Improvements to GraphPack: A System to Manipulate Graphs and Digraphs

M. KRISHNAMOORTHY, A.SUESS, M. ONGHENA,
F. OXAAL AND T.SPENCER

Abstract

This paper describes the improvements that we have carried out on GraphPack, the system that we have developed here. These improvements are specifically four fold. We have incorporated GraphPack to communicate with other systems; specifically, Mathematica, Maple, Matlab and SetPlayer. Further, we have incorporated commands to perform graph algorithms on MasPar,a SIMD machine. A lot of display routines are added for different embeddings of graphs. We also incorporated a 3D display of graphs, including rotation and swapping of vertices to get a better embedding. Finally, the language LiLa and the interpreter has been enhanced so as to write application programs easier.

1 Introduction and Problem Definition

As is often the case for software packages, modifications and additions are constantly desired. Such is certainly the case for GraphPack and this proved to be the main impetus for this paper, with concentration in several areas, each of which will be described in detail in this paper. The broad categories of modification are as follows:

- **Communication with External Programs**

  While GraphPack is able to execute many graph and set commands, there are numerous useful operations it cannot do and many of these might be

---

1991 Mathematics Subject Classification. Primary 05C85, 05C90, 68N20

Supported in part by the National Science Foundation under Grant Number CDA-8805910.

This is the final version of the paper

considered beyond the scope of GraphPack. However, a number of already existing programs could perform these functions with relative ease. For example, Mathematica, Maple and Matlab can perform many matrix operations and the Rensselaer program SetPlayer can compute the minimum vertex cover easily. GraphPack currently cannot execute any of these commands.

It was also found desirable to have the ability to send graphs to some user created programs. In this way, a user could add a feature to GraphPack without worrying about how things are stored within the "real" code.

Thus, the goal of this part of the program is to somehow incorporate the power of external programs into GraphPack in order to make this program much more useful and powerful.

- **Implementing Functions using the MasPar**

  Related to the problem described above, there is also interest in using the power of some available Rensselaer hardware to enhance GraphPack. This work concentrated on the SIMD MasPar machine. The algorithms implemented on the machine were those which could be executed extremely quickly on an adjacency matrix. Examples of commands which were added for the MasPar include **m_bridge** which finds the bridges in the graph, **m_minspan** which finds the minimum spanning tree of a graph, and **m_tsales** which finds a reasonable solution to the traveling salesman problem.

  Thus, the problem definition for this part of the project is to determine how to best "connect" GraphPack and MasPar, and then to examine which graph algorithms are most suitable for MasPar.

- **Additional Graphical Commands for XGraphPack**

  Before this, there were a number of functions in the X version of GraphPack which were either lacking or had functional problems. For example, previously if a user desired to display a new graph, displaying the graph would erase the already drawn graph. To correct this, a command which allowed the displaying of multiple graphs was created.

  Another desired enhancement was to allow users to get the positions of vertices from the graphics window. Users are then more easily able to find configurations they might desire.

  Of major importance for these commands is coordinating communication between the graphics window and the interpreter window.

  Also, software for the 3-D display of graphs was eveloped including rotations and other transformations.

- **Additional Interpreter Commands**

  As previously mentioned, users often come up with small additions which make software better or easier to use, and this proved true for GraphPack.

The assorted commands include searching for commands in help files, allowing vertices to be stored in numerical order and generating random weights for graphs. In particular, faster operations on graphs are carried out.

As far as the authors know, the *GraphPack* system is comparable to other existing systems such as Combinatorica, NetPad, and GraphView [10][9] [2]. This system is slightly better than other projects with a comparable theme of which we are aware; among them, Netsolve and Graphix [3]. Netsolve is a library of procedures; thus, it is of no help to users who want to solve problems that Netsolve does not provide algorithms for. Graphix, on the other hand, is a collection of primitive operations for graph algorithms. These are the kinds of operations that will appear in the Kernel. The Kernel of *GraphPack*, however, also contains elementary set operations; no elementary set operations are supported by Graphix. Graphix does not contain any programs at all. More recently, a user interface for graph and group algorithms was done in Groups and Graphs [4]. This system does not provide a textual interface to the system. The system GMP, developed at Michigan State University [11] does support an interactive window, but there is no textual or language support. Further, their system does not handle directed graphs. Another system, EDGE , [7] is an interactive graph editor developed at Karlsruhe; however, this system lacks library or language support. Yet another system, ISI grapher [8] is a portable software package that allows graphs to be displayed pictorially. This system does not provide a language support.

