1 GAIGS "as is" – that is, without extending its built-in data structure visuals

The GAIGS (Generalized Algorithm Illustration via Graphical Software) is an *algorithm visualization* scripting language that captures and renders *snapshots* of the state of an algorithm at critical points (*interesting events*) in its execution.

Textual snapshots read by GAIGS

The following two rules underlie the structure of GAIGS show files:

1. The group of text lines necessary to draw a data structure is called a textual snapshot. A show file contains one or more textual snapshots, all of which need not be of the same data structure type. When the show file is loaded into the GAIGS program, GAIGS creates one data structure picture corresponding to each textual snapshot.

2. All textual snapshots are divided into three components: the introductory block, the title (or caption), and nodes. The introductory block specifies optional viewing parameters, the type of data structure, the number of text lines per data structure node (which must be at least one), and the optional graphical parameters. The title is an arbitrarily long list of text lines (character strings of 40 or fewer characters – longer lines are truncated) explaining the snapshot. A node is divided into two parts: the information necessary to locate it in relation to other nodes in the data structure and the list of text lines. In the simplest of cases, there will only be one node per snapshot. In such cases, it is relatively easy to link your own graphical renditions into the more standard data structure pictures that are already built into GAIGS.

The introductory block is standard, always constituting at least the first two lines of the textual snapshot. Note in the following format definition that the bracketed parameters are optional:

{view scale x}

{view pan x y}

{view windows n }

{view jump n}

{view algo <URL> }

DataStructureType[1] {Textheight}[2] {TitleTextheight}[3]

Number of text lines per data structure node

Explanation of numbered terms:

***Viewing Parameters*: All these parameters are optional, and case-insensitive.**

***Scale* will scale the snapshot by x, where x is a real, $0 < x <= 2$.**

***Pan* translates the image *x,y* from its original position. x and y are reals in the range from 1 to 1. You may supply any value for x or y, but since the GAIGS viewing environment is only (0,0) to (1,1) any excess panning will force the image completely off the screen. In most cases, it will be better if you "drag" the graphic with the mouse for panning.**

***Windows* requests n windows for the given data structure.**

Algo **will create a code box, fill it with code from text file specified by filename, and highlight the lines in range start thru end. Start and end are integers.** ***start*** **and** ***end*** **are optional. If** ***end*** **is not specified, then only** ***start*** **is highlighted. If neither** ***start*** **nor** ***end*** **are specified, then whatever was previously specified will remain highlighted.**

[1]**DataStructureType: One of the eight built-in data structure types: Stack, Queue, LinkedList, MD_Array, BinaryTree, GeneralTree, Network, Graph OR, your own built-in structure.**

[2]**TextHeight: The text height in world coordinates, which must be greater than zero, to be used for all text in the snapshot except the title. If no text height is specified, this parameter defaults to .01, so that each character's height is 1/100th of the total screen height.**

[3]**TitleTextHeight: The text height in world coordinates, which must be greater than zero, to be used for all text in the snapshot title. If no title text height is specified, this parameter defaults to the text height.**

All of the above optional parameters are intended to give you nearly complete control of the size and appearance of snapshot drawings. Using these parameters will allow snapshots of nearly any size to be drawn so that they nicely fit on the drawing surface, or so that they are at least able to be entirely seen using the "scale" and "pan" options in the viewing environment.

The snapshot title always begins on the line immediately following the introductory block. Note the following format for a snapshot title of n lines:

Title of Snapshot Line 1

–

Title of Snapshot Line n

******** (End of Title Marker)**

If a snapshot has no title, the End of Title Marker should be positioned on the line immediately following the number of lines per data structure node specification. Similarly, if a snapshot does not contain any nodes (i.e., it is an "empty" snapshot), then the End of Snapshot Marker (*^***) should be placed on the line below the End of Title marker.**

After the standard n + 3 lines constituting the Introductory Block and Title, it is necessary to have the textual information that describes the actual content of the data structure.

Example – The built-in stacks, queues, and linked Lists

Since these structures are linear in nature, no extra information in required to position nodes relative to other nodes – the order of the nodes in the file is the order that they will appear in the data structure. The above suggests the following format for a Stack, Queue, or Linked List having k lines per node and n total nodes:

Line 1 of node 1

–

Line k of node 1

–

Line 1 of node n

–

Line k of node n

*****^*** (End of Snapshot Marker)**

Note that, by convention, node 1 is the top of the stack, the head of the queue, and the head of the linked list, so that node n is the bottom of the stack, the tail of the queue, and the end of the linked list (its link field is automatically set to NIL).

