

GDR: A VISUALIZATION TOOL FOR GRAPH ALGORITHMS *

MATTHIAS STALLMANN , RANCE CLEAVELAND AND PRASHANT HEBBAR

1. Introduction. This report describes *GDR* (for *Graph Drawing*), a tool for editing graphs and animating graph algorithms. The motivation for animation tools comes primarily from the classroom; students often have difficulty mastering the formal concepts of graph theory, even though they usually have no trouble in following the associated visual intuitions. An appropriate animation tool can provide an invaluable link between formalism and graphical intuition, since students are given the opportunity to see how a formal presentation of a graph algorithm directly translates into visually intuitive operations on graphs. For this reason, animation systems can also play a very useful research role as testbeds for prototyping new algorithms and for testing conjectures about graphs.

The design of GDR was strongly influenced by our desire for the tool to be easy-to-use (even by relatively inexperienced programmers), portable, and flexible. Our approach was to develop GDR as a *tool* rather than a self-contained system, so that it is easily modified and simple to interface with other software. This has led to two approaches to using GDR in conjunction with other tools. In the first, GDR functions as a graph editor; users create graphs using the tool, save them in files, and feed the files as input into the other tools. This mode of interaction only requires that front-ends be written for the other tools that can parse the (very simple) output generated by GDR. The second mode of interaction is *object-oriented*; GDR provides a high-level interface to *graph objects* that programmers can write programs to manipulate (using calls to functions implemented in GDR). Users of GDR can then create graphs and apply the programmer-supplied routines to these graphs. Moreover, GDR is written in C and uses the library routines supplied by X-windows to implement its graphical capabilities. Thus the tool can be run on a number of different platforms.

The purpose of this report is twofold: to describe the current implementation of GDR and some of its uses and to outline ideas for future versions of the tool. Accordingly, Section 2 gives an overview of the GDR design and GDR features, while Section 3 describes the interface for the creation and editing of graphs. Section 4 presents the abstract data type through which programs have access to graphs created by GDR. Section 5 gives a detailed account of the use of GDR for algorithm animation. In Section 6 extensions and future plans for GDR are discussed. There are also 2 appendices: Appendix A contains user documentation for the current implementation and Appendix B gives additional detailed examples of algorithm animations implemented using GDR.

In what follows we distinguish between the *programmer*, a person who writes an application program (e.g. an algorithm animation) using GDR as a tool, and the *user*,

* please direct all correspondence to: Matthias Stallmann, Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206; phone: (919) 515-7978; e-mail: mfms@euler.csc.ncsu.edu

a person who interacts with GDR, either to edit a graph or to run an animation created by a programmer.

Related Work. In the remainder of this section we compare and contrast GDR with other animation packages. The most sophisticated of these is the *BALSA* system [Bro87], which provides a very flexible tool for developing detailed graph animations. The system is much more advanced than tools like GDR, but a drawback is that it is difficult to use; developing animations requires a knowledgeable technical staff. With GDR, by comparison, it is possible for an individual teacher/researcher, and even advanced students, to generate the code necessary for producing a simple animation within a few hours.

The features that distinguish GDR from other, simpler animation tools include its object-oriented design, its portability, and its fully interactive nature. In contrast to the system proposed by Bentley and Kernighan [BK87], GDR provides high-level primitives for manipulating the basic objects (graphs, edges and labels) in graphs; the user is insulated from low-level details involving specific data structures and graphical operations. Two other animation packages—*GMB*, developed by Jablonowski and Guarna [JG89], and *GraphView*, developed by Birgisson and Shannon [BS89]—are integrated systems rather than stand-alone tools; that is, the user or programmer is required to adopt the underlying abstract data types, and to develop graph representations entirely on the system. In contrast, the design of GDR encourages integration with other unrelated tools. One side effect of this philosophy is that while the basic GDR tool does not yet offer all of the sophisticated features of these systems, the advantages of those features may be gleaned by either using GDR in conjunction with other tools or by developing custom enhancements to GDR for particular applications.

Moreover, the GMB system is not fully interactive. While there are fairly sophisticated features for interactively viewing graph data structures created under program control, the creation or modification of the graphs cannot be done on the fly. A disadvantage of GraphView is that it has been developed on the NeXT machine. Although implementations on other systems are planned, the overall design appears to make extensive use of the special characteristics of the NeXT environment. GDR, by way of contrast, uses standard user interface tools: the current version is written in C and uses X-Windows (only the standard X library, no widgets or toolkits).

2. Overview. The current implementation of GDR is a prototype for testing ideas that may be used in future implementations. Nonetheless it is quite usable even in its crude present form. Originally, GDR was intended as a simple tool to generate input for implementations of graph algorithms. The idea was that the user of GDR would draw vertices and edges on the screen, and GDR would produce a listing of the graph in adjacency list format, suitable for input to other programs. Such a tool requires editing features, and hence can also be used simply for creating and editing drawings of graphs. This mode of operation is discussed in detail in Section 3.

The goals of GDR have evolved substantially from its original purpose; the philosophy we have adopted is that any editing operation that can be invoked by the user

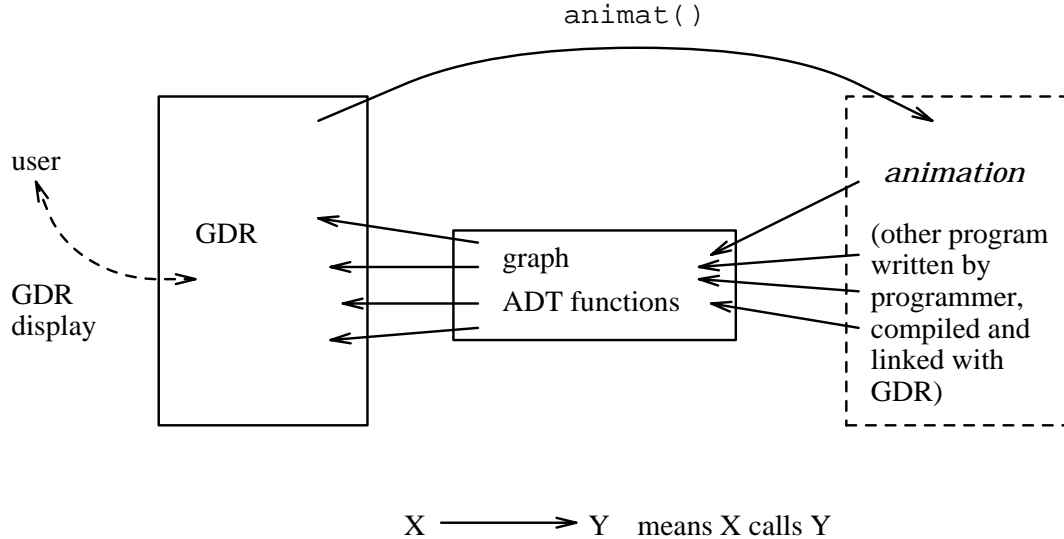


FIG. 1. *The GDR programming interface.*

with a mouse can also be activated by a program used in conjunction with GDR. This allows GDR to act as an algorithm animation tool for graph algorithms or as a tool for debugging graph algorithm implementations. As is apparent from the presentation in Section 5 and from the other examples in Appendix B, the animation features, though crude, are quite useful for both teaching and research. Their simplicity is one of the chief virtues of the animation features of GDR, as simple animations can be created in a matter of hours.

Ideally, GDR and the program that is to be animated would run as separate processes, communicating via some protocol supported by the operating system (perhaps even over a network). The difficulty of defining such a protocol and the technical problems inherent in developing a portable tool that interacts with external processes in non-trivial ways led us to take a simpler approach initially. The current implementation of GDR can only interact with a program that is specifically compiled and linked with GDR. This program is henceforth referred to as the *animation*. The interface is a programmer-defined C function `animat()` which in turn calls the GDR functions that interact with the display. Since `animat` can occur in a separately compiled module, the process of rebuilding GDR with a new program takes only a few seconds. The restriction that there be only a single function is not really limiting, since `animat` can create its own popup windows to interact with the user, and could, for example, supply a whole menu of different programs to run. Figure 1 illustrates the programming interface for GDR.

At the heart of the animation interface is a mechanism for accessing the graph itself as an abstract data type. The specific routines and conventions for this are discussed in detail in Section 4. Although the language used for implementing GDR is plain C without any object-oriented features, the design and mechanisms of GDR are motivated by an object-oriented philosophy. From GDR's point of view, a graph is a composite object whose components are vertices and edges. Each vertex and edge

has several attributes that can be manipulated by the user or by the animation. For clarity we classify the attributes into three categories: *logical attributes* are those that are completely independent of any display of the graph, such as the adjacency list of a vertex or the endpoints of an edge; *geometric attributes* are those that govern the location of objects in the display, such as the location of a vertex; *display attributes* are those that govern other aspects of the display, such as whether or not a vertex is highlighted. Table 1 gives a list of the attributes that are currently supported. Both internal and external representations of graphs record all attributes with each vertex and edge so both the logical and physical representation of a graph can easily be reconstructed. (Note: internal representation is what GDR stores while executing, external is what GDR writes to a file at the request of the user.)

Vertices and edges have labels which may contain arbitrary character strings. Labels are logical attributes since they are used for such things as the distance or cost of an edge. But the label of an edge has geometric attributes defining its position relative to the line segment connecting the two endpoints of the edge and a display attribute determining whether the label is *exposed* or *hidden*. The current implementation does not allow the user or program to change the location of a vertex label; however, vertex labels, like edge labels, may be exposed or hidden.

GDR allows graphs to have multiple edges between the same endpoints. Since multiple edges cannot all be represented by straight lines, each edge is displayed as 3 line segments (which are collinear when the edge is a straight line). The two intermediate points, referred to henceforth as *knots*, are included among the geometric attributes of an edge. Their position, like that of the edge label, is calculated relative to the straight segment connecting the two endpoints. When an edge is initially created its knots are spaced equally wrt the straight segment and are positioned far enough away from the straight segment to avoid collisions with any previous edges between the same two endpoints (current implementation does not check for collisions with edges having different endpoints). Knots can be moved any time under user or program control. Figure 2 shows some edges with knots, specifically the edges $(0, 0)$, $(0, 1)$, and $(2, 0)$. Note how the relative position of labels and knots remains unchanged when vertices (and their incident edges) are moved as illustrated in Figure 3.

Since one of our applications is the animation of finite-state machines, GDR also allows self-loops, which are represented as triangles. The self-loop, like other edges, has three segments and two knots. The only difference here is that two of the segments share the endpoint of the edge. For self-loops the positions of knots and labels cannot be defined relative to the straight segment between the two endpoints. Our current strategy, which is not entirely satisfactory, is to define knots using coordinates relative to the endpoint of the edge (and oriented the same way as the overall coordinate system — see Figures 2 and 3). When a self-loop is created, its knots must be defined by the user, as we have not yet come up with a reasonable heuristic for doing so.

3. Creation and Editing of Graphs. Interacting directly with the GDR user is the interface that allows creation and editing of graphs, either directed or undirected. The current interface was designed for simplicity and ease of use.

