GraphPack: Design and Features

M. S. Krishnamoorthy, F. Oxaal, U. Dogrusoz, D. Pape,
A. Robayo, R. Koyanagi, Y. Hsu, D. Hollinger and A. Hashmi
Department of Computer Science,
Rensselaer Polytechnic Institute,
Troy, NY 12180.

We present several elegant methods used in *GraphPack*, an educational package, for three dimensional embedding of graphs, as well as a two dimensional layout method, namely *barycentric embedding.* After that, a program for automatic recognition of graphs is introduced. In addition, we show how graphs in *GraphPack* format can be exported into *Xfig* format. Finally, we describe the client/server capabilities of *GraphPack* designed for providing communication with other tools and application programs written in C and C++. **Keywords:** *Graph theory, educational tool, visualization of graphs, embedding graphs, manipulating embeddings, recognition of graphs, client/server model.*

1 Introduction

GraphPack is an educational package [5] developed at Rensselaer that provides users with the capability to study and visualize graph theoretic algorithms. The graphical user interface is built on X windows. This application can manipulate both digraphs and graphs, and can draw them under different embeddings. *GraphPack* itself is composed of two components: the kernel and Xgraphpack. The kernel has an interpreter that accepts Lila (*Li*ttle *La*nguage) commands from the user and communicates with Xgraphpack, which is an X windows program that draws graphs and performs certain operations on them (i.e., by clicking on predefined buttons) such as browsing, grouping, fetching and saving. Figures 1 and 2 show examples of some of the capabilities of *GraphPack*. It would be interesting to contrast Figure 2, which is a truncated icosahedron, with Figure 14 on page 35 of [4], which is a truncated octahedron. Please refer to [3] for information on several other graph manipulation systems.

The next section is dedicated to the description of 3-d and 2-d layout algorithms. The common goal of the 3-d algorithms presented is to increase our ability to think about them and view them "three-dimensionally" by evenly spreading a given number of points (vertices) on the surface of a sphere. An even spreading is one in which the great circle distance between any two neighboring points is the same. Such even spherical spreadings are geometrically impossible in most cases. So, the algorithms must find "good" approximate solutions as rather than exact ones. Since the algorithms are motivated by