Example – The built-in MD_Arrays

In the array textual snapshot the following two integer specifications, both of which must be greater than 0, are required immediately following the End of Title Marker (******):

Number of Rows
Number of Columns

Since GAIGS is able to represent arrays that have indices of any enumerated variable type (integer, character, boolean, or userdefined), the file format requires that the row and column label of each array entry precede the lines of text of that entry. This makes the file structure for an x by y array having k lines per array entry look like the following:

Row label of Row 1, Column 1
Column label of Row 1, Column 1
Line 1 of entry 1,1
.
.
Line k of entry 1,1
.
.
.
Row label of Row x , Column 1
Column label of Row x, Column 1
Line 1 of entry x,1
.
.
Line k of entry x, 1
.

.
.
Row label of Row 1, Column y
Column label of Row 1, Column y
Line 1 of entry 1,y
.
.
Line k of entry 1,y
.
.
.
Row label of Row x, Column y
Column label of Row x, Column y
Line 1 of entry x,y
.
.
Line k of entry x,y
****^*** (End of Snapshot Marker)*

Note that the length of a row label cannot exceed 30 characters, and the length of a column label cannot exceed the length of the longest line of data in the snapshot. Labels longer than these limits will be truncated. Also keep in mind that for onedimensional arrays, all entries should have a column label of "1". This column label will not be included in the array picture.

Example – The built-in graphs and networks

In order to draw a graph or network, information about which nodes are connected is necessary. GAIGS gets this information by requiring that an integer label be specified for each node. These integer labels must start at one and ascend consecutively. So that GAIGS can make the proper node connections, you then include with each node a list of integer node labels to which that node is connected.

Two other parameters can be included with each node contained in a connected node list. For networks, the edgeweight, which is a string of at most 15 characters, is required. For both graphs and networks, the optional DrawArrow delimeter (\A, the "A" must be capitalized), when positioned in front of a connected node's integer label, tells GAIGS to draw an arrow on the connecting line pointing to that node. Use this option to draw directed graphs or networks.

You must choose one of two different methods to draw a graph or network. In the first method, you specify the x and y coordinates of the center of each graph node, keeping in mind that the display is 1.0 x 1.0, so that coordinates outside of this range are allowable, but will be clipped off the screen initially. Using this method allows you to have complete control of the appearance of your graphs/networks. However, this method

also requires you to figure out the graph/network configuration yourself, which could involve some extensive calculation. To avoid manual coordinate calculation, the second method automates the configuration process at the expense of potentially uglier graphs/networks with many edge crossings. This method draws networks and graphs using the following simple algorithm: Given a graph or network containing n nodes, divide a circle (whose circumference is computed to be (2n * node diameter) node diameter) into n arcs, and position a node at the endpoint of each arc around this circle. Then connect each node appropriately into the graph or network. Since the nodes are arranged circularly, every node is capable of being connected to every other node, guaranteeing that every possible graph can be illustrated.

What follows is the file structure for graphs and networks. Note that the only difference between a network and a graph textual snapshot is the edgeweight following each connected node, and that the only difference between textual snapshots drawn by the two different methods is the absence or presence of x and ycoordinates on the first line of each node.

1 (Integer label) {xcoordinate} {ycoordinate}
{\A} Number of Connected Node 1
Edgeweight to Connected Node 1 (Networks only)
.
.
{\A} Number of Connected Node k
Edgeweight to Connected Node k (Networks only)
32767 (End of Connected Nodes Marker= MAXINT on many machines)
Line 1 of Node 1
.
.
Line t of Node 1
.
.
.
n (Integer label) {xcoordinate} {ycoordinate}
{\A} Number of Connected Node 1
Edgeweight to Connected Node 1 (Networks only)
.
.
{\A} Number of Connected Node m
Edgeweight to Connected node m (Networks only)
32767 (End of Connected Nodes Marker = MAXINT on many machines)
Line 1 of Node n
.
.