We like to think of the *GraphPack* system in the same spirit as Macsyma, Mathematica, or Maple but specifically useful for graph operations. Furthermore, our system has both a textual and graphical interface.

Our paper is organized as follows: In the next section, Overall System Design, we briefly describe all the blocks of the system that existed before. In the subsequent sections, we describe the new features added on to the system for communicating with other systems such as Maple, Mathematica (Combinatorica), and Matlab. Also, the section describes use of the SIMD parallel computer Maspar that is used to do some graph operations and how those programs communicate. Finally, the 3-D display of graphs and their rotations and other displays are described.

2 Overall System Design

The overall system, i.e., *GraphPack*, consists of four components: the User Interface, which contains the GraphPack language LiLa and its interpreter, the Kernel, the Library, and the Auxiliary Tool Component. The components are arranged in a layered fashion.

The language LiLa (Little Language), is based on C with additional graph and set theoretical primitives, such as set, graph, tree, etc. There is both a compiler and an interpreter, each of which function under suitable circumstances. We shall refer to these two together as the translator and consider it as a part of the User Interface (UI). The other parts of the User Interface are input-output routines and a menu facility for a list of commands, along with the possibility of command completion, and a debugger. Thus, the User Interface also provides a software development environment. Further, all the emacs (an editor) commands are available for the textual window. This enables the user to move and edit textual commands.

The Kernel contains routines written in C for the primitive operations on the data structures such as graphs, sets, stacks, queues and lists. The design of Kernel posed many interesting problems involving data-structures. The development required a substantial amount of preliminary research.

The Library contains the best implementation of various graph algorithms; among them are heuristic algorithms for traveling salesman, minimal vertex coloring, algorithm for matching, generation of all spanning trees, planarity testing and transitive closure, and path problems. The choice of modules was made in consideration of the available resources.

The Auxiliary Tools include shell tools, time-measurement routines, and a type checker. These three will be important for the users working in the analysis of algorithms.

3 Design Decisions and Implementation Details

3.1 Communication with External Programs

As this was the largest segment of the project, most of the design decisions were made in regard to the question of how to do this communication. Some previous work had been done in GraphPack which allowed GraphPack to execute a GraphPack command on another machine and return the results. The command **remote** (now obsolete), allowed users to send any command with any variable to a remote machine so that it could do the processing and then return the result. This command did not increase the true scope or power of GraphPack as it could only possibly speed up processing; however, it did provide the ground work for this project which did increase the scope and power of GraphPack.

What needed to be decided was what functions should the users be allowed to do with this external communication. The following commands were found to be necessary:

- Executing the server
- Executing a single remote command in Mathematica, Maple, Matlab or SetPlayer
- Shortcut commands for the single remote command
- Allowing multiple commands in the same session with Mathematica, Maple, Matlab or SetPlayer
- Allowing User-Created Programs to be executed from GraphPack
- Run User-Created X Window programs
- Run XMaple

A fundamental requirement for one wishing to do work on a remote machine is that the local computer can "talk" to that remote machine. This becomes even more difficult when the communication must be two way. The realistic possible options for doing this basically boil down to one ;that is using sockets. Indeed, this was the technique used for implementing the communication with GraphPack. The specific code needed to implement the socket will not be described here, however it is important to understand, in general, how the communication is initiated and maintained. Here, we assume that the reader is familiar with certain aspects of UNIX systems programming (i.e., fork, hosts,sockets and dup).

Server information

The local execution of GraphPack communicates over a socket with a program we shall call the server. The server is responsible for creating the socket, waiting for any remote attempts to use the socket and then communicating through it. More specifically, the GraphPack server, once communication is initiated, attempts to determine what the user wishes to do remotely. This is necessary because there are a number of programs that may be run through the server including user created programs.

