July 29, 1994

SRC Research Report 126

The 1993 SRC Algorithm Animation Festival

Marc H. Brown

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Abstract

This report describes the 1993 SRC Algorithm Animation Festival. The festival continues an experiment in developing algorithm animations by non-experts, started the previous year, and described in SRC Research Report #98. This year nineteen researchers at Digital Equipment Corporation's Systems Research Center worked for two weeks on animating algorithms. Most of the participants had little (if any) experience writing programs that involved graphics. This report explains why we organized the festival, and describes the logistics of the festival and the advances in our algorithm animation system. This report presents the complete code for a simple, but non-trivial, animation of first-fit binpacking. Finally, this report contains snapshots from the animations produced during the festival.

1 Background

In SRC Research Report #98, we reported on the 1992 SRC Algorithm Animation Festival [2], held during the summer of ’92 at Digital Equipment Corporation’s Systems Research Center (SRC). The primary goal of that animation festival was to promote the Modula-3 programming language, by developing a comprehensive set of interactive animations of fundamental algorithms to accompany a university-level data structures or algorithms course.

It was not our goal (or expectation) that professors would change their algorithms or data structures courses to use Modula-3, just because there existed some potentially useful courseware that was implemented in Modula-3. Rather, we hoped that professors would use the systems, and, at some point, look at the source code—perhaps to animate their own algorithm, to add a new view, or perhaps to simply modify one of the algorithms or views that we provided.

Although we did not achieve our goal of building a comprehensive set of animations for a complete course, we did develop quite a few intriguing animations of fundamental algorithms. The algorithms included string searching, priority queues, parsing, hashing, network flow, convex hulls, closest points, graph traversals, balanced trees, binpacking, and of course, sorting. The animations, along with the Zeus algorithm animation system [1], have been ftp’d from `gatekeeper.dec.com` by over 300 sites. About a dozen of those sites have used the software for teaching. At the University of Illinois, in fact, students in Prof. Sam Kamins’ compiler course were given the option of implementing an animation of a part of a compiler as their final project, rather than the usual “build a toy compiler for a toy language.” At Dortmund University, Profs. Anulf Mester and Peter Herrmann led a year-long project with eleven students developing animations of distributed algorithms and communication protocols [9].

More important than the breadth of our algorithms coverage was the discovery that the animation festival had a number of intangible, positive results within our research laboratory: SRC researchers had an opportunity to work closely with colleagues with very different technical backgrounds and interests, and many of the participants developed new appreciation for and skill in using graphics and animation as communication media. As a result, we decided to offer a second algorithm animation festival during the summer of ’93. This report describes the ’93 animation festival.

The remainder of this report is organized as follows: The next section describes the logistics of the animation festival, followed by observations about the participants and the projects. Then, we describe the advances we made to our algorithm animation system, and show the complete code for an animation of first-fit binpacking. Finally, we offer some concluding thoughts. The figures at the end of this report are screen dumps from the animations produced by the participants during the animation festival. (The animation from one of the groups is not included, pending a patent application.) Because a static image cannot do justice to an interactive animation, each figure contains a fairly detailed caption. A videotape showing some of the animations will be available in the near future as a SRC research report.

2 The ’93 Animation Festival

This section describes the structure of the festival, and presents an overview of the software that was used.

The structure of the ’93 festival followed that of the ’92 festival: the first week was devoted to teaching the participants about algorithm animation, and the second week to implementing animations. The participants demonstrated their work to the rest of SRC during the following weeks.

Most participants learned all that they needed to know about the mechanics of using Zeus for preparing an algorithm animation, and about using the Zeus control panel, from the 2-hour tutorial on Monday morning. The afternoon of the first day was used to get participants not familiar with the SRC Modula-3 programming environment up-to-speed. There was an hour-long lecture about GraphVBT, a high-level animation package [7]; an hour-long lecture about FormsVBT, a user-interface toolkit; and an hour-long introduction to Modula-3. The user-interface toolkit is needed for developing control panels for specifying input to an algorithm.