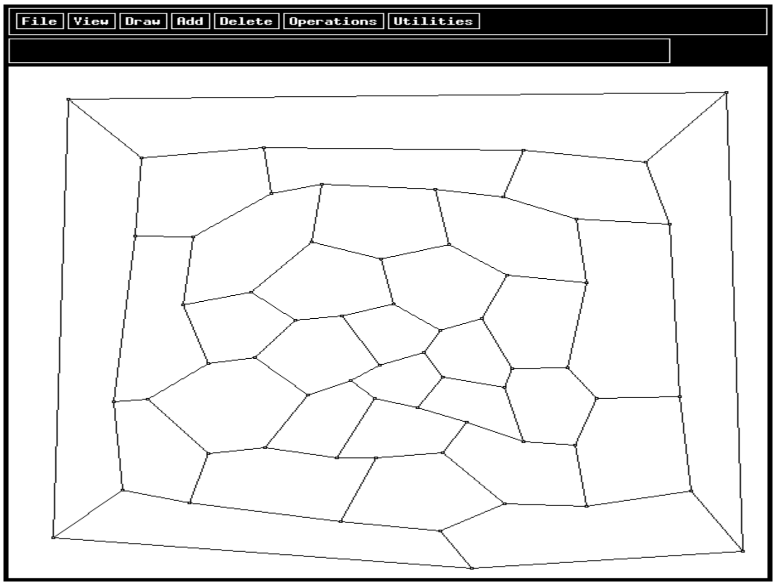

Figure 1: *A graph of a soccer ball in 2-d planar layout.*

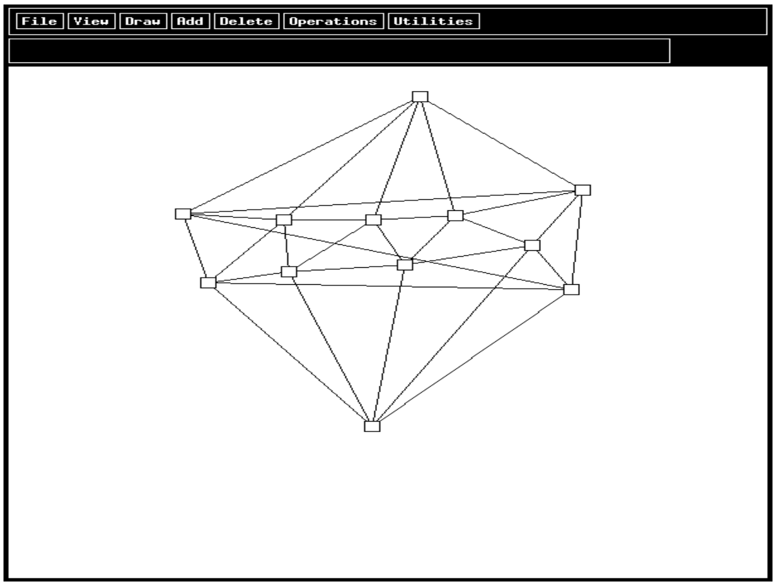

Figure 2: *Groupings of all pentagons of the soccer ball in Figure 1 resulting in an icosahedron.*

graph visualization, their implementation is within a special graphical viewing environment, namely *GraphPack*. This environment allows the user to rotate the points around each of three coordinate axes, and to permute graph vertices so that more visually palpable configurations could be obtained. The 2-d layout algorithm, on the other hand, utilizes barycentric embedding of Tutte[9]. An outer cycle of a specified length, which can be chosen by the user, is used in this algorithm.

In Section 3, we present a method to identify the vertices and edges of an undirected unlabeled graph from the physically scanned images of a printed graph. The printed graph is obtained from Xfig[8] (with its interface to *GraphPack*). The resultant graph along with the respective positions of its vertices are stored in *GraphPack* format. The reverse of this operation can also be performed when the user wants to export a graph from *GraphPack* into Xfig format.

In addition, *GraphPack* contains a feature that provides data communication between itself and other related applications such as Maple[2], Mathematica [7], Matlab[6], and Mosaic[1]. From a computer network approach, this feature allows *GraphPack* to be a client when it invokes tools such as Maple, Mathematica, and Matlab as servers. In other words, *GraphPack* is able to send requests to these applications which in turn service the request and return a result. On the other hand, *GraphPack* acts as a server when a C/C++ program or applications such as Mosaic activate and use it. The aim of a server in this sense is three-fold:

1. To make *GraphPack* act as a server. This will grant users flexibility when interfacing with *GraphPack* by writing a multimedia animation program for coloring as an example. This is accomplished by the ability to send commands to *GraphPack* without setting up a communication mechanism and with a minimal knowledge of LiLa.

2. To write multimedia graph programs with ease of of portability.

3. To provide a drawing facility within *Mosaic* framework.

This feature of *GraphPack* is described in Section 4, which is followed by a final section of concluding remarks.

2 Layout

We will first describe the three different 3-d layout methods followed by a description of a 2-d layout method, namely barycentric method that *GraphPack* uses.

2.1 3-d Layout Methods

The differences in the presented methods arise from the way they spread points of the graph on a sphere as follows:

- populating evenly spaced lines of latitude,
- incremental point repulsion,
- subdivision of octahedron and random subtraction.

Lines of Latitude Population

In this algorithm, first a table showing how many circles of latitude are needed to accommodate n vertices such that the latitudinal difference between the circles is the same as the longitudinal difference between the points is built.

After that, 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 onto. These circles will lie at even increments of latitude.

We then cycle through the circles of latitude, populating them with graph vertices using the same longitudinal spacing for each vertex. As we do this, 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.

The table is built as follows:

1. Divide a sphere into progressively larger numbers of equally spaced circles of latitude.
2. At each new number, add up the total circumference of all the circles.
3. Now, there will be a maximum of the numbers of vertices which can be spread on all the circles such that the longitudinal spacing of the points is the same as the latitudinal spacing between the circles (simply divide the total circumference by the latitudinal angle between the circles).
4. Enter that maximum number into the table.

Now, the number of lines of latitude needed is simply a table look up based on the number of vertices in the given graph.

A vector may now be generated for each vertex based on the number of circles of latitude which when divided into 180 degrees gives us the latitudinal angle between them – which in turn is used as the value to longitudinally space the vertices along each circle.