The server is run in one of two ways. 1) Log into the machine where the server should be run and enter:

kerneld

or from the LiLa prompt from within GraphPack one can enter:

addhost(machine);

which will start the server on the machine indicated as long as it is in the users .rhosts file.

Currently, the GraphPack server knows the directories of these five programs: GraphPack, Mathematica, Matlab, SetPlayer, and Maple. However, because it was also necessary to perform other operations from the server, it became neccessary to somehow inform the server which operation needed to be done.

Two lines of input are sent to the server to indicate which job to perform. The first line was one of three things:

- **X**
  This indicates that the program to be run is probably a X– windows program and the user does not need to wait for output. Thus, the program will be forked. This occurs when the user used the **fork** command.
- **D**
  This indicates that the user wishes to change the display name of the server. By the default the X–windows display name is set to unix:0. However, it may be desirable to change this so that one may show another user a graph remotely. The command which does this is **setdisplay(displayname)**.
- **blankline**
  This indicates that the next line is the program to run and that the user will wait for a result. This is used for the **execute** command and any related commands.

The second line indicates one of three things also:

- **One of the six known programs**
  These are gp, math, maple, matlab, setp or xmaple. Respectively, these stand for GraphPack, Mathematica, Maple, Matlab, SetPlayer or XMaple.
- **The name and directory of a program to be run**
  Generally this is user created programs which need GraphPack graphs sent to them.
- **The display name the server should have**
  This is only used for the **setdisplay** command.

One note of great importance is what the server does to the input and output for the program run. Both input and output are redirected through the socket using the C **dup** function.

The specific type of socket used was a stream socket.

A final note regarding the server is what it does to indicate that a process has been completed. The server always forks a process to wait for the program to complete and when it completes it sends a message over the socket and then closes the socket connection. For X window programs, this message is sent immediately over the socket to tell GraphPack to stop waiting.

The server may be killed with the command:

killhost(hostname);

Single remote commands

Once the server is started, it is then possible to execute a command in a remote program. The basic form of this function is:

execute (program name, host name, command);

To work correctly, the server must be running on the host indicated; the program name must be one of **gp, math, maple, setp, matlab** or the full pathname of an executable, and the command must be a valid command of the program run remotely.

The implementation details behind the **execute** command can be summarized as follows:

First, the connection between the server and GraphPack is initiated. Next, GraphPack sends the server a message over the socket as to which program to run as described in the server section. The server then runs the indicated program while also changing all input and output for the program to go through the socket.

Lastly, GraphPack attempts to send the command over the socket. However, it must first parse the command for any variables (which need to be converted to form the remote program "understands") before being sent over the socket. For example, a graph variable may need to be converted to a Mathematica or Maple adjacency matrix. Currently the only variables converted are graph, integer, float and string variables.

In addition, the command is also parsed for some "special" functions. For Mathematica, two special functions exist – cpoly and cnum, which find the chromatic polynomial and chromatic number of a graph, respectively. They are special because they are not true Mathematica functions. GraphPack loads Mathematica with a macro file which has functions taken from the Combinatorica [10]. Once loaded, the polynomial and number of a graph may be found.

Also, for SetPlayer the function **mincov** exists. This function calls SetPlayer iteratively until a set of vertices which cover the graph are found.

For example, these commands could be used as follows:

```
<lila> addhost(vega);
<lila> set(g,random_graph(5));
<lila> set(a1,execute(math,vega,cpoly(g)));
<lila> set(a2,execute(math,vega,cnum(g)));
<lila> set(a3,excute(setp,vega,mincov(g)));
```

As indicated by the above examples, the execute command also returns a result. To do this, it must read the result through the socket. Determining where the end of the output is a critical issue. The general rule of implementation was to search the beginning of each line for a string which the server would send over the socket to indicate the end of a process. Another method used for Matlab and SetPlayer was to search for the prompt. When it was found, the output was the string of characters before that string and any prompt.

