```
#include<iostream.h>
#include<LINK/graph/DBinGraph.h>
#include<LINK/basic/Sequence.h>


main()
{
        MDBinGraph mg;
        DBinGraph g;
        Sequence<Vertex*> s;
        Vertex *v1, *v2;

        v1 = mg.addVertex("v1");
        v2 = mg.addVertex("v2");
        s.append(v1); s.append(v2);
        mg.addEdge(s); mg.addEdge(s);
        cout << mg << endl;

        s.clear();

        v1 = g.addVertex("v1");
        v2 = g.addVertex("v2");
        s.append(v1); s.append(v2);
        g.addEdge(s); g.addEdge(s);
        cout << g << endl;
}
```

```
{[v1 v2]  {<v1 v2> <v1 v2>}}
{[v1 v2]  {<v1 v2>}}
```

Figure 5.5: Directed binary multigraph

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