There were two one-hour lectures on Tuesday afternoon: developing code views and data views in Zeus, and using GEF [8]. GEF is an interpreted, Lisp-like, embedded language. Because GEF is interpreted, turnaround is fast. Because GEF is based on GraphVBT, animation operations and graphical elements are easy to create and manipulate.

There were also two one-hour lectures on Wednesday afternoon. The first lecture was on the various advanced input styles that Zeus supports, and the second lecture was on Obliq, an interpreted, embedable language [5].

The final lectures, on Thursday morning, consisted of an hour devoted to techniques for creating effective algorithm animations, followed by a “culture” hour devoted to viewing a collection of classic algorithm animations.

There were two fundamental changes from the previous festival: The first change was that each lecture was followed by exercises reinforcing the material. The exercises were useful in ensuring that the participants learned about *all* the tools, not just about the subset they *thought* they would need. The second change was that participants paired up immediately, rather than waiting until the second week when they started working on an animation. All teams had a non-trivial animation of binpacking by lunch time on Tuesday.

During the second week, participants and instructors met at the end of each day for some R&R, that is "rushes and refreshments." This was an opportunity for participants to demonstrate what they had done that day and to get feedback from other participants and the instructors.

Of the 19 participants, three were Ph.D. students who were at SRC as part of its Summer Research Internship program, one was a member of Digital's Western Research Laboratory (WRL) in Palo Alto, one was a faculty member at Staten Island College, and the others were fulltime members of the research staff at SRC with a variety of research interests.

Last year, we provided the participants with a list of "most wanted algorithms." Because we were not trying to cover any specific algorithms this year, we offered neither direction nor constraints in choosing an algorithm to animate.

Two groups animated algorithms they were currently working on (Figs. 2and 7). One group had always wanted to develop a particular algorithm, and the festival gave them the perfect excuse (Fig. 5). Two groups animated algorithms that they had developed within the previous couple of years in other contexts (Fig. 6 and 8), two other groups worked on algorithms that members found intriguing (Figs. 4 and 9). Finally, one group did an educational simulation (Fig. 1), and another did a computer game (Fig. 3).

All of the participants, except one, used the Zeus algorithm animation system. The exception (Fig. 7), a member of the research staff at WRL, started with a working implementation of a label-placement algorithm in Scheme, and added many Zeus-like features to the program in a program-specific way. (Furthermore, the label-placement algorithm [6] was developed by a researcher at Digital's Cambridge Research Laboratory.)

The essence of animating an algorithm in Zeus is to separate the *algorithm* from each *view* and to create a narrow interface, specific to each algorithm, between them. More specifically, an algorithm is annotated with procedure calls that identify the fundamental operations that are to be displayed. An annotation, called an *interesting event*, has parameters that identify program data. A view is a subclass of a window, with additional methods that are invoked by Zeus whenever an interesting event happens in the algorithm. Each view is responsible for updating its graphical display appropriately, based on the interesting events. Views can also propagate information from the user back to the algorithm.

It is important to realize that the mechanics of producing an algorithm animation is the easiest part of animating an algorithm. A harder (and more enjoyable) part is deciding on effective and informative visualizations. This is necessarily an iterative process, and one that requires constant feedback from many different people. (An even harder, and yet even more enjoyable, part is to incorporate the animations into a lecture; we did not address this problem in the animation festival, however.)

In order to address the reality that designing views is an iterative process, we provided two embedded interpreted languages, GEF and Obliq, for writing views. Two groups used GEF and six groups used Obliq. Only one group used Modula-3, due to performance limitations of the interpreted languages. (Since the festival, the GEF system has been retired, primarily because Obliq subsumes the functionality of GEF, and offers many more features.) The reaction to using GEF and Obliq for writing views was unanimous enthusiasm.