The weakness of this method is that when the number of latitude lines increases by 1, the number of vertices needed to fill the new latitude line is more than 1. Thus, for graphs with many vertices, it will often be the case that the circle of latitude closest to the north pole will not be completely filled with vertices.

Figure 3 shows an example of this method for a cube graph of eight vertices.

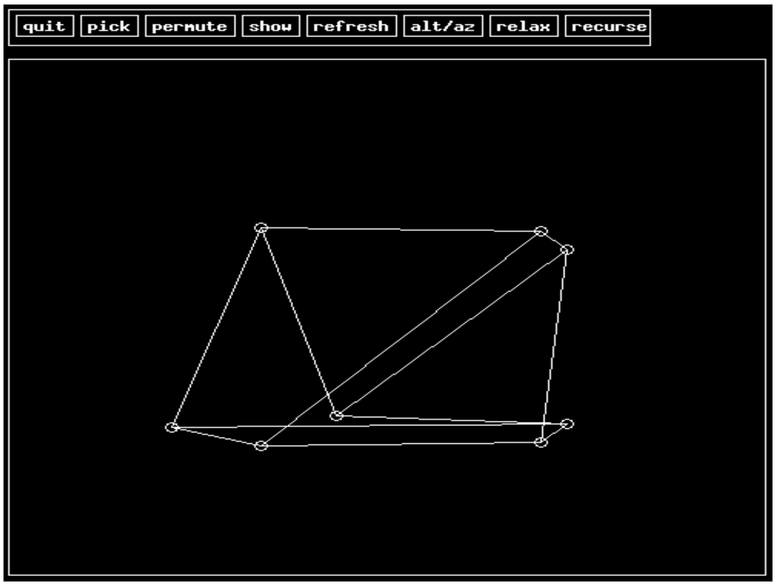

Figure 3: Lines of latitude population method *for a cube.*

Point Repulsion

In the *point repulsion method*, on the other hand, points are randomly distributed on a sphere and then caused to repel each other in small increments until the distance between each set of neighbors is above a certain threshold. This threshold is arrived at through experimentation. The maximum value it can take is the cap angle of the cap which has surface area equal to the surface area of the sphere divided by the number of points to be spread out. This number will be a little too big in most cases, and if used, will cause the procedure to loop forever. Thus, an experimentally derived smaller number is used. Since this procedure is not guaranteed to terminate, the program will automatically exit if a specified maximum number of times through the loop is reached. The algorithm on average takes $\Theta(n^2)$ steps to compute. An example of *point repulsion method* is given in Figure 4.