Unfortunately, there are a couple of factors which could invalidate results. The first is, if that string is a part of the result, then any characters between that string and the real prompt would be excluded. Also, in a program such as Matlab, it is possible that the program will ask a question if the command is entered incorrectly with a command-dependant prompt. Thus, the process would never end because it is waiting for a response. Both of these occurrences were found to be rare enough to be acceptable.

A technical point concerning the output reading is that the remote program must have the output file line or character buffered. This means that the output will appear on the socket with each outputted line or character as opposed to each block. Fortunately, GraphPack, Maple, Mathematica, Matlab and SetPlayer all do this now. (The C code which does this is – setlinebuf(NULL); or setbuf(output,NULL);)

Another question regarding the result of the **execute** command is what type does GraphPack store it as. If the result is a single integer or float, then that will be how it is stored. In all other cases the result will be a character string.

A final point regarding getting the result from the socket is how long to wait for an answer. By default, GraphPack will wait 30 seconds for a complete answer. If none is found in that time a timeout error will occur. The timeout can be set manually from within GraphPack with the **settimeout** command. Syntax is:

settimeout(integer).

Executing Shortcut Commands

Because of the wordiness of the **execute** command, it was soon found to be necessary to add some shortcut commands for the known programs. The syntax of these functions is:

rgp(command);

rmath(command);

rmaple(command);

rsetp(command);

rmatlab(command);

These commands work exactly as **execute** works by allowing a default host. The default host is used by the shortcut commands as the assumed server to be used. This default is set in one of two ways. A call to **addhost** will set the default host to the newly started server. It is also possible to switch between servers with the command **sethost**. Its syntax is:

sethost(hostname).

Thus, aside from the default host, the shortcut commands work exactly as the **execute command**.

Executing Multiple Remote Commands

The **execute** and shortcut commands were found to be lacking because they only allowed one command to be executed in the remote session. Thus, it became desirable to allow interactive sessions with these programs, but still through the remote server and host. The commands which were added to do this are:

gomatlab;

gomath;

gomaple;

gosetp.

These commands allow muliple commands to be entered in the same session with one of these four programs. GraphPack was excluded as it seemed unnecessary to allow multiple commands in GraphPack remotely.

A final point for multiple commands is how to stop. To provide compatibility between the commands, any of the commands: **exit, quit, done, Exit, Quit or bye** will cause the remote program to terminate and will return the user to GraphPack.

Executing User Created Programs

Obviously, as described above, the **execute** and related commands are limited to the programs which are included in the server; namely, Mathematica, Maple, SetPlayer and Matlab. While these are all powerful programs, their scope is still far from infinite. Furthermore, a user might wish to execute a program on a graph without having a programmer add it internally to GraphPack which could take some time. Thus, it became desirable to allow GraphPack to execute user created programs which could do some more functions on graphs.

This was done by allowing the user to enter the name of the program to be run as the first parameter to the **execute** command. The user can enter the full path name of the file or the file with a path if the file is found in the current directory or the default path. The default path may be set with:

setpath(path).

A difference between this and the normal **execute** command is that it can take two to infinity parameters. Many graphs may be sent to the remote user created program.

A sample run using the commands described in this section follows:

<lila> **addhost(berra);**
<lila> **set(g1,random_graph(5));**
<lila> **set(g2,random_graph(5));**
<lila> **set(a1,execute("sample1",berra,g1));**
<lila> **setpath("/profs/cs2/le");**
<lila> **set(a2,execute("sample2",berra,g1,g2));**

Forking X Windows Programs

Unlike the commands described above, it was found to be desirable to execute some X Windows programs without waiting for a reply. These programs would display their results so it was unnecessary for GraphPack to wait for them to conclude. The syntax of the command created is:

fork(program name, host name, [graph]*).

Fork works in the same manner as **execute** does with the user-created programs except the programs would be forked in the server. The server then sends a message over the socket to indicate completion.

A sample run using these commands follows:

```
<lila> addhost(aaron);
<lila> set(g1,random_graph(5));
<lila> fork("drawgraph1",berra,g1);
<lila> setdisplay(berra,"ferrari:0");
<lila> fork("drawgraph1",berra,g1);
```

Running XMaple Commands

Another program which also needed to be run remotely was XMaple. This was implemented in a different method from the **execute** command as it does not run through a server. This is because XMaple only needs to be run through the local host (at least at this point). The syntax of the command which performs this is:

xmaple(command).