TABLE 1
Attributes of Vertices and Edges

Logical Attributes	
vertex <i>id</i> <i>adjacency list</i> <i>label</i>	unique nonnegative integer for each vertex a linked list of edges incident on this vertex character string (interpretation defined by programmer or user)
edge <i>head,tail</i> <i>label</i>	two vertices incident on the edge; choice of <i>head</i> versus <i>tail</i> is arbitrary for undirected graphs; a directed edge is directed from <i>tail</i> to <i>head</i> character string (interpretation defined by programmer or user)
Geometric Attributes	
vertex <i>position = (x, y)</i> <i>label position</i>	coordinates of the displayed vertex, relative to upper left corner of display coordinates of the upper left corner of the vertex label
edge <i>knot1/knot2 position</i> <i>label position</i>	coordinates of the 2 knots of the edge coordinates of the upper left corner of the edge label <i>note: knot and label positions are calculated relative to the straight-line segment between the endpoints of the edge</i>
Display Attributes	
vertex <i>highlighted</i>	true if vertex is highlighted (displayed with black id on white background), false if not (white id, black background)
edge <i>highlighted</i>	true if edge is highlighted (displayed with each line segment doubled), false if not
vertex/edge label <i>exposed/hidden</i>	exposed if label appears in the display, hidden if it does not appear

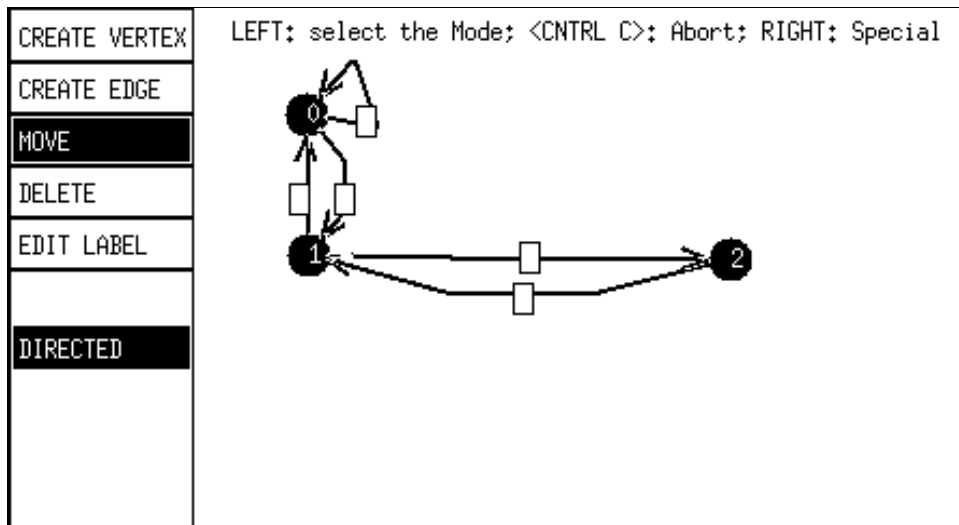


FIG. 2. *Edges with knots.*

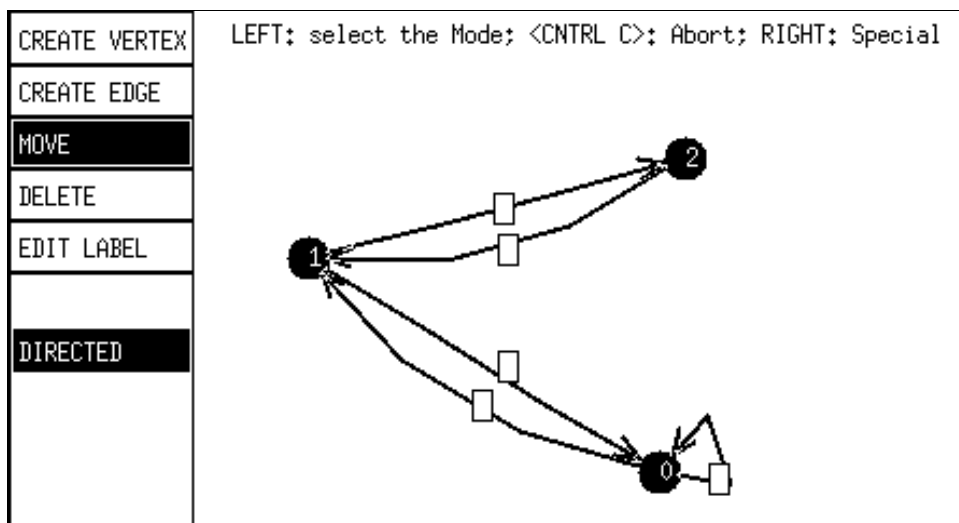


FIG. 3. *Knots after vertices are moved.*

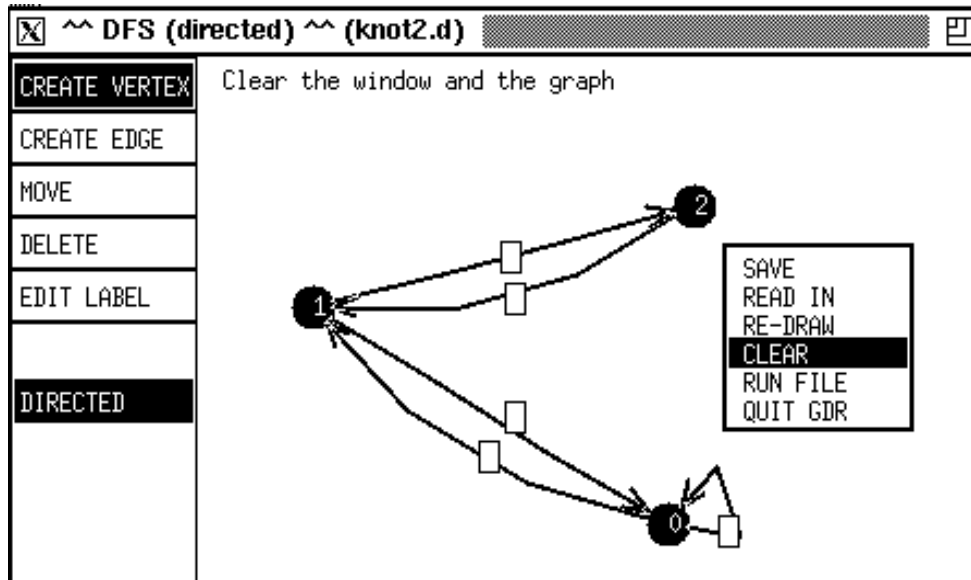


FIG. 4. *The GDR window.*

The GDR window, as shown by Figure 4, responds to user commands either directly or via a menu. On the left side of the window is the *panel*, where the user, by pointing to one of the entries and clicking left, can select the current editing mode. The mode determines GDR's response to mouse clicks or keystrokes in the *display*, the part of the window where the graph is drawn. For example, when in **CREATE VERTEX** mode, GDR puts a new vertex in the display at the position of each left click (if there already is a vertex at that position, the user is prompted to try again). A typical sequence used in the creation of new graphs is to create all vertices first, enter **CREATE EDGE** mode to create edges, then enter **EDIT LABEL** mode to put text in the labels of vertices and/or edges. **MOVE** and **DELETE** modes allow for the repositioning or deletion of objects already in the display. At the bottom of the panel is an indicator to let the user know whether the graph being created is directed or undirected. The status of the graph (directed or undirected) cannot be changed during an edit session, but is determined either by a command line parameter when GDR is called or by the animation.

The interpretation of mouse clicks and keystrokes for each of the edit modes is described in Table 2.

Another method of user interaction is via a pop-up menu (also shown in Figure 4), which is activated by depressing the right mouse button. The menu commands, described in Table 3, are executed immediately. Note that the more frequently used commands have keystroke synonyms (this feature was added to allow "canned" animations for classroom demonstrations to be stepped through without use of a mouse).

The top line of the display is used to show short reminders to the user about what is expected in the current mode of operation.

4. Access to Graphs. Access to graphs created by GDR may be either external or internal. External access is provided via the file created when a graph is saved.

TABLE 2
Panel Options in GDR

CREATE VERTEX	Left mouse click creates a new vertex (number is one greater than most recently created vertex, first vertex has number 0).
CREATE EDGE	Left mouse click on vertex i followed by left click on vertex j creates an edge from i to j . If $i = j$, two more clicks define knots of loop.
MOVE	Point to vertex, edge label, or knot, push middle mouse button and hold until mouse points to new position.
DELETE	Left mouse click on vertex or edge deletes the vertex or edge.
EDIT LABEL	Move mouse inside rectangle representing a label to change the label; any typed text becomes the new label ([Delete] acts as backspace). Left mouse click on vertex exposes the label or hides it if it's already exposed.

TABLE 3
Menu Options in GDR

Menu entry	Key synonym	Description
SAVE	s or S	Current graph is saved in a file (user is prompted for the file name and asked to confirm the action if the file already exists).
READ IN	r or R	A graph is read from a file (user is prompted for file name). Current graph is erased.
RE-DRAW	none	The window is redrawn (in case things get messed up).
CLEAR	none	The current graph is erased.
RUN FILE	p or P	The animation program is executed. Whenever a window with the prompt RESUME ? appears at the bottom right of the screen, control returns to the user interface: the user can edit the graph (e.g. move things around, modify labels, etc.). The program continues execution if there is a left mouse click in the RESUME ? window, or a QUIT GDR command issued.
QUIT GDR	q or Q	Exit from the GDR program, or resume execution of the animation program if it is running.

```

/* traverse adjacency list for vertex v (directed or undirected) */
e = first_out_edge(v);
while (e != NULL_EDGE) {
    w = other_vertex(v,e);
    /* process edge e and/or vertex w */
    e = next_out_edge(v,e);
}

/* an alternative C macro using a for loop: for_adjacent(v,e,w)
   traverses the adjacency list of vertex v, letting e and w denote each
   edge and other endpoint, respectively */
#define for_adjacent(v,e,w) for (e = first_out_edge(v);\
    e != NULL_EDGE ? (w = other_vertex(v,e), TRUE) : FALSE;\
    e = next_out_edge(v,e))

```

FIG. 5. Code for traversing adjacency lists.

The format of this file can easily be read by another program implementing a graph algorithm (it essentially provides an adjacency list for each vertex). There is also a procedure that prints only the logical attributes of the graph in an easy-to-use format (graphical information is omitted).

Internal access to the graph occurs when the animation (the program compiled and linked with GDR) is executed via the `RUN FILE` menu command. The paradigm for accessing the graph is *edge oriented* [Ebe87], meaning that each vertex has access to a list of its incident edges (and multiple edges and loops are allowed). Table 4 shows the functions that may be used to access the logical structure of the current graph, while Table 5 shows those that modify logical graph attributes. GDR supplies definitions of `vertex` and `edge` types. C program fragments for traversing the adjacency list of vertex v in an undirected and directed graph are shown in Figure 5. The constants `NULL_VERTEX` and `NULL_EDGE` are supplied by GDR to denote undefined values of type `vertex` and `edge`, respectively.

Our implementation of the underlying graph structure is similar to Ebert's, except that we use explicit linked lists rather than arrays, so that edges are identified by pointers rather than by integers. Vertices are still represented as integers in our current implementation, but we are likely to adopt a pointer-based representation in the future (a pointer-based representation is already used in `VTVIEW` [Tre92], a derivative of GDR used for designing concurrent programs). A feature that our implementation shares with Ebert's is the storage of incoming edges in the same list as outgoing edges. This allows the underlying undirected graph of a directed graph to be extracted easily. The list of outgoing (incoming) edges is traversed by skipping the incoming (outgoing) edges. Another advantage of this representation is the fact that an undirected edge is represented only once (with an arbitrarily chosen direction).

Future implementations of GDR will use `first_vertex` and `next_vertex` to access a linked list of all vertices, analogous to how this is done for edges (see Table 4).

TABLE 4
Access to Logical Graph Attributes in GDR

<code>max_vertex()</code>	returns the maximum vertex id possible (current implementation does not guarantee that it's valid)
<code>is_valid_vertex(vertex:v)</code>	returns TRUE if <code>v</code> is a valid vertex, FALSE otherwise (the vertex whose id is <code>v</code> may have been deleted)
<code>get_a_vertex()</code>	returns the first valid vertex in the graph
<code>are_edges_equal(edge:e1,e2)</code>	returns TRUE if the edges are the same, FALSE otherwise
<code>head(edge:e)</code>	returns the vertex to which <code>e</code> is directed (the second vertex to be added to <code>e</code> in case of an undirected edge)
<code>tail(edge:e)</code>	returns the vertex from which <code>e</code> is directed (the first vertex to be added to <code>e</code> in case of an undirected edge)
<code>other_vertex(vertex:v,edge:e)</code>	returns the vertex that along with <code>v</code> defines the edge <code>e</code>
<code>first_out_edge(vertex:v)</code>	returns the first outgoing edge from vertex <code>v</code> (first edge on <code>v</code> 's adjacency list if undirected graph)
<code>next_out_edge(vertex:v,edge:e)</code>	returns the next outgoing edge after <code>e</code> from vertex <code>v</code> (to be used in conjunction with <code>first_out_edge</code> for sequential access of adjacency lists)
<code>first_in_edge(vertex:v)</code>	returns the first edge into vertex <code>v</code> (for directed graphs; first edge on <code>v</code> 's adjacency list if undirected)
<code>next_in_edge(vertex:v,edge:e)</code>	returns the next edge after <code>e</code> going into vertex <code>v</code> (analogous to <code>next_out_edge</code>)
<code>first_edge()</code>	returns the first edge from the list of all edges in the graph
<code>next_edge(edge:e)</code>	returns the next edge after <code>e</code> in the list of all edges
<code>vertex_label(vertex:v)</code>	returns the label of vertex <code>v</code> in an allocated string
<code>edge_label(edge:e)</code>	returns the label of edge <code>e</code> in an allocated string

TABLE 5
Changing Logical Graph Attributes in GDR

<code>add_vertex(int:x,y;string:label)</code>	adds a new vertex and returns its id; the vertex is drawn at position <code>(x,y)</code> and is given <code>label</code> as its label
<code>add_edge(vertex:v1,v2;string:label)</code>	adds a new edge from <code>v1</code> to <code>v2</code> and returns its id; label of the new edge is <code>label</code>
<code>delete_vertex(vertex:v)</code>	deletes vertex <code>v</code> and all incident edges from the graph
<code>delete_edge(edge:e)</code>	deletes edge <code>e</code> from the graph
<code>change_vertex_label(vertex:v;string:label)</code>	changes the label of <code>v</code> to <code>label</code>
<code>change_edge_label(edge:e;string:label)</code>	changes the label of <code>e</code> to <code>label</code>

The current implementation accesses vertices by their id, an integer between 0 and `max_vertex()`. Since vertices may get deleted it is necessary to check the whether a vertex corresponding to an id exists (is valid) before attempting to access it.