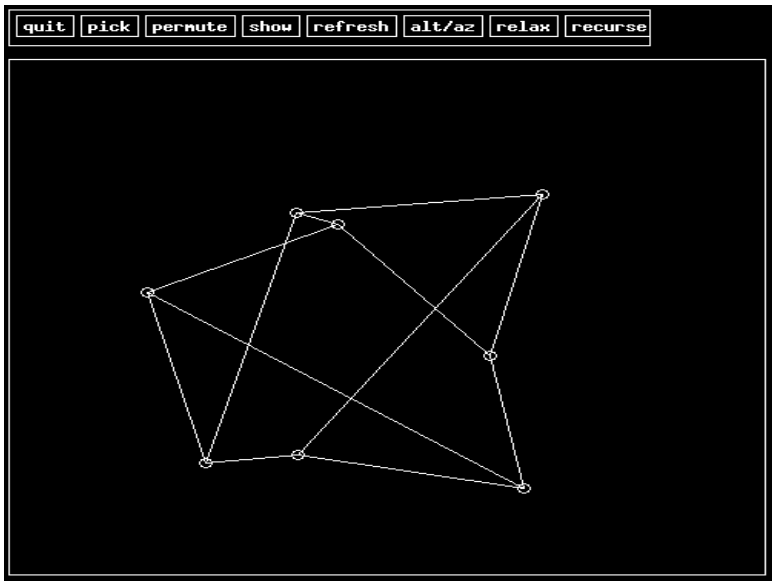

Figure 4: Point repulsion *on a cube.*

Subdivision

An octahedron has 6 vertices, and 8 equilateral triangular faces. If each face is replaced by four faces by dividing each edge in half with a new point and connecting these points with new edges, then 8 triangles become 32 triangles and 6 vertices become 18 vertices. In general, each subdivision produces 4 times the number of triangles previously existed and the number of vertices is related to the number of triangles by the following formula: $number\_of\_vertices = (3 \cdot number\_of\_triangles - 24)/6 + 6$ (for a tetrahedron the formula would be $(3 \cdot number\_of\_triangles - 12)/6 + 4$.) Here, the desired number of vertices is obtained by first subdividing an octahedron until the needed number is exceeded, and then subtracting vertices at even intervals until the needed number is reached. Thus, for example, if 112 vertices are needed, 258 vertices would have to be generated first, and then 146 of them subtracted by the table below.

Level of Recursion	*Number of Triangles*	*Number of Vertices*
1	8	6
2	32	18
3	128	66
4	512	258
5	2048	1026
6	8192	4098

Clearly, the evenness of the octahedron distribution is compromised since many subtractions are needed in most cases.

In order to implement this, an existing utility, which generates the three vertices of each of the triangles produced by the subdivision described above, was located. Thus, for example, at recursion level two, this utility produces the (normalized) coordinates of $3 \cdot 32$ or 96 points. Many of these points are duplicates. To obtain the 18 unique points, the output must be sorted and filtered using the Unix **sort** and **uniq** utilities. An example of this method can be found in Figure 5.

Observations

"How can complex graphs be viewed intelligibly on two-dimensional surfaces?" Some efforts have focused on displaying such graphs as projections of three-dimensional wireframe objects. Such efforts fail when too many graph edges cross through the volume defined by the wireframe object. Such an object when projected produces an unintelligible jumble of criss-crossing lines on the two-dimensional viewing surface.

Here are four possible ways of improvement:

1. Allow the user to interactively rotate the object in three dimensions, thereby adding more "visual bandwidth" to the two-dimensional viewing medium.

2. Use stereoscopic technology to clarify which edges are behind other edges whose projections intersect.

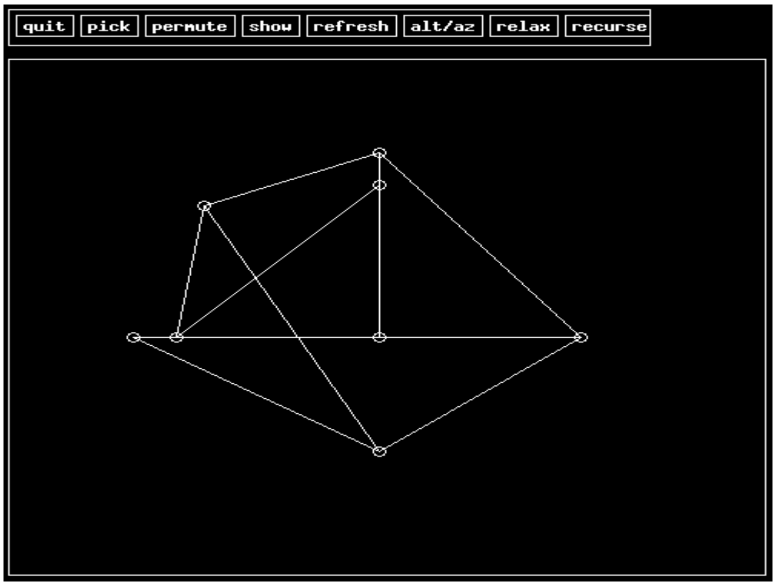

Figure 5: Subdivision method *for a cube.*

3. Intelligently spread the vertices of the graph and reorder the edges, so that two-dimensional projections are optimized for intelligibility.

4. Use some visual representation of the graph other than wireframe to either replace the wireframe approach, or supplement it.

The effort here implements options 1 and 3 without the “intelligence.” Motion, when added as a dimension to visual communication, adds “visual bandwidth.” Objects which are ambiguous when viewed statically may be resolved when the information which motion provides is added. However, for wireframe objects, sometimes even motion is not enough. Options 2 and 4 would address this problem.