Line t of Node n
****^*** (End of Snapshot Marker)*

It is important to observe the following restriction when building snapshots of graphs: no node can be connected to a node whose number is less than one or greater than n

Example – Binary and general trees

You may be wondering why there is a separate data structure type for both binary and general trees. "Isn't a binary tree just a special case of a general tree," you may ask. In theory this is certainly true, but when it comes to illustrating them, different rules govern the way in which each tree is drawn. General trees, by convention, are drawn using the rule that a parent should be centered over its evenlyspaced children. This says that if a parent has only one child, that child should lie directly below its parent. The convention used for general trees is definitely not the convention used for binary trees, however. Binary trees are drawn using the rule that a left child must always lie to the left of its parent, and a right child to the right, with the parent centered over its children. For this to hold in the case wherein a parent has only one child, the position in which the missing child *would reside* must be calculated, so that the parent can then be centered over both of its children, even if one of them doesn't exist.

With these two different rules governing the drawing of general and binary trees, two different textual snapshot structures are necessary. Both of them require that tree nodes with their level in the tree (use the convention that the root is at level 0) be listed in the order resulting from a *preorder* traversal of the tree (parent node first, and children next, from left to right). Binary Trees, however, require that each node also contain a childtype flag, either 'L' or 'R', indicating whether it is a left or a right child.

This textual scheme gives an n level tree having k lines per node the following format:

0 (Level of Root Node)
R (Binary Trees only stands for "Root" in this case)
Line 1 of Root Node
.
.
Line k of Root Node
1 (Level of Root Node's Leftmost Child)
L (Level Indicator Binary Trees only)
Line 1 of Root's Leftmost Child
.
.
Line k of Root's Leftmost Child
.
.
(PreOrder Traversal of Root's Leftmost child's

children with nodes as above)
.
.
1 (Level of Root Node's Rightmost Child, if it exists)
R (Level Indicator Binary Trees only)
Line 1 of Root's Rightmost Child
.
.
Line k of Root's Leftmost Child
.
. .
(PreOrder Traversal of Root's Rightmost child's
children with nodes as above)
.
.
****^*** (End of Snapshot Marker)*

Example – Bars (that is, "sticks" representation of numeric data

Still needs to be written

Using flagging characters in snapshot files to color nodes and title lines

For many applications it may be advantageous to highlight crucial nodes in your data structure snapshots. Similarly, in graph and network illustrations, you may want to draw attention to certain node connections. You may then want to use the title of the snapshot to create a color legend of the nodes and/or node connections that you have highlighted. The coloring of nodes, node connectors, and title lines is made quite easy through the use of special flagging characters in your snapshot files.

To color a title line, insert the flagging characters as the first characters of the line. To highlight a node connection, insert the flagging characters in front of a node on a connected node list, but after the \A arrow delimiter (if an arrow is to be drawn pointing to the node). To highlight a node, insert the flagging characters as the first characters in the first text line of the node. The title line, the connecting line *and label (for graphs)*, or the node, is then highlighted in the format associated with the inserted character, with the flagging character removed in the graphical display. The following table gives the ten available highlighting schemes and their corresponding flagging characters.

Table 2.1 Highlighting schemes associated with flagging characters

Flagging character	Fill Color (for highlighted nodes) Text Color and label color (for connecting lines) Text Color (for title lines)	Text Color (for highlighted nodes)
\X	Black	White
\G	Bright Green	Black
\R	Red	Black
\B	Blue	Yellow
\W	White	Black
\L	Light Blue	Black
\M	Magenta	Black
\Y	Yellow	Black
\#XXXXXX	6-digit hex color code for RGB	White (Black if code is white)

2 Extending the "snapshots" rendered by GAIGS – adding your own structures

Since GAIGS is object orientated, theoretically, it can be expanded to visualize anything. The expansion is broken down into two parts, adding a data structure to GAIGS, and adding an algorithm. Every data structure has StructureType as an ancestor.

```
StructureType
 Md_Array
 LinearList
  Stack
  Queue
  LinkedList                  Each structure contains
  Bar                         a nodelist among other
 BinaryTree                   information
 GeneralTree
 Graph_Network
  Ggraph
  Network
```

StructureType has a nodelist field, so any descendant of it will also have a nodelist field. This field can reference a list of nodes, with the type of node in the list dependent on the data structure. For instance, a stack.nodelist is a list of plain Nodes, while Md_Array.nodelist is a list of LinearNodes. The node hierarchy is also given. You may choose, of course, to ignore this nodelist field, and store your data structure any way you want. This is done by extending the abstract *StructureType*. In the extended type that you define, four methods from *StructureType* should be overridden:

- `public void calcDimsAndStartPts(LinkedList llist, draw d)` – used to establish geometric variables that will determine how to position your graphics when rendered in *drawStructure*. In particular, by calling *super* in your calcDimsAndStartPts, you will have calculated for you *TitleEndy*, which is the y-coordinate of where the snapshot title ends in the (0,1) NDC space in which GAIGS renders a snapshot.
- `void loadStructure (StringTokenizer st, LinkedList llist, draw d)` – Here is where you grab and parse the tokens in one instance of your structure.
- `void drawStructure (LinkedList llist, draw d)` – Here you render one instance of the structure use the rendering functions from the *GKS* class.