The next section describes, by way of a detailed example, how internal graph access, using display attributes along with logical attributes, can be exploited to design simple algorithm animations. Other examples are shown in Appendix B.

5. Algorithm Animation Features. To illustrate the capabilities of the animation system, let us take a detailed look at an example. Suppose we want to design a simple animation to illustrate depth-first search on directed graphs. We begin by writing a C program that implements depth-first search — the program that we use is based on the description in Cormen, Leiserson, and Rivest [CLR90], § 23.3. Figure 6 shows the recursive function called as each vertex is visited.

The only part of the code that is consciously written with GDR in mind is the header of the `for_adjacent` loop, which uses the macro described in Figure 5. For now, we have put in print statements for each of the important events in the search. These will later be replaced by calls to appropriate animation routines. Declarations of the global variables, shown in Figure 7, are also needed as is a driver program to start the search at each unvisited vertex, as shown in Figure 8.

The linkage with GDR is completed by an

```
#include "gr.h"
```

at the beginning and a definition of the function `animat` at the end, in this case with a call to the driver:

```
void animat() {DFS();}
```

To test the program on an example, one draws an example using GDR — we'll use the one in [CLR90], page 479. The GDR drawing is shown in Figure 9.

Recall that the ordering of the adjacency lists affects the order in which vertices are visited during depth-first search. In our example we sought to reproduce the sequence illustrated in Figure 23.4 of [CLR90]. Since GDR has no facility for reordering adjacency

```

void DFS_Visit(u)
vertex u;
{
    vertex v; /* current other vertex in adjacency list */
    edge e; /* current edge in adjacency list */

    color[u] = GRAY;
    d[u] = time = time + 1;
    for_adjacent(u,e,v) {
        if (color[v] == WHITE) { /* tree edge */
            printf("tree edge: (%d,%d)\n",u,v);
            DFS_Visit(v);
        }
        else if (color[v] == GRAY) { /* back edge */
            printf("back edge: (%d,%d)\n",u,v);
        }
        else if (d[u] < d[v]) { /* forward edge */
            printf("forward edge: (%d,%d)\n",u,v);
        }
        else { /* d[u] > d[v] -- cross edge */
            printf("cross edge: (%d,%d)\n",u,v);
        }
    }
    color[u] = BLACK;
    f[u] = time = time + 1;
}

```

FIG. 6. Recursive function for depth-first search.

```

#define MAX_VERTEX 50 /* maximum number of vertices */
short color[MAX_VERTEX]; /* WHITE if unvisited, GRAY if on stack, BLACK
                           if finished visit */

#define WHITE 0
#define GRAY 1
#define BLACK 2

int d[MAX_VERTEX]; /* time that vertex is discovered (preorder number) */
int f[MAX_VERTEX]; /* time that visit is finished (postorder number) */
int time; /* current time stamp */

```

FIG. 7. Declarations for depth-first search program.

```

void DFS()
{
    vertex u;

    for (u = 0; u <= max_vertex(); u++) {
        color[u] = WHITE;
    }
    time = 0;
    for (u = 0; u <= max_vertex(); u++) {
        if (color[u] == WHITE && is_valid_vertex(u)) {
            DFS_Visit(u);
        }
    }
}

```

FIG. 8. Driver program for depth-first search.

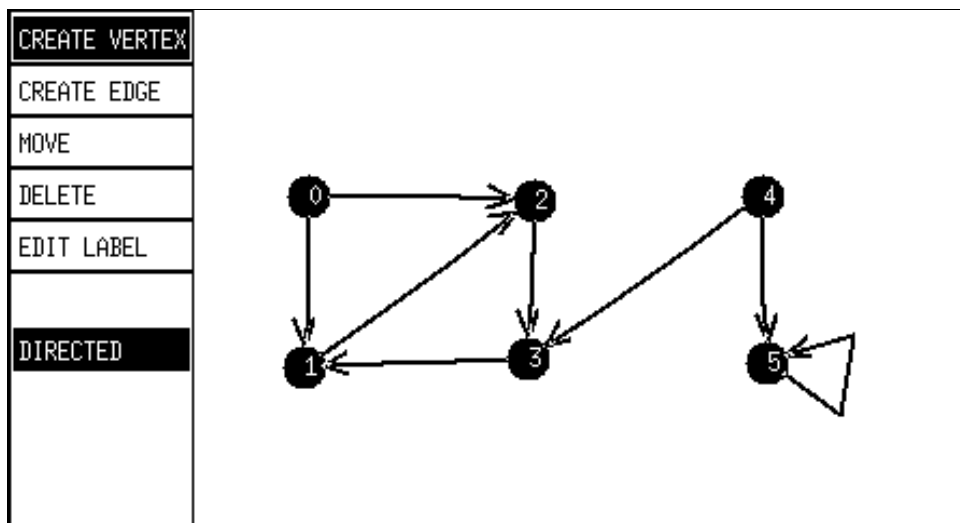


FIG. 9. Graph example for depth-first search animation.

```

for (u = 0; u <= max_vertex(); u++) {
    highlight_vertex(u);
    hide_vertex_label(u);
}
for (e = first_edge(); e != NULL_EDGE; e = next_edge(e)) {
    hide_edge_label(e);
}

```

FIG. 10. Code for initializing the depth-first search animation.

```

Window pwin; /* window for user prompts */
#define XW 1      /* prompt window coordinates */
#define YW 1
#define init_prompt_window (pwin = create_text_window(XW,YW,""))
#define destroy_prompt_window (kill_window(pwin))
#define wait(msg) (write_text_window(pwin,msg),suspend_animation(),\
                    hide_window(pwin))

```

FIG. 11. Declarations for the `wait` macro.

lists, this was accomplished by making sure that the edges were drawn in reverse of the desired adjacency list order (corrections were made by deleting and recreating any edge that was supposed to be at the front of a list).

At the beginning of the animation, we want all vertices to be white and the labels of all vertices and edges to be hidden. The function `highlight_vertex` is a synonym for “whiten vertex”. The current implementation has only two colors, white and black, for vertices (see Sections 6.2 and 6.3 for a discussion of colors in future implementations). We will simulate a gray vertex as a white vertex with an exposed label. Code for the initialization is illustrated in Figure 10. For convenience GDR supplies a list of all the edges in the graph, accessed using the functions `first_edge` and `next_edge`.

The key events in the algorithm are the beginning of a vertex visit, the end of a vertex visit, and the point at which an edge is classified (as tree, back, forward, or cross). At the beginning and end of each vertex visit we would like to inform the user about the progress of the algorithm and pause the animation. The pause, brought about by a call to `suspend_animation` (an operation provided by GDR) returns GDR to edit mode and allows the user to rearrange the display or even to change those parts of the graph that have not been accessed by the algorithm (parts that have been accessed may also be changed but the results may not be predictable). Figure 11 shows the declarations needed for a macro `wait` that effects such a pause.

GDR supplies easy access to X11 facilities allowing the programmer to create and write text into windows at any time during the animation. Four routines accomplish this: `create_text_window` creates a new text window at the specified coordinates (relative to the top left of the GDR display region), `write_text_window` writes a string of text into the specified window, `hide_window` causes the window to be erased from the display, and `kill_window` destroys the window. To aid in positioning message windows

```

void DFS_Visit(u)
vertex u;
{
    /* local declarations for DFS_Visit */

    sprintf(msg,"Start visit, vertex %d\nClick RESUME to continue",u);
    wait(msg);

    /* executable part of DFS_Visit */

    sprintf(msg,"End visit, vertex %d\nClick RESUME to continue",u);
    wait(msg);
}

void DFS()
{
    /* local declarations and animation initialization */

    init_prompt_window;
    wait("Ready to start depth-first search\nClick RESUME to continue");

    /* executable part of DFS (the driver program) */

    wait("End of depth-first search\nClick RESUME to exit animation");
    destroy_prompt_window;
}

```

FIG. 12. *Adding pauses to the animation of depth-first search.*

relative to other corners of the display, the functions `window_height` and `window_width` return the height and width of the display area.

The function `suspend_animation` returns GDR to edit mode and displays a small window with the word `RESUME ?` in the bottom right corner of the display. Execution is resumed by clicking the mouse in the `RESUME ?` window (or by pressing `q` or `Q` to exit from edit mode).

We now put pauses at the beginning of DFS, after the initialization of vertices and edges, and at the end. Pauses are also added to the beginning and end of `DFS_Visit`. These additions are illustrated in Figure 12. The variable `msg` needs to be declared, of course:

```

#define MSG_SIZE 100 /* maximum length of a message */
char msg[MSG_SIZE];

```

We can use the same string variable for all of our messages, including vertex and edge labels — all GDR functions that display strings create allocated copies of them.

So far we have done nothing in the way of actual animation. At the beginning of

```

void DFS_Visit(u)
vertex u;
{
  vertex v; /* current other vertex in adjacency list */
  edge e; /* current edge in adjacency list */

  color[u] = GRAY;
  d[u] = time = time + 1;

  expose_vertex_label(u);
  sprintf(msg,"%d/",d[u]);
  change_vertex_label(u,msg);
  sprintf(msg,"Start visit, vertex %d\nClick RESUME to continue",u);
  wait(msg);

  /* remaining executable part of DFS_Visit */

  un_highlight_vertex(u);
  sprintf(msg,"%d/%d",d[u],f[u]);
  change_vertex_label(u,msg);
  wait("End of depth-first search\nClick RESUME to exit animation");
  destroy_prompt_window;
}

```

FIG. 13. *DFS_Visit with vertex animations.*

a vertex visit, we want to show its label (to represent the color gray) and display the time stamp. At the end of a vertex visit we change the color of the vertex to black and indicate the finishing time along with the time of discovery. Figure 13 shows how the code for `DFS_Visit` is augmented to include these features. Note that the `wait` call at the beginning has been moved to occur after the display of the new time stamp.

To indicate the status of the edges that are encountered, we replace the `printf` statements with appropriate display modifications. Tree edges are highlighted (in our implementation this means the single line segments are replaced by double line segments):

```
highlight_edge(e);
```

Back, cross, and forward edges are indicated by changing the edge label to B, C, and F, respectively, and exposing it (code for back edges is shown):

```
expose_edge_label(e);
change_edge_label(e,"B");
```

Figures 14 through 19 show the display at various strategic points during the animation. Figure 14 shows the beginning of the first call to `DFS_Visit`. Figure 15 shows the first completion of a call to `DFS_Visit`, i.e. the bottom of the first sequence of recursive calls, when vertex 1 changes from gray to black. Note the highlighting of tree edges (0,2), (2,3), and (3,1) and the label on the back edge (1,2). In Figure 16 the first call to `DFS_Visit` ends — the forward edge (0,1) has just been detected. Figure 17

shows the start of the second top-level call — the unvisited vertex 4 is discovered in the main loop of the driver. Figures 18 and 19 show the bottom of the second sequence of recursive calls and the completion of the second top-level call, respectively. Note that the cross edge (4,3) is discovered before the recursive call that visits vertex 5 — it apparently occurs before (4,5) on vertex 4's adjacency list. Also note that the loop at vertex 5 is correctly classified as a back edge.

The animation can be enhanced further by causing each edge to blink as it is processed by the algorithm. The line:

```
    blink_edge(e,DELAY,COUNT);
```

can be placed at the beginning of the `for` loop body in `DFS_Visit`. `COUNT` is the number of times the edge will blink and `DELAY` is the duration of each blink (in machine-dependent units). On a Sun 3/60, the following work well:

```
    #define DELAY 50000
    #define COUNT 4
```

A further enhancement makes the animation more interactive. Instead of starting the search at vertex 0 and then checking vertices by increasing ID's, the user is prompted to select a vertex at which to begin the current search. Consider the following replacement for the main `for` loop in `DFS`.

```
write_text_window(pwin,"Select vertex for starting DFS");
while ((u = select_vertex()) != NULL_VERTEX) {
    if (color[u] == WHITE) {
        DFS_Visit(u);
        write_text_window(pwin,"Select another starting vertex");
    }
    else {
        hide_window(pwin);
        write_text_window(pwin,"Vertex already visited, try again");
    }
}
```

The function `select_vertex` waits until the user either clicks left on a valid vertex or types `Q` or `q` to quit; in the first case, the ID of the valid vertex is returned; in the second, the `NULL_VERTEX` is returned. The call to `hide_window` simply causes the window with the error message to flash, calling attention to itself.