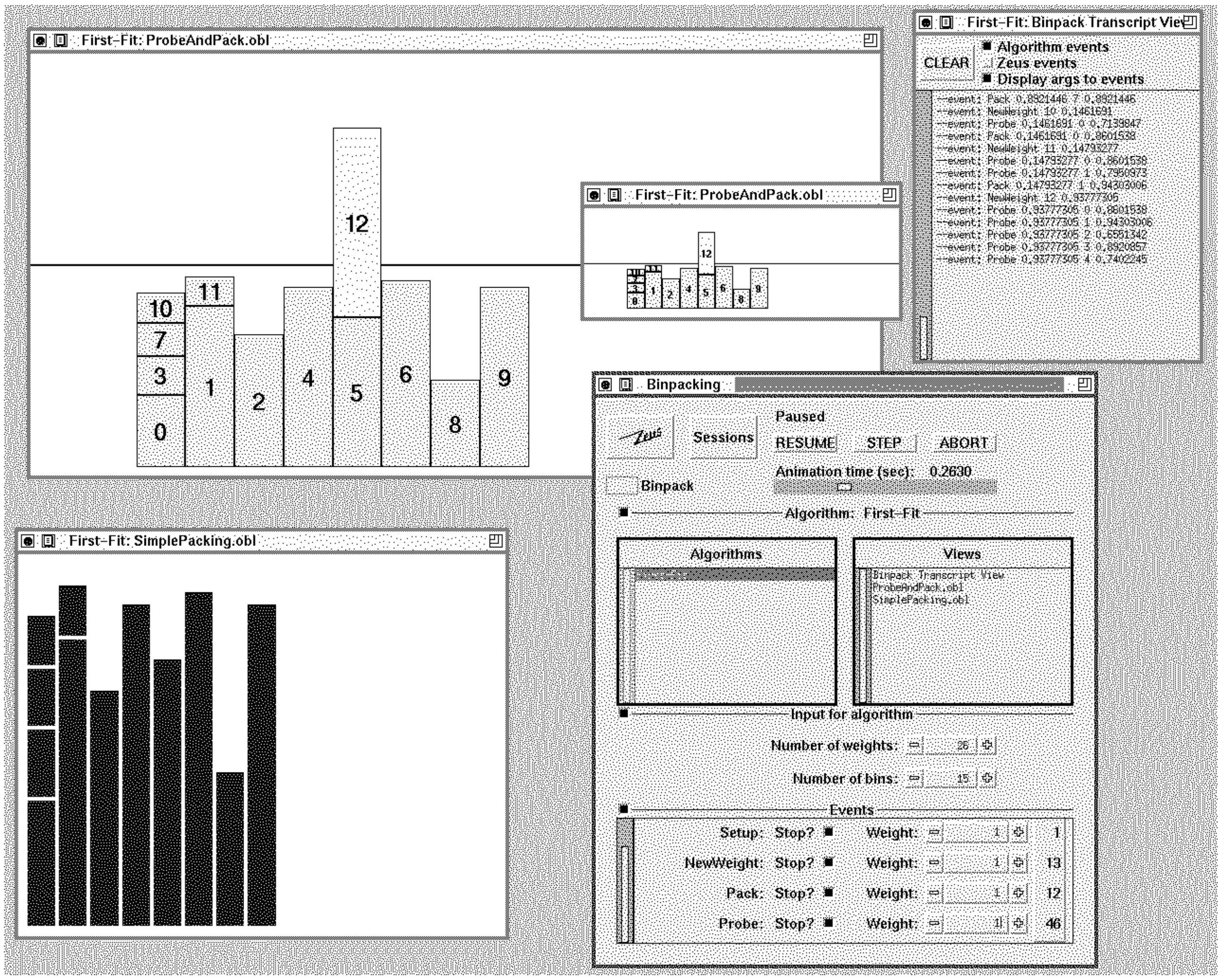
3 An Example Animation

This section contains the code for an animation of first-fit binpacking, as shown on the facing page. There are seven source files, written in five different languages. Although some of the languages will be unfamiliar to you, nonetheless, they are hopefully understandable without much difficulty. The purpose in presenting the code here is to give you an understanding of what's involved in preparing an animation for Zeus; it is not meant to be a tutorial on preparing Zeus animations. A tutorial will appear as a SRC research report in the near future.