- `boolean emptyStruct()` – Here you tell GAIGS how to recognize an "empty structure" for the particular structure you are defining. If an empty structure isn't possible, merely return *false*.

- Finally, in *draw.java* you will need to add a test to recognize an instance of your structure:

```
if ( structType.toUpperCase().compareTo("DEMO_STR") == 0)
    return (new Demo_str());
```

2.1 Example – Defining a new *Demo_ str*

Sample file with three Demo_str snapshots:

```
Demo_str
1               <-- Number of lines per node, here only ''filler''
Hi
There
***\***         <-- End of title
2               <-- Color of four-sided filled polygon
0.3             <-- Next eight lines are the vertices of the polygon
0.7
0.7
0.7
0.7
0.3
0.3
0.3
***^***
Demo_str
1
Number
Two
Snapshot
***\***
4
0.3
0.6
```

```
0.7
0.6
0.7
0.1
0.3
0.1
***^***
Demo_str
1
That's all there is!
***\***
5
0.01
0.7
0.99
0.7
0.99
0.3
0.01
0.3
***^***
```

Implementation of the *Demo_ str* class:

```
package gaigs;
import java.io.*;
import java.awt.*;
import java.util.*;

public class Demo_str extends StructureType {

    private int str_color;

    private double x[], y[];

    public Demo_str () {
        super();
        x = new double[5];
```

```
    y = new double[5];
}

public void calcDimsAndStartPts(LinkedList llist, draw d) {
    super.calcDimsAndStartPts(llist, d);
}

void drawStructure (LinkedList llist, draw d) {

    double xline[], yline[];

    super.drawStructure(llist,d);
    xline = new double [2];
    yline = new double [2];
    yline[0]=TitleEndy;
    yline[1]=yline[0];
    xline[0]= 0;
    xline[1]= 1;
    // The polyline is drawn immediately under the title/caption
    GKS.polyline(2,xline,yline,llist,d);
    GKS.set_fill_int_style(bsSolid,str_color,llist,d);
    GKS.fill_area(5,x,y,llist,d);
    GKS.set_fill_int_style(bsClear,White,llist,d);
    // The final Set_fill_int_style is necessary to insure that later
    // titles are displayed on a white background.
}

boolean emptyStruct() {
    return(false);
}

void loadStructure (StringTokenizer st, LinkedList llist, draw d)
    throws VisualizerLoadException  {

    String tline;

    if (st.hasMoreTokens()) {
        tline = st.nextToken();
```

```
        }
        else
            throw (new VisualizerLoadException
                   ("End of data in Demo_str when expecting color number"));
        str_color = Format.atoi(tline);
        for (int i = 0; i < 4; i++)  {
            if (st.hasMoreTokens()) {
                tline = st.nextToken();
            }
            else
                throw (new VisualizerLoadException
                       ("End of data in Demo_str when expecting x coord"));
            x[i] = Format.atof(tline);
            if (st.hasMoreTokens()) {
                tline = st.nextToken();
            }
            else
                throw (new VisualizerLoadException
                       ("End of data in Demo_str when expecting y coord"));
            y[i] = Format.atof(tline);
        }
        x[4] = x[0];
        y[4] = y[0];
        if (st.hasMoreTokens()) {
            tline = st.nextToken();
        }
        else
            throw (new VisualizerLoadException
                   ("End of data in Demo_str when expecting end of snapshot marker"));

    }

}
```