There are other functions that are used in animations to access graph attributes. Tables 6 and 7 give summaries of all the functions that can access geometric and display attributes, respectively.¹ Table 8 shows other functions that enhance user interaction in GDR. The functions in these four tables have all been implemented. Other functions could easily be added to the repertoire. For example, we could add a variety of functions to access and modify geometric attributes, but have not found much use for them so far. Even the modification of vertex position is not used in any of our animations; it would, however facilitate the placement heuristics discussed in Section 6.2.

¹ access to logical attributes was already described in Tables 4 and 5.

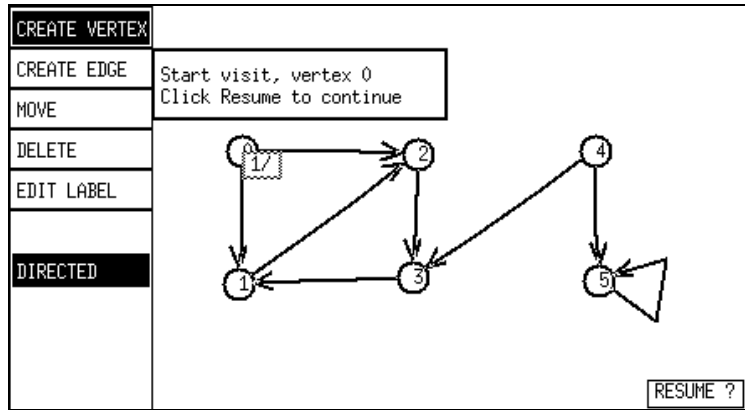


FIG. 14. Beginning of first call to DFS_Visit.

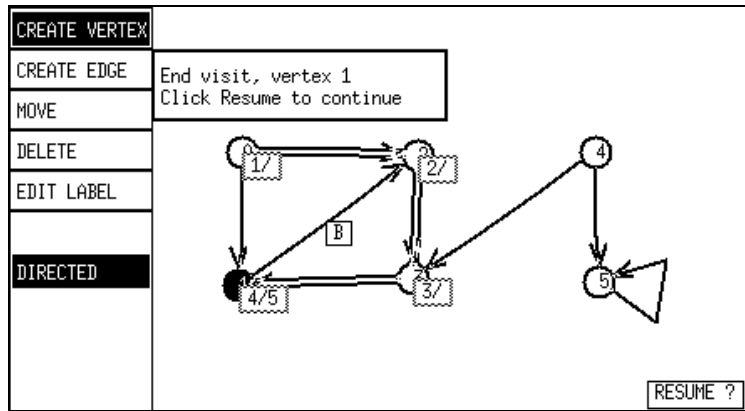


FIG. 15. Bottom of first sequence of recursive calls.

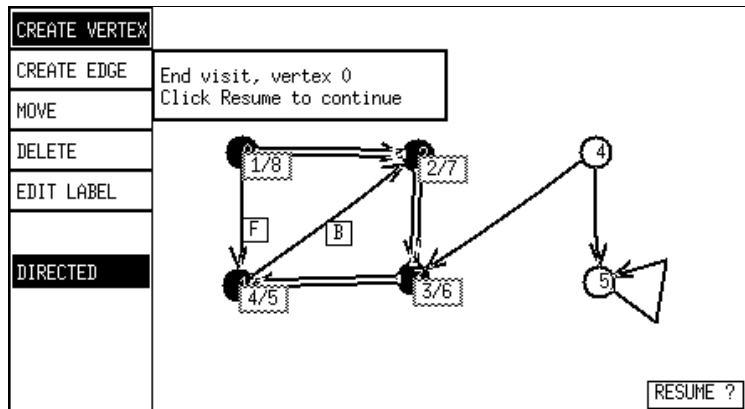


FIG. 16. End of first top-level call to DFS_Visit.

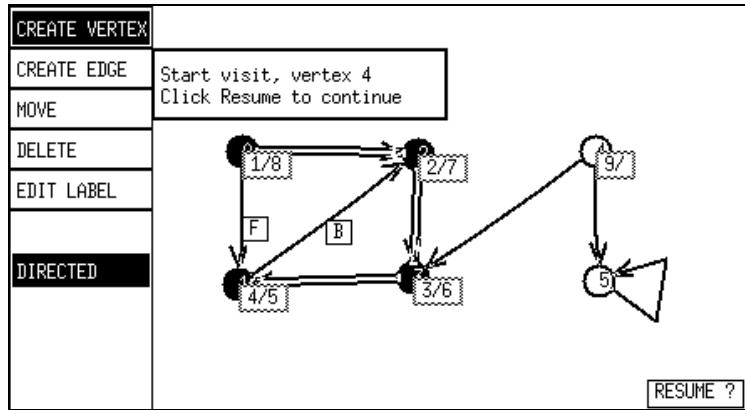


FIG. 17. Beginning of second top-level call to DFS_Visit.

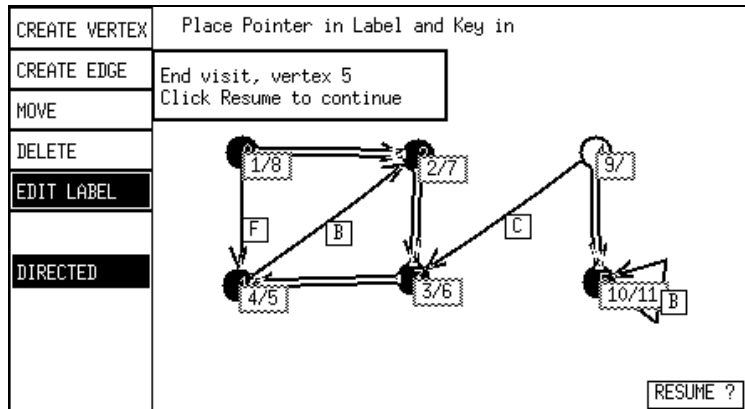


FIG. 18. Bottom of second sequence of recursive calls.

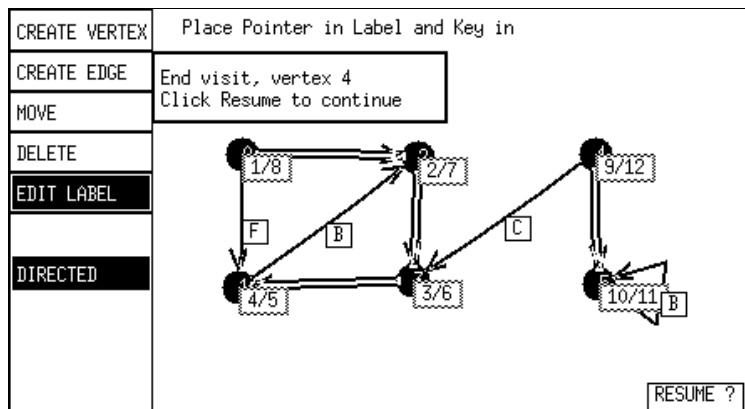


FIG. 19. End of second top-level call to DFS_Visit.

TABLE 6
Access to Geometric Attributes in GDR

<code>add_vertex(int:x,y;string:label)</code>	adds a new vertex and returns its id; the vertex is drawn at position (x,y) and is given <code>label</code> as its label
<code>vertex_x(vertex:v)</code>	returns the x-coordinate of the vertex <code>v</code>
<code>vertex_y(vertex:v)</code>	returns the y-coordinate of the vertex <code>v</code>
<code>move_vertex_relative(vertex:v;int:d_x,d_y)</code>	moves the vertex <code>v</code> to the right <code>d_x</code> units and down <code>d_y</code> units (left and/or up if negative numbers are used); also moves the associated edges; returns <code>FALSE</code> on error, else returns <code>TRUE</code>
<code>window_width()</code>	returns the width of the graph display area
<code>window_height()</code>	returns the height of the graph display area

TABLE 7
Access to Display Attributes in GDR

<code>is_highlighted_vertex(vertex:v)</code>	returns <code>TRUE</code> if the vertex <code>v</code> is highlighted
<code>is_highlighted_edge(edge:e)</code>	returns <code>TRUE</code> if the edge <code>e</code> is highlighted
<code>is_exposed_vertex_label(vertex:v)</code>	returns <code>TRUE</code> if the label of vertex <code>v</code> is exposed
<code>is_exposed_edge_label(edge:e)</code>	returns <code>TRUE</code> if the label of edge <code>e</code> is exposed
<code>highlight_vertex(vertex:v)</code>	highlights vertex <code>v</code> , i.e. makes its background white
<code>un_highlight_vertex(vertex:v)</code>	removes highlighting from vertex <code>v</code> , i.e. makes its background black
<code>highlight_edge(edge:e)</code>	highlights edge <code>e</code> , i.e. doubles all of the line segments representing <code>e</code>
<code>un_highlight_edge(edge:e)</code>	removes highlighting from edge <code>e</code> , i.e. represents the edge as single line segments
<code>expose_vertex_label(vertex:v)</code>	causes the vertex label of <code>v</code> to appear on the display
<code>expose_edge_label(edge:e)</code>	causes the edge label of <code>e</code> to appear on the display
<code>hide_vertex_label(vertex:v)</code>	causes the vertex label of <code>v</code> to be erased from the display
<code>hide_edge_label(edge:e)</code>	causes the edge label of <code>e</code> to be erased from the display

TABLE 8
Miscellaneous Programmer Functions in GDR

<code>blink_vertex(vertex:v,int:delay,count)</code>	causes vertex <code>v</code> to blink (change from unhighlighted to highlighted and back) <code>count</code> times with a <code>delay</code> time units between each change (time units are dependent on machine speed)
<code>blink_edge(edge:e,int:delay,count)</code>	causes edge <code>e</code> to blink (details same as vertex blinking)
<code>print_graph_data(string:name;flag:append)</code>	outputs the current graph, in GDR format, to file <code>name</code> ; if <code>append</code> is <code>TRUE</code> , the existing file gets appended, else it is overwritten (can be used for automatic generation of stills from animation)
<code>suspend_animation()</code>	gives control back to edit mode; program execution is resumed when user clicks left in the <code>RESUME ?</code> window or presses <code>q</code> or <code>Q</code>
<code>create_text_window(int:x,y;string:message)</code>	creates a window with upper left corner at position <code>(x,y)</code> and <code>message</code> written into it (<code>message</code> can have multiple lines, delimited by <code>\n</code>), and returns a window identifier (type <code>Window</code>)
<code>write_text_window(Window:w,string:message)</code>	changes the content of window <code>w</code> to <code>message</code> and causes window <code>w</code> to appear (if hidden)
<code>hide_window(Window:w)</code>	makes window <code>w</code> disappear from view
<code>kill_window(Window:w)</code>	destroys the window <code>w</code> and unmaps it in the process; warning: any further references to the window identifier <code>w</code> will result in an X Window Protocol error.
<code>get_xy_from_mouse(int reference:x,y)</code>	prompts user to click the mouse and the <code>x</code> and <code>y</code> coordinates are passed back by reference (returns <code>FALSE</code> if user hits <code>[Control-C]</code> to abort, else returns <code>TRUE</code>)
<code>select_vertex()</code>	returns the id of a vertex selected by left mouse click

6. Extensions to GDR. This section discusses existing and anticipated extensions to GDR. Some of these were part of the original design, but their implementation was deferred in the interest of rapid development of a working prototype. Others were conceived as the development of animations using GDR led us to envision new possibilities.

6.1. Vtview. One significant development is the implementation of a spinoff tool based on GDR. VTVIEW [Tre92] (vt stands for *Verifiers Toolkit*) is a graphical editor that supports the modular design and analysis of concurrent systems such as communications protocols. The tool allows users to define *networks* consisting of *sites* connected to one another using communication links; each site may itself contain a network or an individual process in the form of a finite-state machine. In contrast with other graphical design tools [MSG88, RdS90], VTVIEW provides a true abstraction and modularization mechanism while permitting bottom-up as well as top-down system design.

VTVIEW is intended to be a tool that can be interfaced with other tools for analyzing and verifying concurrent systems. Accordingly, the tool follows the same object-oriented design strategy as GDR. VTVIEW permits users to define *network* objects, and it also provides abstract access functions that programmers of other tools can use to manipulate the different pieces of networks. Thus, programmers desiring to build systems that interface with VTVIEW need not know the specifics of the data structures used to represent networks. At the moment, one such tool is being built that translates networks created using VTVIEW into a format recognizable by the Concurrency Workbench [CPSar], a system for verifying concurrent systems. Another tool is being planned that would permit the graphical simulation of networks designed in VTVIEW.