2.2 A 2-d Layout Method: Barycentric Method

The basic idea of the barycentric method, devised by Tutte [9], is that the location of each vertex should be the average of its neighbors' locations. A set of vertices are given fixed positions, which are then used to compute the positions of the remaining vertices. This is done by solving the systems linear equations $Mx = x'$ and $My = y'$. M is a $k \times k$ matrix, where k is the number of vertices that were not initially positioned. It is based on the adjacency matrix of the graph: $M_{i,j} = -1$ if vertices i and j are adjacent, 0 otherwise. The vectors x' and y' are computed from the vertices that were initially positioned; ${x'}_i$ is equal to the sum of the x coordinates of the vertices adjacent to unpositioned vertex i and y' uses the sums of the y coordinates. Solving the two systems of equations will give the locations of the unpositioned vertices.

The vertices whose locations are fixed initially can be selected in three ways. By default, the program will find any arbitrary cycle of the graph and use those vertices, placing them in a circle. Another way is to let the user specify a required length for the cycle. If no cycle of that length exists, the drawing routine returns failure. Thirdly, the user may select specific vertices and place them where desired, and the locations of the remaining vertices will be computed from them.

Figures 6, 7, and 8 show examples of barycentric method with different cycle lengths.

3 Undirected Graph Recognizer

Another feature of *GraphPack* is its ability to extract the logical meaning of an undirected graph given the bit-mapped image of the graph. In other words,