The file `m3makefile`, written in quake, contains a declaration of the various pieces of the application. Here it is:

```
import         (zeus)
zume           (Binpack)
oblume         (Binpack, SimplePacking)
oblume         (Binpack, ProbeAndPack)
module         (AlgFF)
resource       (binpackinput.fv)
implementation (Main)
bundle         (BinpackBundle)
program        (binpack)
```

You just need to type "`m3build`" to the shell in order to rebuild the entire application. The m3build command reads in the contents of the `m3makefile`, runs various filters and compilers, and eventually creates an application named `binpack`.


First-Fit: ProbeAndPack.obl
12
10
11
7
3
0
1
2
4
5
6
8
9
First-Fit: ProbeAndPack.obl
First-Fit: SimplePacking.obl
First-Fit: Binpack Transcript View
CLEAR
Binpacking
Zeus
Sessions
Binpack
RESUME
STEP
ABORT
Algorithms
Views

■ Algorithm events
□ Zeus events
■ Display args to events

```
--event: Pack 0.8921446 7 0.8921446
--event: NewWeight 10 0.1461691
--event: Probe 0.1461691 0 0.7139847
--event: Pack 0.1461691 0 0.8601538
--event: NewWeight 11 0.14793277
--event: Probe 0.14793277 0 0.8601538
--event: Probe 0.14793277 1 0.7950973
--event: Pack 0.14793277 1 0.94303006
--event: NewWeight 12 0.93777305
--event: Probe 0.93777305 0 0.8601538
--event: Probe 0.93777305 1 0.94303006
--event: Probe 0.93777305 2 0.6551342
--event: Probe 0.93777305 3 0.8920857
--event: Probe 0.93777305 4 0.7402245
```

Paused

Animation time (sec): 0.2630

Algorithm: First-Fit

Views:
Binpack Transcript View
ProbeAndPack.obl
SimplePacking.obl

Input for algorithm

Number of weights: 26

Number of bins: 15

Events

Setup:	Stop?	■	Weight:	1	1
NewWeight:	Stop?	■	Weight:	1	13
Pack:	Stop?	■	Weight:	1	12
Probe:	Stop?	■	Weight:	1	46

The file `Binpack.evt` defines the interesting events for binpacking algorithms in Zeus's Event Description Language:

```
OUTPUT Setup (nBins, nWts: INTEGER[Fmt.Int]);
(* Call once at the beginning to identify
   for views how many diff weights there
   will be for packing, and the maximum
   number of bins that will be used. The
   bins are numbered starting at 0, and each
   bin can hold at most 1.0 unit of weight. *)

OUTPUT NewWeight (id: INTEGER[Fmt.Int];
                  wt: REAL[Fmt.Real]);
(* Got a new weight to try packing into a bin.
   Id's are between 0 and nWts-1. It is not
   guaranteed that the id's will be sequential. *)

OUTPUT Pack (wt: REAL[Fmt.Real];
             bin: INTEGER[Fmt.Int];
             total: REAL[Fmt.Real]);
(* Pack the current weight into the specified bin,
   raising the amount stored in that bin to the
   specified total. *)

OUTPUT Probe (wt: REAL[Fmt.Real];
              bin: INTEGER[Fmt.Int];
              total: REAL[Fmt.Real]);
(* Check whether the current weight can fit into
   the specified bin, which has the specified total.
   The bin is between 0 and nBins. *)

(*
**   The following regular expression defines the
**   output event stream generated by a Binpack alg:
**
**           Setup (NewWeight Probe+ Pack)*
**
*)
```

Zeus processes this file and produces a variety of objects and files that mediate the communication from an algorithm to the views. The derived files are automatically compiled and linked into the application. There are two derived files that we'll make use of later: `BinpackIE` and `BinpackAlgClass`.