As with GDR, VTVIEW is designed to be highly portable. The system is implemented on C and uses the Motif widget set in conjunction with X-windows to provide the graphic features available to the user.

6.2. Aesthetics. As is apparent from the drawings in Section 5, there is considerable room for improvement in the aesthetics of the images created by GDR. This is an important issue not only to the viewers of animations but also for the creation of “stills” from an animation or in the use of GDR to edit drawings of graphs independent of any programs. Enhancements to the aesthetics of GDR fall into three main categories, listed in order of increasing difficulty: new editing capabilities, new graph attributes, and automated drawing features.

There are many editing features that can be added to GDR without significantly increasing the complexity of the software or the user interface. For example, there could be commands to align selected vertices horizontally or vertically so that more of the edges would appear as horizontal or vertical straight lines. Or the placement of vertices could be aligned to a grid whose density was specified by the user. Also useful would be menu-selected options on the size and shape of vertices or the thickness of lines used for edges (with defaults taken from a user-supplied configuration file). There could be an option to display edges as splines through the knots rather than as three line segments. Although the position of a vertex label is an attribute, it cannot be changed under user

control and is fixed so that the upper-left corner of the label is at the center of the vertex. Position of the label should be controlled by the user. The main difficulty in doing this is the ambiguity between a command to move a vertex and one to move a vertex label (if vertex and label are in the same position). One possibility is to adopt the convention that a mouse drag in `EDIT LABEL` mode is used to move a label, while `MOVE` mode is reserved for moving vertices and knots. Finally, to simplify the editing of graphs, there could be commands to select, move, and copy vertex-induced subgraphs as a unit. This would simplify the creation of aesthetically pleasing drawings of graphs with lots of symmetry.

New attributes are discussed in more detail in the next section. The main issue here is the use of additional display attributes to increase the number of ways vertices and edges can be distinguished visually from each other during an animation. The most natural extension here is, of course, the use of color to display different subsets of vertices and edges (thickness and/or shading could be used on a black and white monitor in place of color). Another place where the addition of new attributes could enhance aesthetics is in the use of multiple knots, the knots being represented as a linked-list of 0 or more positions. This is the strategy used by `VTVIEW`.

The current implementation of GDR leaves all decisions about placement of objects (except for the initial placement of knots and labels) to the user. If an animation creates new vertices, the user must be prompted for their location, for example. Moreover the current version of GDR has no facility for reading a logical graph as input and creating a drawing of it from scratch. The issue of creating aesthetically pleasing drawings of graphs has been studied extensively (Di Battista, Eades, and Tamassia [DET93] give an annotated bibliography of recent results).

There are actually two issues for GDR here. One is the conversion of the logical representation of a graph obtained from another source to a physical (and logical) representation that can be manipulated by GDR. The best strategy here may be to deal with this external to GDR, i.e. use separate tools to translate logical representations of graphs in some standard format to GDR files. This has the advantage that users can choose the translation tool or interactive mechanism that best suits the particular application (e.g. some tools work only for planar graphs, others are specifically for directed acyclic graphs, etc.).

The other issue is heuristics for placement of new objects added to an existing graph under user or programmer control. The placement of labels and knots, which is already automated, could be improved by the use of better heuristics. For example, the current implementation introduces non-straight edges only in the case of multiple edges between the same two vertices. In some applications, however, it is desirable to have a horizontal row of vertices with some edges between non-consecutive vertices of the row. Such edges end up being drawn on top of each other and the intervening vertices, and their knots are difficult to move because they may coincide with vertices or knots of other edges. It is also difficult for the user to figure out what happened — in the current implementation (exclusive or is used for drawing) the placement of the new edge appears to erase other edges as illustrated in Figure 20 (similar problems occur with

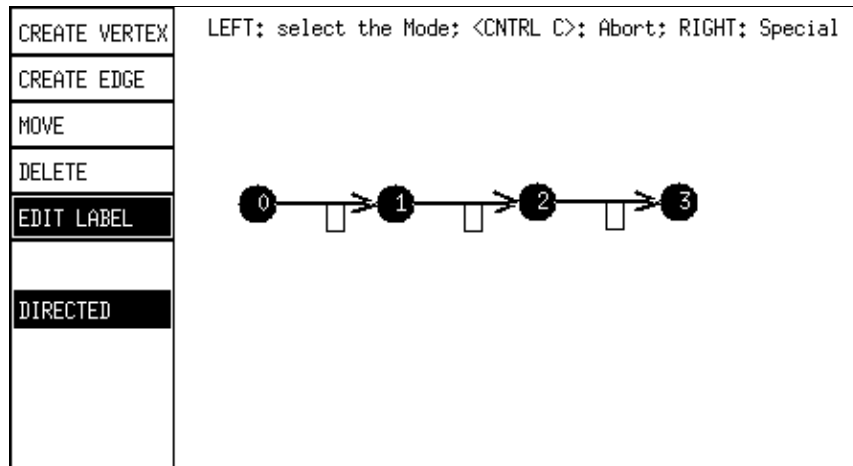
other “graphics contexts”).

6.3. Representation. In keeping with the philosophy that GDR is a tool rather than a system, we need to have a simple universal mechanism that allows GDR to communicate with other tools. This mechanism already exists in the form of the GDR file, which records, in a straightforward ASCII format, the attributes of vertices and edges of a graph. Figure 22 shows the GDR file for the graph in Figure 21. The first line gives information about the graph as a whole: the sequence “ \wedge \$(” identifies this as a GDR file, 3 is the number of vertices in the graph, and 1 identifies this as a directed graph (an undirected graph has 0 in this position). Then there are two lines in the file for each vertex of the graph. The first line gives the vertex id, its position, its label (terminated by ?; this allows for empty labels), the position of the label, and the *display attributes byte*, which tells whether the vertex is highlighted or not and whether the label is exposed or hidden. The second line gives a list of the edges that have that vertex as their tail, terminated by a -1 . The entry for each edge gives the head of the edge (as a vertex id), the edge label, the position of each knot and the label (6 coordinates in all), and the display attributes byte for the edge.

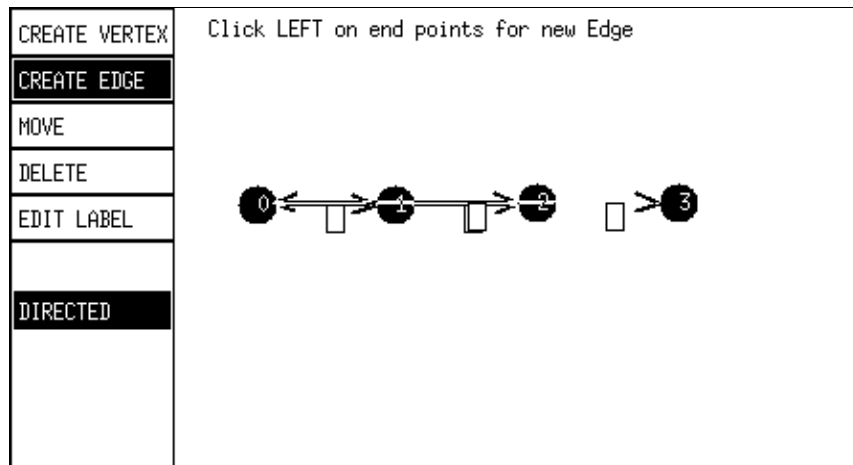
Tools that translate a GDR file into either a simple logical representation of the graph (adjacency list format or just a list of all the edges) or a physical representation (for example, PostScript) can easily be developed. Procedures that access GDR files can be written in the same style that we used for program access to internally stored attributes. However, GDR file format is still under development. A procedure to translate GDR format to logical representation already exists and would be easily adaptable to changes in GDR file format. Translation to physical representation is a more complex issue, since it is useful in a variety of contexts.

In considering the translation of GDR format to a physical graph representation, we need to look at the implications in three areas. First, there is the creation of *stills*, sequences of printable picture, from an animation. The figures showing GDR in action in this report were created using `xwd` and `xpr`, i.e. they are bitmap dumps of the window image, a rather inefficient format for storing images of graphs. For stills a literal translation of the display image to PostScript would be sufficient. The stills would accurately reproduce what was seen in the animation. Recall that the creation of stills as GDR files is a simple matter — just use the `SAVE` menu option at any pause point in the animation.

A second issue is the use of GDR as a tool for creating drawings of graphs for documents, i.e. as a drawing tool that is smart about the logical structure of graphs. Here the desired scheme is a tool that takes a GDR file and a *style file* as input and produces a PostScript drawing as output. The style file would contain information about the desired thickness of lines used for edges, the size and shape of vertices, fonts used for labels, etc. Once the concept of a style file is introduced, it is natural to have GDR itself use such a file to customize the screen display of the graph. One advantage of the style-file approach is that the display attributes can be interpreted in different ways depending on the style file currently in use. For example, the display byte for vertices and edges, which currently uses only 2 bits, could be expanded to 8 or more



(a) before the addition of edge (3,0)



(b) after the addition of edge (3,0)

FIG. 20. Difficulties caused by default placement of knots for a new edge.

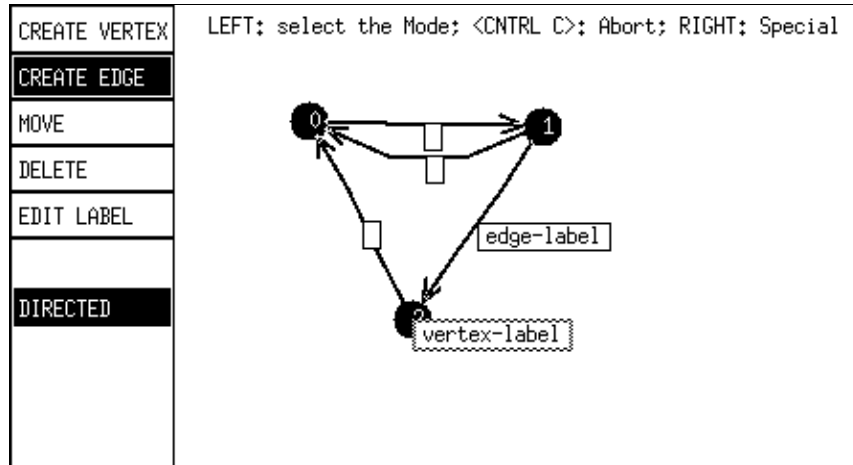


FIG. 21. Graph example for illustrating GDR file format.

```

^$)( 3 1
0 156 60 ? 156 60 0
 1 ? 196 61 236 62 216 61 1 -1
1 278 63 ? 278 63 0
 0 ? 237 79 197 78 217 78 1 2 edge-label? 256 96 232 129 244 113 1 -1
2 210 163 vertex-label? 210 163 1
 0 ? 192 129 176 95 184 112 1 -1

```

FIG. 22. GDR file for the example in Figure 21

bits. The extra bits could be used to indicate color on a color monitor/printer or some combination of shape, thickness, and texture on a black and white monitor/printer. The style file could customize the interpretation of the display bits.

Finally, GDR file format could become a convenient mechanism for compact storage of graph drawings or for transmitting such drawings electronically to other sites. It is therefore important that the format of GDR files be carefully conceived and easily extendible. Such extensions as the use of more than 2 knots must be weighed carefully with respect to their impact on GDR file format.

6.4. Object-Oriented Approach. Our original vision of GDR was as a stand-alone tool that could interact with programs written in any suitable language. The overall scheme was object-oriented. GDR is a process that encapsulates an object called a graph. Other processes interact with GDR by sending messages that access or modify the attributes of the graph (see Table 1 for a list of attributes and Tables 4 – 7 for a list of functions that access those attributes).

We deferred the strict interpretation of the object-oriented philosophy in order to get a prototype working quickly. An implementation of the original idea would first require development of a suitable protocol for exchanging messages between GDR and other processes. Second, programs that interact with GDR would have to be compiled or preprocessed in a way that would translate GDR function calls to the appropriate

message interchanges.

The advantages of the object-oriented approach are clear. GDR would be able to interact with tools written in languages other than C. There would be no need to create a separate executable copy of GDR for each application program. A single graph display could be accessed simultaneously by several different programs. For example, a user could run one program to effect nontrivial modifications on the graph and another to repeatedly execute a depth-first search or some other analytical routine. A single program could also create and manipulate several GDR displays. An example of this would be a program that creates a random graph in one GDR window and runs some algorithm on it, while another window displays a data structure or an auxiliary graph used by the algorithm.

A major disadvantage of adopting the pure object-oriented approach is a practical one. The techniques for managing processes that communicate are fairly esoteric and system-dependent. Portability, a real advantage of the current GDR implementation, would have to be sacrificed initially. We would also have to define GDR function calls for each algorithm implementation language and devise mechanisms for translating these to the appropriate message interchanges.