This command is implemented by modifying the **.mapleinit** file before calling XMaple. This was necessary because there was no other way to tell XMaple to execute a command automatically. GraphPack puts the command into the **.mapleinit** file and then deletes that file after 10 seconds. It is deleted so that the next time the user enters Maple, it does not execute that command.

It is to be noted that the **.mapleinit** file is destroyed by this command.

Also, XMaple is forked so that GraphPack does not have to wait for the user to exit and may keep the result displayed.

Implementing Functions using the MasPar

Another major goal of this project was determining how to design and implement functions using the MasPar. The MasPar has the advantage of 2048 processors which allow certain SIMD algorithms to run much faster than sequential algorithms. Because a graph can be treated as a matrix and the MasPar's processors are set up as an array, we decided to add some of the power of the MasPar to GraphPack.

Before describing which algorithms were added, the actual method in which the communication between GraphPack and the MasPar was handled must be discussed. Because GraphPack would be running on a different machine than the MasPar, it became neccesary to somehow network the two, similar to what was done with the **execute** command.

3.2 The MasPar Operations

The current list of functions which may be performed through the MasPar include: transitive closure, all-pairs shortest paths, all bridges, root-based spanning tree, minimum spanning tree and a heuristic for the traveling salesman problem.

The algorithms were chosen because they can be implemented on a SIMD machine with excellent results. Some of the functions are simple enough that the overhead involved with the MasPar and graphs may not be worth it. However, these functions may be the building blocks for current and future MasPar functions.

For the most part, the functions use the **router** command on the MasPar which is not as fast as the **xnet** command. However, the **xnet** command can only be used if each processor needs to look at another processor the exact same distance up, down, left, right or diagonally. Unfortunately, this did not prove feasible for these algorithms.

The algorithms added will now be described. They may be found in the file **MasParfuncs.m** in the GraphPack directory.

- **Transitive closure** This is performed by the:

  m_close(graph);

  function. It takes the graph and finds the transitive closure of it. The graph returned includes all edges included by the closure. The algorithm used simply checks if an edge (a,b) and (b,c) exists and if so the edge (a,c) should also exist. It is able to do this in parallel, however, which reduces the time from $O(n^2)$ to O(n). These running times are well known in the literature [6].

- **All-pairs Shortest Paths** The function to do this is:

  m_shortp(graph).

  This is similar to the transitive closure function and checks if the distance of a path is less than the current distance. If so, it would change the distance to the smaller distance. This is also O(n) instead of $O(n^2)$.

- **Minimum Spanning Tree** The function form is:

  m_minspan(graph).

  Using Kruskal's Algorithm, the edges are sorted and then selected one by one until a spanning tree is found. In the worst case, this may still be O(m) where m is the number of edges.

- **Root-Based Minimum Spanning Tree** The syntax of this function is:

  m_spanning(graph).

  This function will include each edge in the shortest paths from the root to each vertex. The algorithm used first finds the shortest paths for the graph. Using the first vertex it finds any paths which are in the shortest paths from the root. This algorithm is O(n) on the MasPar.

- **Bridges** The syntax for this command is:

  m_bridge(graph).

  This algorithm is the most complicated of those discussed so far. The code works as follows with A being the initial graph:

  S=spanning(A);
  B=and(A, complement(transpose(S)));
  T=close(S);
  B=close(B);
  Any rows that match between T&B are vertices that are bridges. (i.e. row 1 = vertex 1)

  This algorithm is O(n).

- **Traveling Salesman Heuristic** The syntax of this command is:

  m_tsales(graph).

  The current heuristic works by ranking all the edges and finding the smallest edge from the current vertex. Once an edge is found, the next current vertex is the vertex the edge is connected to. The final edge must be between the first vertex and the final vertex found even if that edge does not exist in the graph. This is a greedy heuristic and should be greatly improved in the future. The running time is approximately O(n).