The file `AlgFF.m3` contains the Modula-3 source code for a first-fit binpacking algorithm, annotated with the interesting events

```
MODULE AlgFF;

IMPORT Algorithm, FormsVBT, Random, Thread, VBT, ZeusPanel;
IMPORT BinpackAlgClass, BinpackIE;

PROCEDURE New (): Algorithm.T =
  BEGIN RETURN NEW(BinpackAlgClass.T,
     run := Run,
     data := ZeusPanel.NewForm("binpackinput.fv")).init()
  END New;

PROCEDURE Run (alg: Algorithm.T) RAISES {Thread.Alerted} =
  <* FATAL FormsVBT.Error, FormsVBT.Unimplemented *>
  VAR
    B: INTEGER;                 (* number of bins *)
    N: INTEGER;                 (* number of weights to pack *)
    bin: INTEGER;               (* index into array of bins *)
    amt: REAL;                  (* current weight *)
    totals: REF ARRAY OF REAL;  (* b’th bin has totals[b] *)
    rand := NEW(Random.Default).init(); (* random number generator *)
  BEGIN
    LOCK VBT.mu DO
      N := FormsVBT.GetInteger(alg.data, "N");
      B := FormsVBT.GetInteger(alg.data, "B");
    END;
    BinpackIE.Setup(alg, B, N);
    totals := NEW(REF ARRAY OF REAL, B);
    FOR b := 0 TO B - 1 DO totals[b] := 0.0 END;
    FOR w := 0 TO N - 1 DO
      amt := rand.real();
      BinpackIE.NewWeight(alg, w, amt);
      bin := 0;
      BinpackIE.Probe (alg, amt, 0, totals[0]);
      WHILE totals[bin] + amt > 1.0 DO
        INC(bin);
        IF bin = B THEN RETURN END;
        BinpackIE.Probe (alg, amt, bin, totals[bin])
      END;
      totals[bin] := totals[bin] + amt;
      BinpackIE.Pack(alg, amt, bin, totals[bin])
    END
  END Run;

BEGIN
  ZeusPanel.RegisterAlg(New, "First-Fit", "Binpack");
END AlgFF.
```

The algorithm uses procedures defined in `BinpackIE` for inserting event annotations. The algorithm uses the object defined in `BinpackAlgClass` as a supertype of the first-fit binpacking algorithm.

The algorithm makes reference to the file `binpackinput.fv`, which contains a description of a graphical user-interface with two numeric widgets:

```
(VBox
  (HBox
    (Text RightAlign "Number of weights: ")
    (Numeric (Min 1) (Max 1000) %N =26))
  (Glue 10)
  (HBox
    (Text RightAlign "Number of bins: ")
    (Numeric (Min 1) (Max 100) %B =15)))
```

The user-interface description is written in FormsVBT. It is visible in the middle of the Zeus control panel, in the lower-right of the screen dump.

The file `SimplePacking.obl` contains a view, written in Obliq:

```
let view = {

  graphvbt => graph_new(),

  Setup => meth (self, nBins, nWts)
    graph_setWorld(self.graphvbt, 0.0, real_float(nBins), 1.0, 0.0);
    graph_setMargin(self.graphvbt, 2.0);
    end,

  Pack => meth (self, wt, bin, total)
    let xpos = 0.5 + real_float(bin);
    let ypos = total-(wt/2.0);
    let v = graph_newVertex(self.graphvbt);
    graph_setVertexSize(v, 1.0, wt);
    graph_moveVertex(v, xpos, ypos, false);
    graph_setVertexBorder(v, 0.01);
    graph_setVertexColor(v, color_named("Blue"));
    graph_redisplay(self.graphvbt);
    end,
};
```

This view is very simple. It shows how the weights have been packed into the bins. There is no animation per se; each time the algorithms determine the bin into which a weight should go, a new blue rectangle (corresponding to the weight) is drawn.