6.5. Portability. The current implementation of GDR, because of its use of ordinary C and plain X-Windows, can easily be ported to most Unix-based workstations (so far we have only tested this for DEC and Sun workstations). A feature of GDR that suggests portability to graphical user interfaces other than X-Windows is the simplicity of the routines that access and modify the display of the graph. It is likely that these access routines could be rewritten for adaptation to, for example, a Macintosh environment. Some rearrangement of the current code would be desirable to isolate the parts that interact directly with X-Windows.

Adaptation to other graphical user interfaces (GUI's), particularly Macintosh and DOS interfaces, is important if GDR is to be used in the classroom and even if it is to be used as a research tool. The use of GDR in the classroom presupposes either that every student have access to a computing platform running the appropriate GUI or that there be projection equipment attached to a computer controlled by the instructor. Either scenario is much more likely to exist for Macintosh or DOS environments than it is for Unix workstations running X-Windows (at least given present technology).

There are two issues in the design of the GDR user interface that affect portability. One is the design of the routines that draw the graph on the screen, which can easily be decomposed into primitives that draw vertices, line segments (for edges), and boxes containing text (for labels). The other is the mechanism that responds to keyboard and mouse events, which can be encapsulated in a procedure `input_event`. Handling of input events is similar for the different GUI's. The differences have mainly to do with the translation of keystrokes into character codes.

REFERENCES

- [Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis, Second Edition*. Addison-Wesley, 1988.
- [BK87] J. L. Bentley and B. W. Kernighan. A system for algorithm animation: Tutorial and user manual. Computing Science Technical Report 132, AT&T Bell Laboratories, January 1987.
- [Bro87] Marc H. Brown. *Algorithm Animation*. The MIT Press, 1987.
- [BS89] B. Birgisson and G. E. Shannon. GraphView: An extensible interactive platform for manipulating and displaying graphs. Technical Report 295, Computer Science Department, Indiana University, December 1989.
- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [CPSar] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, To appear.
- [DET93] G. Di Battista, P. Eades, and R. Tamassia. Algorithms for drawing graphs: An annotated bibliography. Available by anonymous ftp from `wilma.cs.brown.edu`: files `/pub/gdbiblio.tex.Z` and `/pub/gdbiblio.ps.Z`, March 1993.
- [Ebe87] J. Ebert. A versatile data structure for edge-oriented graph algorithms. *Communications of the ACM*, 30(6):513 – 519, 1987.
- [JG89] D. Jablonowski and V. A. Guarna, Jr. GMB: A tool for manipulating and animating graph data structures. *Software Practice and Experience*, 19(3):283 – 301, 1989.
- [Mar91] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Inc., 1991.
- [MSG88] J. Malhotra, S.A. Smolka, A. Giacalone, and R. Shapiro. Winston: A tool for hierarchical design and simulation of concurrent systems. In *Proceedings of the Workshop on Specification and Verification of Concurrent Systems*, Stirling, Scotland, 1988.
- [RdS90] V. Roy and R. de Simone. AUTO/Autograph. In *Computer-Aided Verification '90*, pages 477–491, 1990.
- [Tre92] Vikas Trehan. VTVIEW: A graphical editor for hierarchical networks of finite-state processes. Master's thesis, Dept. of Computer Science, North Carolina State University, 1992.

APPENDIX

A. User Documentation. (written for students in an algorithms class; the users are not necessarily familiar with the X-Windows interface)

Manual entry of a new graph.. The following steps are recommended for manually entering a new graph (Directed or undirected).

1. Select **CREATE VERTEX** on the panel (click left in the box labeled **CREATE VERTEX** if this is not already in reverse video). Create the vertices of your graph by positioning the mouse and clicking left for each vertex. The first vertex created will be vertex 0 (usually the start vertex for algorithm simulations). Others will be designated with consecutive numbers (there is currently a limit of 25 vertices).
2. Select **CREATE EDGE** on the panel. To create an edge from vertex i to vertex j , where $i \neq j$, click left with the mouse on vertex i (vertex i will be highlighted – turned white – at this point), then click left again with the mouse on vertex j . An arrow or line connecting the two should appear with an empty rectangular label in the middle. If the line crosses other existing vertices or edges, the diagram may look strange; consult the section on **Moving things around** if this is a problem. To create a loop from vertex i to itself, click twice on vertex i . A message will instruct you to click two more times to indicate the position of the loop. The loop will then appear as a triangle, with vertex i and the mouse positions of the two additional clicks as its three points.
3. If the edges of the graph need to be labeled (for algorithms with weighted edges, for example), select **EDIT LABEL** on the panel. To create or change a label, move the mouse into the appropriate rectangle (a pencil should appear) and type the new label.
4. If your new creation looks ok, you can save it in a file by typing **s** or selecting **SAVE** in the pull-down menu (right mouse button).
5. Now you can run the appropriate algorithm on your graph by following the instructions beginning in paragraph 2 of **Running the algorithm for an existing graph**. Or you can exit from the system by typing **q** or selecting **QUIT GDR** from the pull-down menu.

Moving things around. It is sometimes hard to avoid having edges run into vertices or labels of other edges or even other edges themselves. Wherever two objects coincide, they cancel each other; for example, if two edges appear on top of each other they effectively erase each other. The system has built-in heuristics for handling the placement of multiple edges between the same two vertices, but there are other cases where edges may collide. Careful planning can obviate some of these problems, but it is possible to move vertices, edge labels, and edge *knots*, the two kinks in an edge that is not a straight line (even straight-line edges have knots, but they are hard to see).

To move any of the above objects, select **MOVE** on the panel, point the mouse to the object, press down the middle button, and hold the middle button while dragging the mouse to the desired location. A black dot shows the mouse position while the object

is being dragged. When a vertex is moved, all incident edges (edges into and out of the vertex) and their labels are moved with it. If the vertex being moved has a loop attached to it, the system will prompt you for two additional mouse clicks to determine the new location of the knots on the loop.

Deleting objects. A vertex or an edge may be deleted by selecting **DELETE** on the panel, pointing the mouse at the vertex or edge, and clicking the left button. Do not delete vertex 0, since this is the start vertex, and any new vertices added will have numbers higher than the current highest numbered vertex. That is, once a vertex is deleted, its number will not be reassigned to newly created vertices. The pull-down menu (right mouse button) also has a **CLEAR** option, which deletes the whole graph and starts from scratch.

Running the algorithm on an existing graph.. To read a file containing a graph, either type **r** or use the right mouse button to pull down the menu and release it while it is pointing to the **READ IN** option. A smaller query window will appear, asking the name of the file. Type the file name followed by **[Return]**. Use the **[Delete]** button to back up over mistakes. If the file does not exist or is unreadable, an error message will appear in the original **xterm** window (you may have to click the title bar of that window with the left mouse button to see the message; this “feature” will be fixed in future versions; note: the *title bar* of any window is the top border area containing the name of the window — clicking on this will bring the window into the foreground).

After the graph appears in the window, type **p** to invoke the simulator program (or select **RUN FILE** from the menu). To start the simulation or to continue it at a breakpoint, either click left with the mouse in the small window labeled **RESUME ?** (at the bottom right) or type **q**. When the simulation stops at any breakpoint, you may alter the graph using any of the commands, or save the current picture in a file (using the **SAVE** command).

List of commands. *Panel options*

CREATE VERTEX Left mouse click creates a new vertex (number is one greater than most recently created vertex, first vertex has number 0).

CREATE EDGE Left mouse click on vertex i followed by left click on vertex j creates an edge from i to j . If $i = j$, two more clicks define knots of loop.

MOVE Point to vertex, edge label, or knot, push middle mouse button and hold until mouse points to new position.

DELETE Left mouse click on vertex or edge deletes the vertex or edge.

EDIT LABEL Move mouse inside rectangle representing a label to change the label; any typed text becomes the new label (**[Delete]** acts as backspace). Left mouse click on vertex exposes the label or hides it if it's already exposed.

Menu options

SAVE (or type **s** or **S**) Current graph is saved in a file (user is prompted for the file name and asked to confirm the action if the file already exists).

READ IN (or type **r** or **R**) A graph is read from a file (user is prompted for file name). Current graph is erased.

RE-DRAW The window is redrawn (in case things get messed up).

CLEAR The current graph is erased.

RUN FILE (or type `p` or `P`) The animation program is executed. Whenever a window with the prompt `RESUME ?` appears at the bottom right of the screen, control returns to the user interface: the user can edit the graph (e.g. move things around, modify labels, etc.). The program continues execution if there is a left mouse click in the `RESUME ?` window, or a `QUIT GDR` command issued.

QUIT GDR (or type `q` or `Q`) Exit from the GDR program, or resume execution of the animation program if it is running.

Note that `[Control]-C` acts as a general panic button: any of the above commands is aborted in response to it.

B. Detailed Examples.

B.1. Prim's minimum spanning tree algorithm. The following animated implementation of Prim's algorithm is based on the description of Prim's algorithm in Baase [Baa88, Section 4.2]. The minimum spanning tree is grown from vertex 0, one vertex at a time. Before each pause, the animation blinks the next edge to be added to the tree, highlights the new tree vertex, highlights any new fringe edges (edges between tree and non-tree vertices), and deletes edges between tree vertices that are not tree edges. Costs are shown only for fringe edges, except that when the final vertex is added, the cost of each minimum spanning tree edge is shown (we could have also added a window to display the total cost of the tree). Vertex labels are used to indicate the cost of adding a vertex to the tree. Figure 23 shows the beginning of the animation on a particular graph (the user has entered edge costs by editing the edge labels). Figure 24 shows an intermediate stage: edges $(0,2)$ and $(2,3)$ have already been added to the tree. Figure 25 shows the end of the animation.

The main drawback of this particular animation is that non-tree edges are actually deleted during execution, so the original graph can only be recovered if the user has saved it in a file. This could easily be remedied by adding the ability to hide a vertex or edge (not only its label), so that non-tree edges could be hidden rather than deleted.

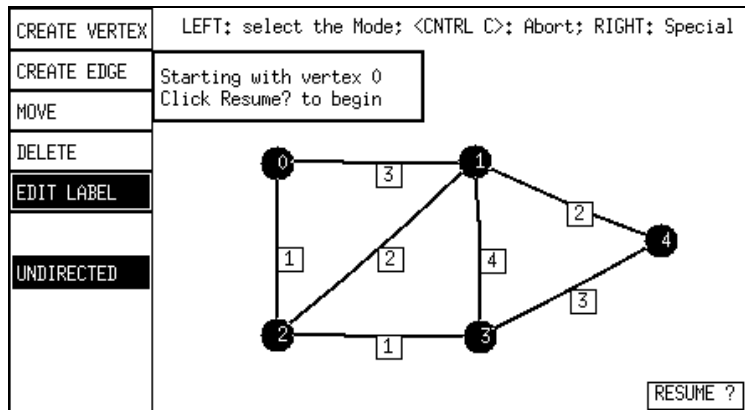


FIG. 23. Beginning of Prim's algorithm.

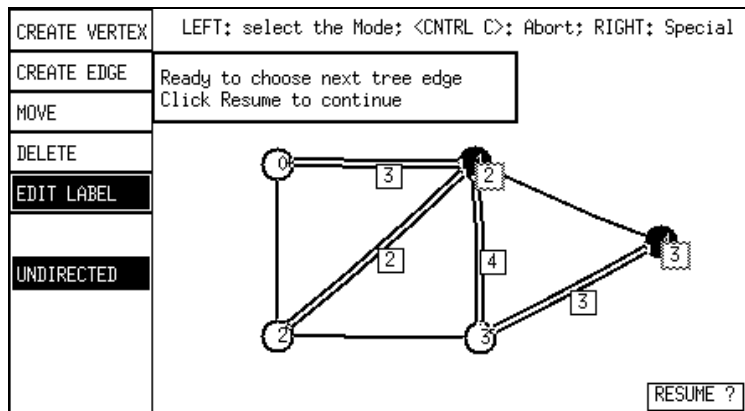


FIG. 24. Intermediate step in Prim's algorithm.