3 Rendering tools available in the GKS class

```
    // Constant color declarations
```

```
    final static int Black  =  1;
    final static int Blue   =  2;
    final static int Green  =  3;
    final static int Red    =  4;
    final static int Magenta = 5;
    final static int LightBlue=6;
    final static int Yellow=   7;
    final static int White=    8;
    final static int LightGray=9;

    // Constant text justification declarations
    final static int     TA_CENTER =0;
    final static int       TA_LEFT =1;
    final static int      TA_RIGHT =2;
    final static int   TA_BASELINE= 0;
    final static int      TA_BOTTOM=1;
    final static int        TA_TOP =2;

    // Constant fill region types
    final static int      bsSolid = 0;
    final static int      bsClear = 1;
    final static int bsHorizontal = 2;
    final static int   bsVertical = 3;
    final static int  bsFDiagonal = 4;
    final static int  bsBDiagonal = 5;
    final static int      bsCross = 6;
    final static int  bsDiagCross = 7;



class GKS {

    public GKS () {
    }

    // Set the interior style and color for a filled region
    public static void set_fill_int_style(int style, int colr,
                                          LinkedList seginfo, draw d) ;
```

```
// ptsx[0] = ptsx[numpts-1] and ptsy[0] = ptsy[numpts-1]

// Given a closed convex polygon with numpts - 1 vertices given by
// the x and y coordinates in ptsx and ptsy, draw a filled polygon
// in the current color and style
public static void fill_area(int numpts, double ptsx[], double ptsy[],
                             LinkedList seginfo, draw d) ;

// Draw a polygonal line connected the numpts points with x and y
// coordinates in the arrays ptsx and ptsy
public static void polyline (int numpts, double ptsx[], double ptsy[],
                             LinkedList seginfo, draw d) ;


// Set the current color for rendering text
public static void set_textline_color(int colr,
                                      LinkedList seginfo, draw d);

// Set the current alignment for rendering text
public static void set_text_align(int horiz, int vert,
                                  LinkedList seginfo, draw d);

// Set the current text size as specified by height
public static void set_text_height(double height,
                                   LinkedList seginfo, draw d);

// Render the text in str at position (x,y) using the current text
// size, color, and alignment
public static void text(double x,double y, String str,
                        LinkedList seginfo, draw d);

// Set the current width for rendered lines
public static void set_line_width(int thickness,
                                  LinkedList seginfo, draw d);

// Draw an elliptical arc
public static void ellipse
```

```
        (double x, double y, double stangle, double endangle,
         double xradius, double yradius,
         LinkedList seginfo, draw d);

    // Draw an unfilled circle
    public static void circle
        (double x, double y, double radius, LinkedList seginfo, draw d);

    // Draw a filled circle
    public static void circle_fill
        (double x, double y, double radius, LinkedList seginfo, draw d);


}
```

One additional trick – to find the length of a string *tstr* in NDC coordinate space, given a *draw* object *d*, use the following:

```
int temp = d.getGraphics().getFontMetrics(defaultFont).stringWidth(tstr);
double length_of_tstr = ((double) temp / (double) d.getSize().width);
```

4 Adding questions to a script file from a program that is writing the script file

- Instantiate a questionCollection – essentially a Vector of questions with a few additional special operations

```
public class questionCollection {

    // Constructor with the output stream to write questions to
    public questionCollection(PrintWriter out){

    // Add question q to the collection
    public void addQuestion(question q){

    // Write the tag for question at index into the output stream
    public void insertQuestion(int index){

    // Write the text of all questions and answers at the end of the script
    public void writeQuestionsAtEOSF(){
}
```

- At times where your script writing program wants to ask a question, instantiate a *tfQuestion*, *mcQuestion*, or *fibQuestion*, all of which extend from the abstract *question* class:

```
public abstract class question{

    // Constructor
    public question()

    // questionText is a string containing the text for this question
    public void setQuestionText(String questionText){
}
```

- Each specific derived question has a constructor that accepts a string id/tag for the question and a *setAnswer* method use to establish the answer for this question. For example the *fibQuestion* class:

```
public class fibQuestion extends question{

    // Construct the fib question, providing its identifying string
    public fibQuestion(PrintWriter out, String id){

    // Set the answer for this fib question
    // Use \n to separate different answers that are allowed
    public void setAnswer(String answer){
```

- So, the general algorithm to create questions is (see *LinearHashing.java* for complete program containing this algorithm:

```
for each snapshot you create
  If you want a question with this snapshot
     Manufacture text of question and the answer
     fibQuestion quest = new mcQuestion(out, (new Integer(qIndex)).toString());
     qIndex++;  // Increment your question counter
     quest.setQuestionText(string-containing-question);
     quest.setAnswer(the-answer);
     Questions.addQuestion(quest);
     Questions.insertQuestion(qIndex);
  Now write the snapshot that is associated with the question
```

- After all the snapshots have been written, be sure to:

```
Questions.writeQuestionsAtEOSF();
```