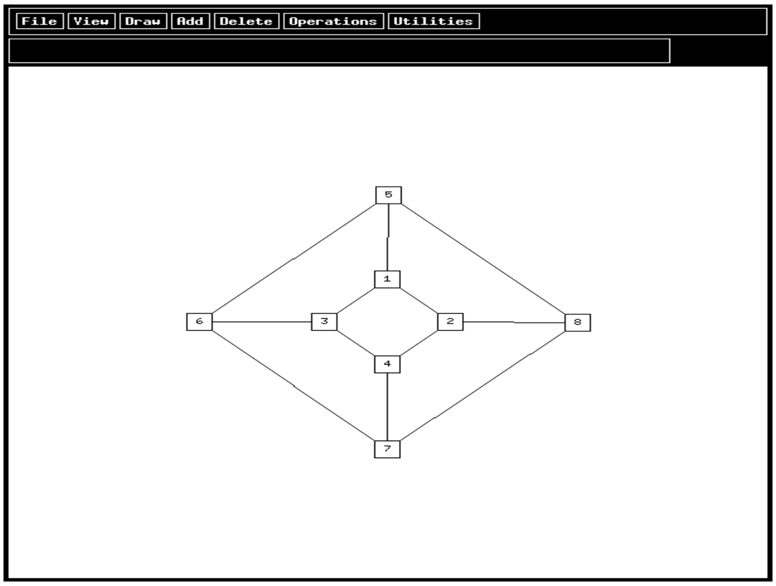

Figure 6: Barycentric *method with an outercyle of length 4 for a cube.*

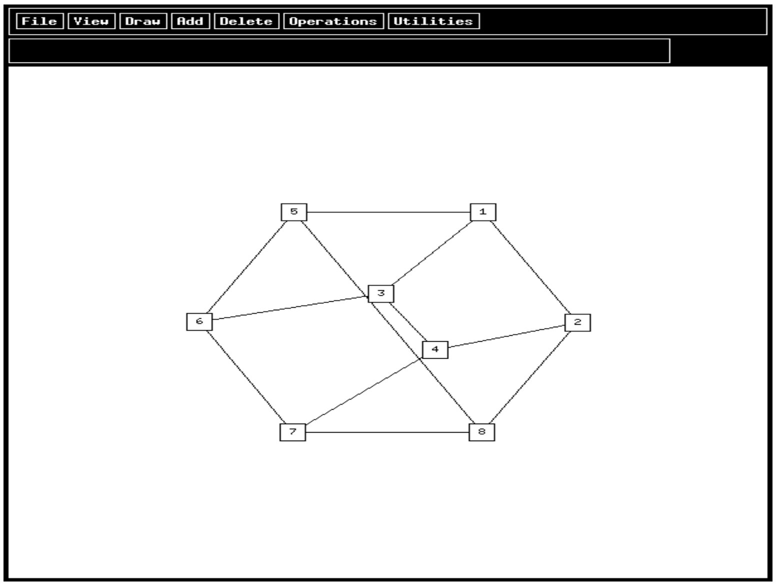

Figure 7: Barycentric *method with an outercyle of length 6 for a cube.*

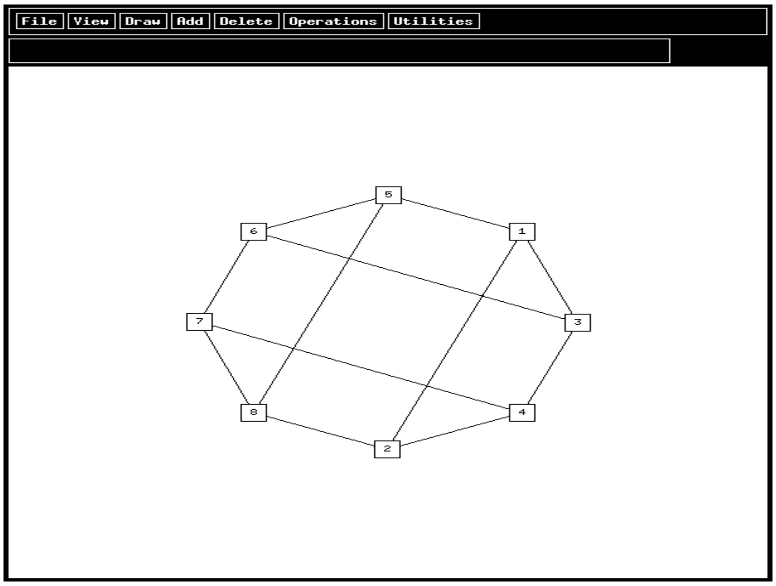

Figure 8: Barycentric *method with an outercyle of length 8 for a cube.*

given a picture of a graph, it can generate a list of vertices, edges, and a connection table.

The program performs better when the graph is more distinctly drawn, with filled circles for vertices and straight lines for edges. The image of the graph is expected to be solid black on white (no grayscale or dithering allowed).

The program first localizes the vertices by finding scan lines of consecutive dark pixels (i.e. corresponding to the areas of the circle near the diameter), and then seeks edges between pairs of vertices. The slopes of the edges adjacent to each vertex are calculated, and the next nearest vertex in that direction is found yielding an edge between this vertex and the original one. This calculation is done for each vertex in order to find the complete adjacency list.

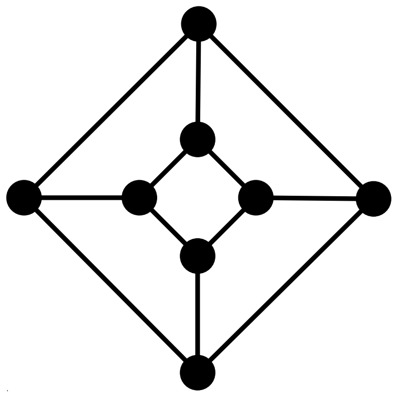

Figure 9: *Picture of an arbitrary graph (a cube).*

For example, from the raster image of the graph in Figure 9, the recognizer produced the following output in *GraphPack* format.

```
VERTICES:{1,2,3,4,5,6,7,8}
VWEIGHTS:*1,1,1,1,1,1,1,1*
EDGES:{{1,5},{1,4},{1,2},{2,6},{2,3},{3,7},{3,4},{4,8},{5,8},{5,6},{6,7},{7,8}}
EWEIGHT:*1,1,1,1,1,1,1,1,1,1,1,1*
```

The following are the vertex pixel co-ordinates corresponding to this graph (note that this layout corresponds to the graph in Figure 6 in *GraphPack*, which is the same as the original graph).

```
(25,270)
(166,269)
(235,337)
(236,481)
```

```
(237,56)
(237,198)
(306,269)
(447,270)
```

The reverse operation, obtaining the graph in Xfig format, can be performed by calling the `draw_info()` function with a graph name in *GraphPack*.

While *GraphPack* is a powerful system for graph drawing and manipulation, there are numerous useful operations, many of which can be considered to be beyond its scope, that it cannot perform. However, these operations can be easily done on several existing tools such as Maple, Mathematica, Matlab, and Mosaic with which *GraphPack* is able to communicate using a client/server model[5]. In the next section, we will describe the implementation of this model in which *GraphPack* acts as a server.