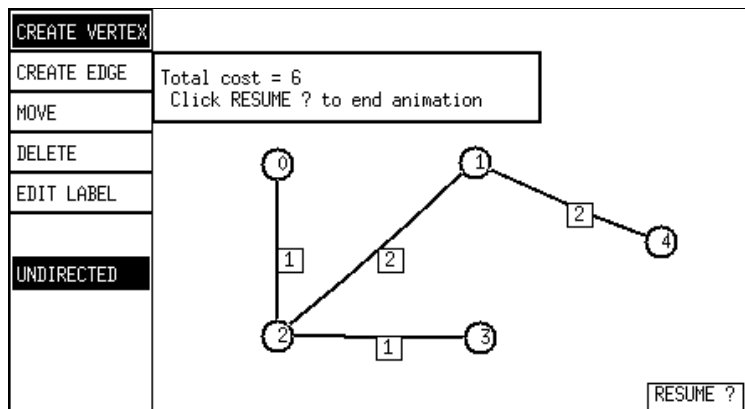


FIG. 25. End of Prim's algorithm.

```

#include "gr.h"

#define MAX_VERTEX 50
edge candidate[MAX_VERTEX]; /* candidate edge for this vertex */
int cost[MAX_VERTEX];      /* cost of the candidate edge */
int status[MAX_VERTEX];    /* status of this vertex -- use defines below */
#define in_tree 0
#define unseen 1
#define fringe 2

/* declarations and macros for the animation */
#define MSG_SIZE 100 /* maximum length of a message */
char msg[MSG_SIZE];
Window pwin; /* window for user prompts */
#define XW 1      /* prompt window coordinates */
#define YW 1
#define init_prompt_window (pwin = create_text_window(XW,YW,""))
#define destroy_prompt_window (kill_window(pwin))
#define wait(msg) (write_text_window(pwin,msg),suspend_animation(),\
                    hide_window(pwin))
#define DELAY 50000
#define COUNT 4

typedef struct cell {
    struct cell *next;
    vertex vx;
} *V_LIST;
V_LIST fringe_list = NULL;

void add_to_fringe_list(v)
    vertex v;
{
    V_LIST new;

    new = (V_LIST)calloc(1,sizeof(struct cell));
    new->next = fringe_list;
    new->vx = v;
    fringe_list = new;
}

void remove_from_fringe_list(prev)
    V_LIST prev; /* pointer to cell before the one being removed;
                 NULL if first cell is to be removed */
{
    V_LIST out = (prev == NULL) ? fringe_list : prev->next;
    if (prev == NULL)
        fringe_list = fringe_list->next;
    else

```

```

    prev->next = out->next;
    free(out);
}

int weight(e) /* returns value of integer label */
    edge e;
{
    int wt;
    sscanf(edge_label(e),"%d",&wt);
    return (wt);
}

char *new_string(i) /* returns a string representation of integer i */
    int i;
{
    char *stg;
    stg = (char *)calloc(10,sizeof(char));
    sprintf(stg,"%d",i);
    return (stg);
}

animat()
{
    vertex x; /* current vertex */
    V_LIST x_pos; /* x's position in the fringe list */
    int edge_count = 0; /* number of edges in tree */
    vertex y; /* generic vertex for loops */
    edge e; /* generic edge */
    int maxv = max_vertex() + 1; /* number of possible vertices */
    int n = 0; /* number of valid vertices */
    int c; /* a cost */
    int min_cost; /* minimum cost in current iteration */
    int total_cost; /* total cost of MST */
    V_LIST prev,ptr; /* generic pointers to fringe list */

    init_prompt_window;
    wait("Starting with vertex 0\nClick RESUME to begin");

    /* hide labels on all edges */
    for (e = first_edge(); e != NULL_EDGE; e = next_edge(e)) {
        hide_edge_label(e);
    }

    /* initialize status and count number of valid vertices */
    for (y = 0; y < maxv; y++) {
        if (is_valid_vertex(y)) n++;
        status[y] = unseen;
    }
}

```

```

total_cost = 0;
x = get_a_vertex();
status[x] = in_tree;
highlight_vertex(x);

while (edge_count < n-1) {
    /* traverse adjacency list for x */
    for (e = first_out_edge(x); e != NULL_EDGE;
        e = next_out_edge(x,e)) {
        y = other_vertex(x,e);
        if (status[y] == fringe) {
            highlight_edge(e);
            expose_edge_label(e);
            if ((c = weight(e)) < cost[y]) {
                /* y has a new candidate edge */
                change_vertex_label(y,new_string(c));
                candidate[y] = e;
                cost[y] = c;
            }
        }
        if (status[y] == unseen) {
            /* y is a new fringe vertex */
            c = weight(e);
            highlight_edge(e);
            expose_edge_label(e);
            change_vertex_label(y,new_string(c));
            expose_vertex_label(y);

            status[y] = fringe;
            add_to_fringe_list(y);
            candidate[y] = e;
            cost[y] = c;
        }
    } /* end of adjacency list traversal */

    wait("Ready to choose next tree edge\nClick RESUME to continue");

    /* choose next tree edge (find lowest cost fringe vertex) */
    if (fringe_list == NULL) {
        printf("Graph is not connected\n");
        return;
    }

    min_cost = MAX_INT;
    for (ptr = fringe_list, prev = NULL;
        ptr != NULL; prev = ptr, ptr = ptr->next) {
        if (cost[ptr->vx] < min_cost) {

```

```

        min_cost = cost[ptr->vx];
        x = ptr->vx;
        x_pos = prev;
    }
} /* x is lowest cost vertex on fringe list */
total_cost = total_cost + min_cost;
remove_from_fringe_list(x_pos);
status[x] = in_tree;
edge_count++;

blink_edge(candidate[x],DELAY,COUNT);
un_highlight_edge(candidate[x]);
hide_edge_label(candidate[x]);
highlight_vertex(x);
hide_vertex_label(x);

/* remove redundant edges */
for (e = first_out_edge(x); e != NULL_EDGE;
     e = next_out_edge(x,e)) {
    y = other_vertex(x,e);
    if (status[y] == in_tree && !are_edges_equal(candidate[x],e)) {
        /* edge xy is redundant */
        delete_edge(e);
    }
}
} /* end of main loop */

/* expose labels on all remaining (tree) edges */
for (e = first_edge(); e != NULL_EDGE; e = next_edge(e)) {
    expose_edge_label(e);
}

/* display total cost of MST */
sprintf(msg,"Total cost = %d\n Click RESUME to end animation",
        total_cost);
wait(msg);
} /* animat ends */

```

B.2. Other Animations. Instead of presenting detailed code for each of the other animations we have implemented, we give a brief description of each and show pictures illustrating them in action.

Nondeterministic finite automaton simulator. The animation program described in this section simulates a nondeterministic finite automaton with empty transitions (an NFA- Λ). The user draws an NFA on the display (a labeled directed graph). The label of each edge is a list of the symbols for the corresponding transition, optionally these are separated by commas (actually the program merely checks to see if the symbol is present in the label). The special symbol \wedge denotes an empty transition. Figure 26 shows the display for the example in Figure 5-8 of Martin's text [Mar91] with an added empty transition from state 0 to state 4.

When the animation is activated, the user is prompted to enter the input string in the label of vertex 0. Then the transition for each input symbol is shown in two phases. The first phase highlights the possible transitions on the next input symbol. The second phase highlights all possible next states, including states reachable from the next states via Λ -transitions, transitions on no input (the Λ -transition edges are highlighted also). The remainder of the input string is shown in a window to the right of the top center of the display. Before the first transition, the animation pauses to highlight the states reachable from the start state via Λ -transitions, as shown in Figure 27 (note: at this point the whole input string, `abab`, appears in the smaller window). Figures 28 and 29 show the two phases of a later transition.

An obvious enhancement to the NFA simulator (and the DFA simulator described in the next section) is the use of color (or texture) to distinguish final states from non-final states. The user could be prompted to select the final states before running the animation and the animation program could report whether or not the input was accepted.

Deterministic finite automaton simulator. The simulator for deterministic finite automata (DFA's) is similar to that for NFA's but considerably simpler. Since only one current state is possible after each transition we use the label of the current state to store the remainder of the input string. The cycle of events for each transition is: (1) pause to let the user view and/or edit the display, (2) unhighlight previous transition edge, if any, (3) highlight current transition edge, (4) hide label on current state, (5) expose label on new state — this label contains what's left of the input, i.e. the previous label with the first symbol deleted. Figures 30 and 31 show a typical transition for a simple DFA.

Depth-first search for undirected graphs. This animation is similar to and simpler than the one doing depth-first search for directed graphs (see Section 5). There is a pause at the beginning and end of each vertex visit and each edge is blinked as it is encountered by the algorithm (in this case, each edge will be blinked twice since it is seen from both of its endpoints). Vertex labels are used to show preorder numbers. Vertices currently on the stack are highlighted. Edges are labeled either T to indicate a

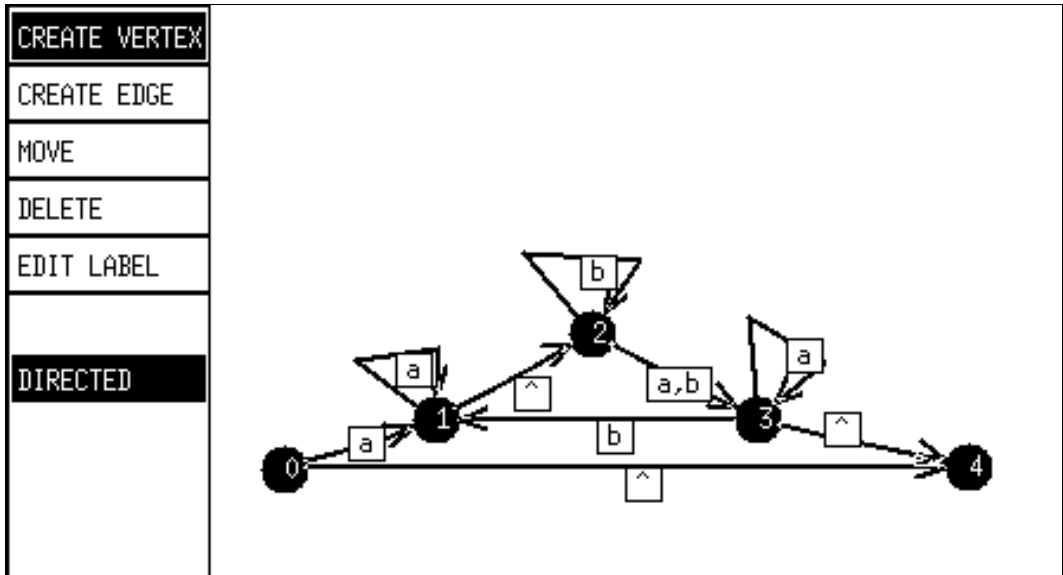


FIG. 26. Sample NFA for simulator program.

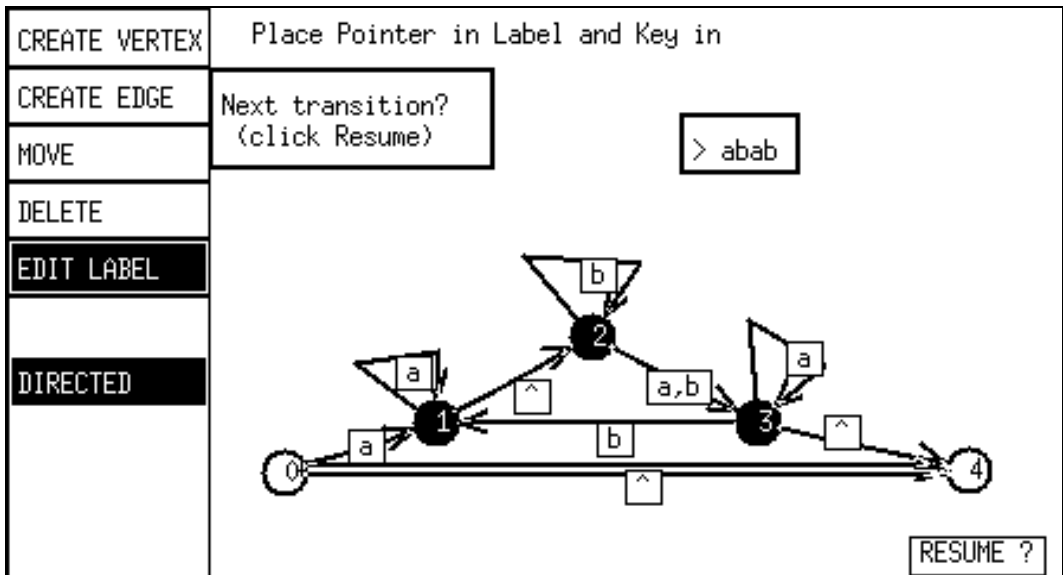


FIG. 27. NFA simulator: empty transitions from the start state.