3.3 Additional Interpreter Commands

In this section, we just highlight some of the interpreter commands. LiLa is the programming language GraphPack programs are written in. It contains support for graphs (undirected and directed), sets, lists, queues and stacks. LiLa supports recursive function calls and has a configurable function library. Since graph are a major part of GraphPack, there are a number of improvements associated with them. A graph (directed or undirected) has the following basic structure:

```
Structure GRAPH: {
set: vertices;
set: edges
}
```

The assignment operator can be used to set the edge or vertex set of a graph in the same way identifiers are assigned. Since the edges and vertices of a graph are both sets, the set operations discussed above work on them also. Unlike normal sets, the + and - operators can be used to add or delete single items from either set.

Edges and vertices are referenced by name. This name can be an integer, double or string value. They can also have a weight. When adding or deleting edges and vertices, this weight is an optional parameter which can be specified along with the edge or vertex name. If a weight is specified when deleting edges or vertices, the edge or vertex will be deleted if and only if the weight specified matches the weight of the actual edge or vertex in the graph.

Adding Vertices: Duplicate vertices will not be added and no error will be flagged. The format of the assignment statement to add or delete vertices is:

```
<lila> vertices(g)=vertices(g) (+/-) VTX [(+/-) VTX...];
<lila>
- VTX can be one of the following:
- an integer.
- a double.
- a string in quotes (").
- one of the above in curly brackets followed by a weight
for the edge.   {name,weight}

Examples:
<lila> vertices(g)=vertices(g) - 6 - 7 + 9 + "v1" + 8.7;
<lila>
removes vertices 6 and 7, and adds 9,  v1, 8.7 to the
 vertices of graph g.
```

```
<lila> vertices(g)=vertices(g) + {6,7}-{"v2",6};
<lila>
adds vertex 6 with a weight of 7, deletes vertex 'v2' iff
vertex v2 exists with a weight of 6.
<lila> vertices(g)=vertices(g)+x+{y,z};
<lila>
X must be an integer, double, string or vertex variable.
Y must be an integer, double or string variable.
Z must be an integer, double or vertex.  Z will be used
for the weight of the vertex.
```

Adding Edges: Adding edges to a graph is similar to adding vertices. The format is:

```
<lila> edges(g)=edges(g) +/- EDG [+/- EDG..];
<lila>
EDG can be:
- an edge variable, as created by make_direcge()
- a description of an edge with two vertices and an optional
weight enclosed in brackets:
{Vertex1, Vertex2 [, Weight] }
```

Vertex1 and Vertex2 must be of type integer, double, string or vertex. An error will occur if either Vertex1 or Vertex2 does not exist in the original graph. Weight is optional and must be an integer, double or string. Duplicate edges can be added. If a vertex is deleted, it is up to the programmer to make sure that the corresponding edges are also deleted. Examples:

```
<lila> print(g);
VERTICES : {1, 2, 3, 4, 5}
EDGES : {{3, 4}, {3, 5}}
<lila> edges(g)=edges(g)+{2,3}+{1,5,65};
Adds edges 2,3 and 1,5 with a weight of 65 to g.
<lila> x=2;
<lila> y=5;
<lila> edges(g)=edges(g)+{x,y,"weight"};
x and y could also be doubles or strings.  A string value
for the weight is
also used here.
<lila> x=4;
<lila> y=5;
<lila> newedge=make_undiredge(x,y);
<lila> edges(g)=edges(g)+newedge;
<lila>
```

```
An edge can also be created using the make_undiredge() or
make_diredge() functions. If a graph has no edges, undirected
edges are the default type which will be added.  To add
directed edges, first create a directed edge using
make_diredge() and then add it to edges of the graph.
```

3.4 Additional Graphical Commands for XGraphPack

We want to have a 3-D embedding of the graph G. The idea here is to spread out the vertices of a graph onto the surface of a sphere in order to be able to view the graph as a three-dimensional object. There is no exact solution to the general problem of spreading points on a sphere such that the great circle degree measure between any two adjacent points is identical. Thus, an approximation must be settled for.