4 *GraphPack* as a Server

4.1 Application Programs

The important aspect of implementing *GraphPack* as a server is the data communication between a user's program and itself. Sockets, which handle data communication efficiently, are used for this purpose. This communication is illustrated below. Consequently, the project is split into three files: `client.c`, `server.c`, and `user.c`. The server sets up the socket connection for the kernel and executes *Loopback* which is an existing application to execute both kernel and Xgraphpack. The client contains functions to execute the server, set up the socket connection for the user's program, send Lila commands to the kernel, and kill the server and *GraphPack* when finished.

Here is the pseudo-code for the server and the client mentioned, in an outline form.

• **Server**:

    set up the address for the socket;
    open and bind the socket;
    listen for the client;
    if client does not connect within TIMEOUT seconds
        exit();
    accept the socket;
    fork();
    Child process:
        redirect I/O to the socket;
        execl(Loopback);

Parent process:
waitpid(for child process);

- **Client**:
gethostname();
fork();
Child process:
execl(executable file of server.c);
Parent process:
set process group to be the leader of all subsequent *Graphpack* processes;
sleep(3) to allow the child process time to open the socket;
set up the socket address;
connect with the socket;
switch input to init_server:
case 0: check if the graph input file exists and then call display();
case 1: nothing;
case 2: call make_graph();
case 3: call input_graph();
Functions:
send_cmd():
display():
exit_server():
error():
cont():
make_graph():
input_graph():

This client/server feature strengthens *GraphPack*'s ability to provide useful services for a variety of users. In addition, with the aid of animation and sound (sound annotations are done through the sound facility provided by the Sun systems using C interface) for visualizing the computation, *GraphPack* is even a more useful tool in the study of graph theory.

4.2 Interface to Mosaic

We created an experimental *GraphPack* server by interfacing Xmosaic to *GraphPack* through an HTTP server running a cgi-bin script written in PERL. Upon entry to the interface document, an Xmosaic user is presented a form to fill out including information about the X window display name currently in use.

Once the user completes the form, the resulting URL request is sent to our HTTP server, which then starts up a *GraphPack* session and sends the window to the X window display indicated. This experimental service provides full access to all of the features of *GraphPack*, but usage is limited because of the security issues involved.

5 Concluding Remarks

In this chapter, we presented drawing and recognition capabilities of an educational package, *GraphPack*. We also discussed how it can be used as part of a client/server model from and within C or C++ programs and other applications such as Maple, Mathematica, Matlab, and Mosaic.

GraphPack is ideally suited for creation, drawing, and manipulation of graphs with not more than thirty vertices. It does not seem to be as efficient and well suited for graphs with seventy or more vertices though. Currently, we are exploring to overcome this limitation. Another improvement we plan to make on this package is the flexibility of having vertices of different sizes and shapes in the same graph.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant Number CDA-8805910.

References

1. T. Berners-Lee, R. Cailliu, and A. Secert, The World-Wide Web Initiative, INET 93 Proceedings, 1993.
2. B.W. Char et al. , Maple V Language Reference Manual, Springer Verlag, 1991.
3. N. Dean and G. Shannon (eds), Computational Support for Discrete Mathematics, Vol. 15, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS Publications, 1994.
4. D. E. Knuth, The Stanford Graph Base, Addison Wesley, MA 1993.
5. M. Krishnamoorthy et al., "Improvements to GraphPack: A System to Manipulate Graphs and Digraphs", in Computational Support for Discrete Mathematics and Theoretical Computer Science, Vol. 15, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS Publications, 1994, pp. 279-296.

6. MATLAB - High-Performance Numeric Computation and Visualization Software (Version 4, User's Guide) The MathWorks Inc. Prentice-Hall, Englewood Cliffs, 1995.
7. S. Skiena, Implementing Discrete Mathematics, Addison Wesley, Redwood City, CA., 1990.
8. S. Sutanthavibul, "XFIG: Facility for Interactive Generation of figures under X11 - Users Manual", University of Texas at Austin, Texas, 1985.
9. W. T. Tutte, "How to Draw a Graph", *"Proc. London Math Soc.* Vol. 3, No. 13, 1963.