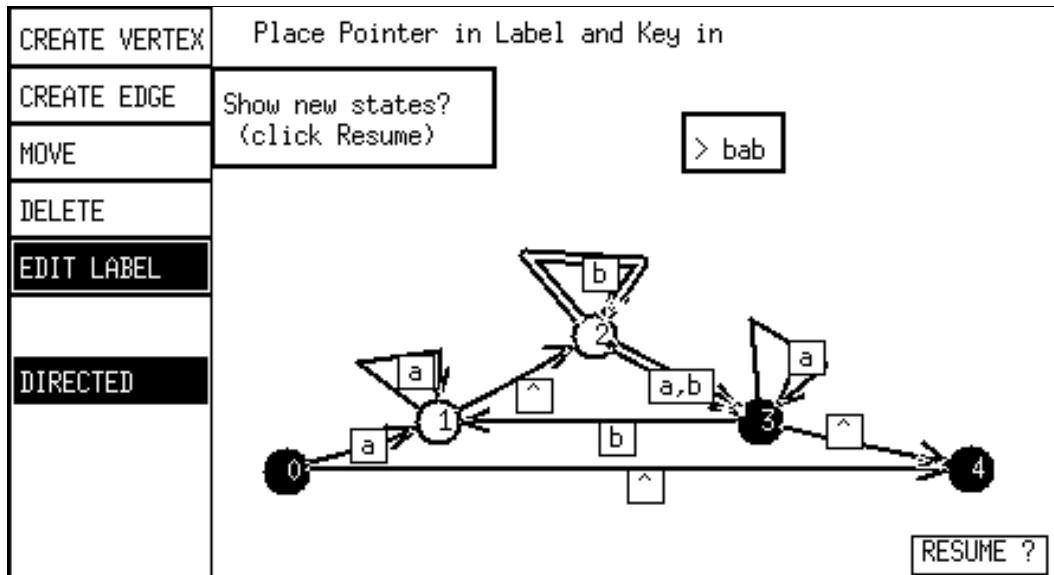


FIG. 28. NFA simulator: first phase of an intermediate transition.

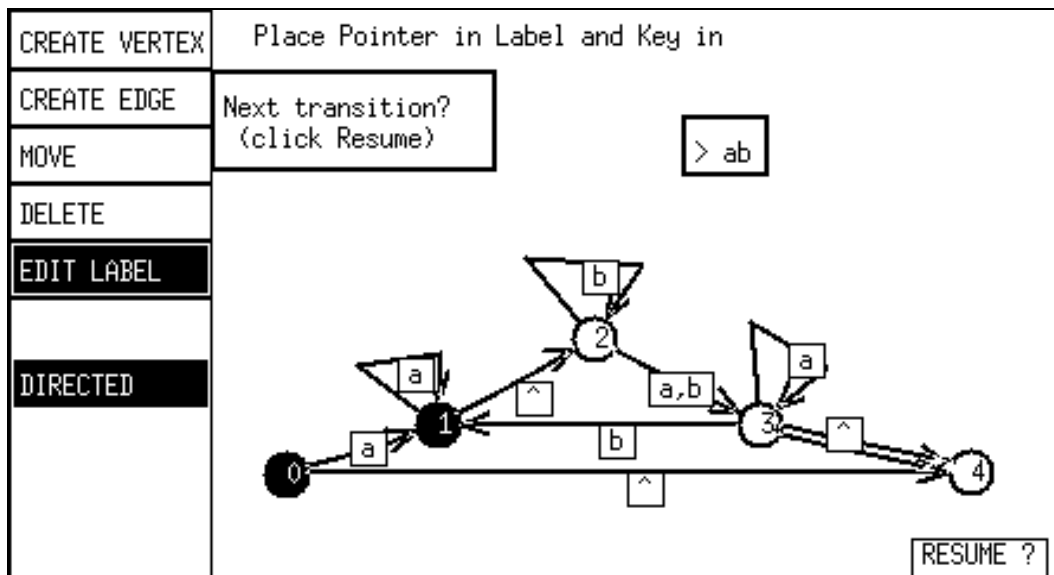


FIG. 29. NFA simulator: second phase of an intermediate transition.

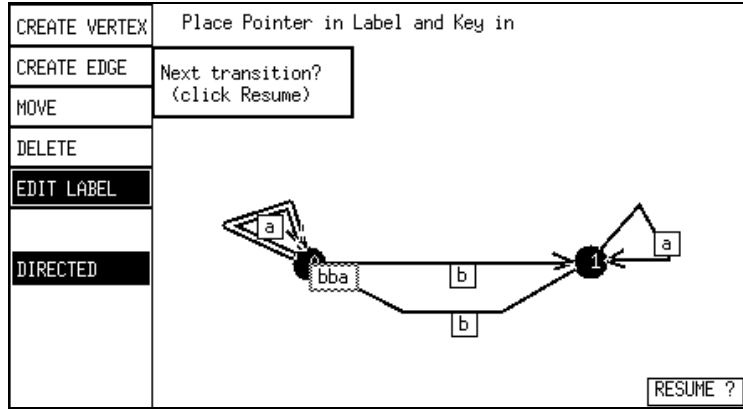


FIG. 30. DFA simulator: before a transition.

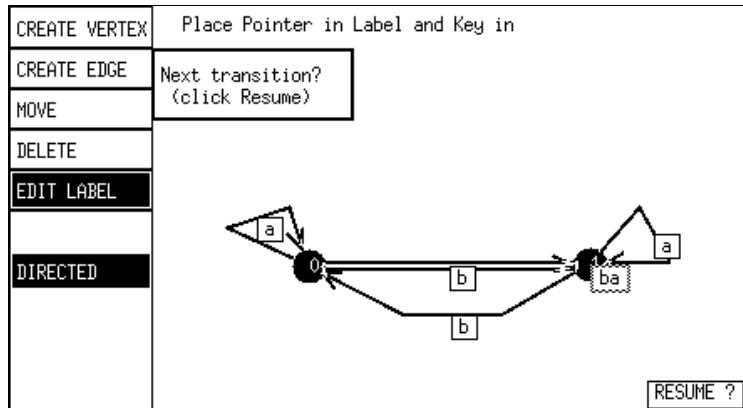


FIG. 31. DFA simulator: after a transition.

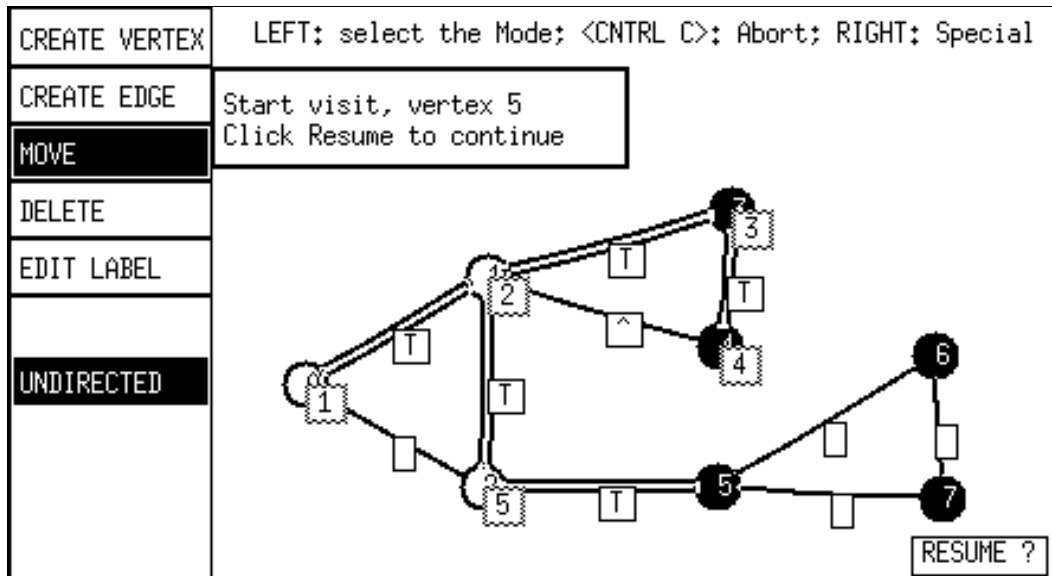


FIG. 32. Undirected DFS: beginning of visit for an intermediate vertex.

tree edge or \wedge to indicate a back edge. In addition, tree edges are highlighted. Figure 32 shows the animation in progress on a small graph.

Biconnected components algorithm. Using the depth-first search animation for undirected graphs as a basis we also developed an animation for an algorithm that identifies biconnected components. The algorithm is taken from Baase [Baa88, page 189], with some modifications (Baase assumes that the algorithm will view the edge vw as being distinct from wv , while our abstract data type considers these to be the same undirected edge). The label of each vertex v now shows two numbers, the preorder number of v ($dfsNumber[v]$ in Baase) and the lowpoint of v ($back[v]$ in Baase), the lowest preorder number reachable from v via a path that follows 0 or more tree edges and ends in a back edge. Tree edges are distinguished only by being highlighted. Labels on the edges are used to identify the biconnected component to which they belong. Figure 33 shows the biconnected component animation on the example used for undirected depth-first search (at the same instant). Figure 34 shows the display at the end of the algorithm.

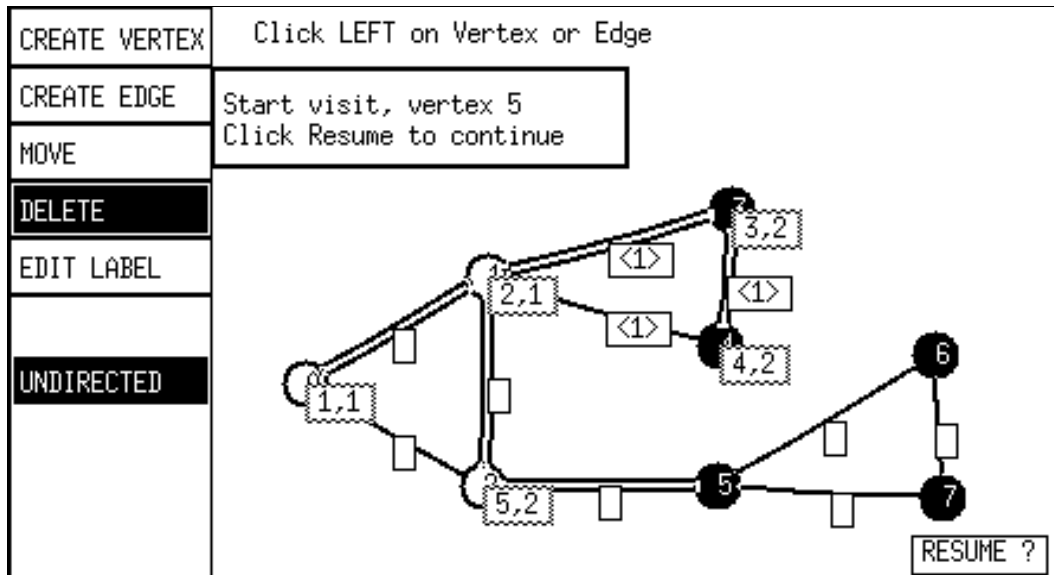


FIG. 33. *Biconnectivity: beginning of visit for an intermediate vertex.*

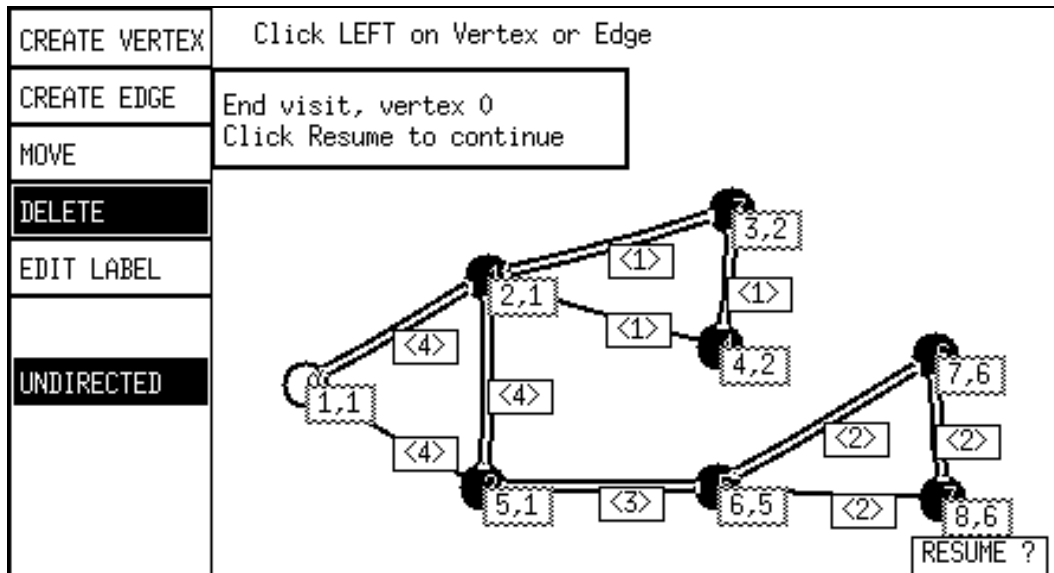


FIG. 34. *Biconnectivity: end of visit for the root.*