1. *loadtable:* The current algorithm first builds a table showing how many circles of latitude are needed to accommodate the number of vertices such that the latitudinal difference between the circles is the same as the longitudinal difference between the points.
2. *determine_number_of_circles:* After the table is built, a graph is read into an array and the number of vertices is used to look up the proper number of circles of latitude to spread the vertices out. These circles will lie at even increments of latitude.
3. *generate_vectors:* We now cycle through the circles of latitude, populating them with graph vertices using the same longitudinal spacing for each vertex. As we cycle through, the altitude and azimuth of each vertex is generated, which is then converted to normalized vector coordinates. Normalization of the vectors causes all points to lie one unit from the origin thereby forming a sphere.
4. Once we know the vector coordinates of each vertex, we may allow the user to rotate and view the graph using standard rotation and perspective transformations.

The problem with this method is that when the number of latitude lines increases by one, the number of vertices needed to fill the new latitude line is more than one. Thus, for graphs with many vertices, it will often be that the circle of latitude closest to the north pole will not be completely filled with vertices.

4 Conclusion

Our major objective is to make *GraphPack* a useful research and educational tool possible for theoretical computer scientists and mathematicians working in Combinatorics and in Graph Theory. We envision a user who needs to check a conjecture quickly, or to find a small but not trivial subsystem of a system of a specified nature. Similar examples can be drawn from quite different areas: complexity theory, graph theory, matrix theory, etc. We believe that GraphPack will be very helpful to researchers by paying careful attention to the interactive mode of computation. There were several design decisions associated with the proposed project. The major ones were as follows:

1. Fancier display of graphs, including a 3-D display and their rotations on all the axis, displaying selected subgraphs and the browsing capability.
2. The determination as to what extent and for which algorithmic methods the programming in LiLa was feasible and efficient.
3. The *GraphPack* system realizes its greatest potential through the routines it uses to call other software packages. Further GraphPack routines could also be called from other software packages.
4. *GraphPack* is a useful tool for programs written for parallel computers.
5. LiLa allows the user to execute remote shell commands. We incorporated commands on the remote machine, including a Sequent Balance 2100 computer with 16 processors sharing a common memory and the results are displayed on some other workstations.

All these improvements have been already implemented in GraphPack and we find the system to be more useful.

References

[1] Berque, D. et al. The SetPlayer System: An Overview and a User Manual, Department of Computer Science, Technical Report 90-20, Rensselaer Polytechnic Institute, Troy, N.Y., July 1991.

[2] N. Dean, NetPad: User's Manual (unpublished), 1990.

[3] Jarvis, J.P., D. R. Shier, and D. E. Whited, NETSOLVE: An Interactive Software Package for Network Manipulation and Optimization, DISTINCT Managment Consultants.

[4] Kocay, W., "Gropus & Graphs, A MacIntosh Application for Graph Theory," J. Combina. Math. Combin. Comput. , 3 (1988), 195-206.

[5] Krishnamoorthy, M.S., Spencer, T., and McCaughrin, M., GraphPack: A software system to manipulate graphs and directed graphs, Department of Computer Science, Technical Report 90-7, Rensselaer Polytechnic Institute, Troy, N.Y., 1990.

[6] Leighton, F.T., Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes,Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[7] NewBery, F., "EDGE: An Extendible Directed Graph Editor," University of Karlsruhe, W. Germany, 1988.

[8] Robins, G., "The ISI Grapher Manual," Information Sciences Institute, Technical Report, Merina Del Ray, CA, 1988.

[9] Shannon, G. et al. Graphview: An Extensible Interactive Platform for Manipulating and Displaying Graphs, Computer Science Department, Indiana University, Tech report no. 295, 1989.

[10] Skiena, S., Implementing Discrete Mathematics, Addison Wesley, Redwood City, CA., 1990.

[11] Zimmerman, A., "Graph Management Program," Michigan State University, Tech. Report, East Lansing, MI, 1987.

M. Krishnamoorthy,
A. Suess,
M. Onghena,
F. Oxaal
Department of Computer Science,
Rensselaer Polytechnic Institute,
Troy, NY 12180

T. Spencer
Department of Computer Science.
University of Nebraska,
Omaha, NE 68182