The file `ProbeAndPack.obl` is a more polished view:

```
let view = {

  graphvbt => graph_new(),
  font => ok,
  currVertex => ok,
  packedColor => color_named("Pink"),
  initialColor => color_named("VeryLightGray"),

  Setup => meth (self, nBins, nWts)
    graph_setWorld(self.graphvbt,
      -2.0, float(nBins), 2.0, 0.0);
    graph_setMargin(self.graphvbt, 2.0);
    self.font := graph_newFont (self.graphvbt,
         "Helvetica", 0.5, "Roman", "Bold", "*");
    let v0 = graph_newVertex(self.graphvbt);
    graph_setVertexSize(v0, 0.0, 0.0);
    graph_moveVertex(v0, -10.0, 1.0, false);
    let v1 = graph_newVertex(self.graphvbt);
    graph_setVertexSize(v1, 0.0, 0.0);
    graph_moveVertex(v1, float(nBins)+10.0, 1.0, false);
    let e = graph_newEdge(v0, v1);
    graph_setEdgeWidth(e, 0.01);
    graph_redisplay (self.graphvbt);
    end,

  NewWeight => meth (self, id, wt)
    let v = graph_newVertex (self.graphvbt);
    graph_setVertexSize(v, 1.0, wt);
    graph_moveVertex(v, -1.0, 1.0, false);
    graph_setVertexFont(v, self.font);
    graph_setVertexBorder(v, 0.01);
    graph_setVertexLabel(v, fmt_int(id));
    graph_setVertexColor(v, self.initialColor);
    graph_setVertexLabelColor(v, color_rgb(0.0, 0.0, 0.0));
    graph_redisplay(self.graphvbt);
    self.currVertex := v;
    end,

  Probe => meth (self, wt, bin, total)
    let xpos = 0.5 + float(bin);
    let ypos = float(total) + (wt / 2.0);
    graph_moveVertex(self.currVertex, xpos, ypos, true);
    zeus_animate(self.graphvbt, 0.0, 1.0);
    end,

  Pack => meth (self, wt, bin, total)
    graph_setVertexColor(self.currVertex, self.packedColor);
    graph_redisplay(self.graphvbt);
    end,
};
```

When the algorithm is given a new weight to process, a rectangle corresponding to the weight is drawn at the left in gray. As the algorithm probes the non-empty bins, the weight smoothly slides to the bin to see if there is room. Finally, when the algorithm decides which bin to place the weight in, the color of the rectangle changes. There are two instances of this view in the screen dump. Notice how the view scales itself to fit into the window.

The most important thing to realize about the views is that the code is short and high-level. Moreover, making a one-line change in the view, for instance, changing the width of the border, is literally a 5-second turnaround.

Finally, the mainline for the application is in the Modula-3 file `Main.m3`:

```
MODULE EXPORTS Main;
IMPORT ZeusPanel, Rsrc, BinpackBundle;
BEGIN
  ZeusPanel.Interact(
    "Binpacking",
    Rsrc.BuildPath("$BINPACKPATH", BinpackBundle.Get()));
END Main.
```

4 Conclusions

The participants in the animation festival seemed to have genuinely enjoyed the experience. Our current plans are to hold a '94 animation festival over one week (rather than two), and to expand the target domain from algorithms to include systems as well as algorithms. In addition, we have recently completed a sophisticated animation package for specifying 3-D animations, so we anticipate that some participants will explore using 3-D graphics.

5 System Availability

The Zeus system and many of the animations developed during the '92 and '93 animation festivals are available via anonymous ftp from `gatekeeper.dec.com`. They are located in the directory `pub/DEC/Modula-3/release`.

You can find out information about Modula-3 in the Usenet news group `comp.lang.modula3`. If you do not have access to Usenet, you can be added to a relay mailing list by sending a message to `m3-request@src.dec.com`.

6 Acknowledgments

The instructors in the '93 Animation Festival were Steve Glassman, John DeTreville, and Marc Brown.

The participants were Yuan Yu, Sam Weber, Jim Saxe, Ricardo Sanchez, Tom Rodeheffer, Sharon Perl, Greg Nelson, Eric Muller, Tim Mann, Mark Manasse, Roberta Klibaner, Jim Horning, Allan Heydon, Stephen Harrison, Hania Gajewska, Dave Detlefs, Joao Comba, Luca Cardelli, and Joel Bartlett.

Hania, Allan, Greg, and Jim Saxe deserve special recognition for having participated in both the '92 and '93 Animation Festivals.

Thanks to Allan, Dave, Luca, Jim Saxe, Joel, Mark, Tim, and Sam for providing the figure captions.

References

[1] Marc H. Brown, Zeus: A System for Algorithm Animation and Multi-View Editing, In *Proc. 1991 IEEE Workshop on Visual Languages*, pages 4–9, October 1991.

[2] Marc H. Brown, The 1992 SRC Algorithm Animation Festival, In *Proc. 1993 IEEE Symposium on Visual Languages*, pages 116–123, August 1993. Also appears as Research Report #98, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (March 1993).

[3] Marc H. Brown and Robert Sedgewick, A System for Algorithm Animation, *Computer Graphics*, 18(3):177–186, July 1984.

[4] M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Research Report #124, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (May 1994).

[5] Luca Cardelli. Obliq: A language with distributed scope. Research Report #122, Digital Equipment Corp., Systems Research Center, Palo Alto, CA (April 1994).

[6] Jon Christensen, Joe Marks, and Stuart Shieber, Placing Text Labels on Maps and Diagrams, In *Graphic Gems*, Vol. 4, Academic Press, 1994.

[7] John D. DeTreville, The GraphVBT Interface for Programming Algorithm Animations, In *Proc. 1993 IEEE Symposium on Visual Languages*, pages 26–31, August 1993.

[8] Steven C. Glassman, A Turbo Environment for Animating Algorithms, In *Proc. 1993 IEEE Symposium on Visual Languages*, pages 32–36, August 1993.

[9] Arnulf Mester. ZADA: Zeus-based animations of distributed algorithms and communication protocols.
http://ls4-www.informatik.uni-dortmund.de/RVS/zada.html,
May 31, 1994.

Fig. 1: An Animated Biology Lesson. This snapshot shows two different biology lessons. Each lesson is implemented as a separate Zeus animation.

The window in the upper-right is a lesson on dominant and recessive genes. Initially, there are four fruit flies, two male and two female. Each fly is labelled with its genotype, and shows its phenotype (how these genes are expressed) visually. We are interested in four characteristics of fruit flies. Each characteristic is shown visually in the drawing of the fly, and encoded in the genotype label, with capital letters for dominant genes and small letters for recessive genes. The characteristics are eye color ("R" for red and "w" for white); eye shape ("C" for circle and "b" for barred); wing shape ("N" for normal and "v" for vestigial); and sex ("Y" for male and "x" for female). Sex is shown visually by a blue or pink body color.

A mating pair is chosen randomly and slides to the center of the screen; these then produce four "eggs" whose genotype is chosen from the genotypes of the parents by the same random selection that sexual reproduction uses. The eggs "hatch" and each new fly's phenotype is determined by its genotype. The bar graphs are histograms of which characteristics are expressed in the offspring. They are essentially the kind of data from which Mendel deduced the existence of dominant and recessive genes.

The window in the lower-left is from a lesson on the details of sexual reproduction. The animation shows how mitosis takes a diloid cell with a full complement of genetic material, and divides it into two haploid cells, each with one (randomly selected) member of each of the chromosome pairs in the nucleus. This cell duplicates each chromosome and divides again, producing real sex cells, sperms or ovae.

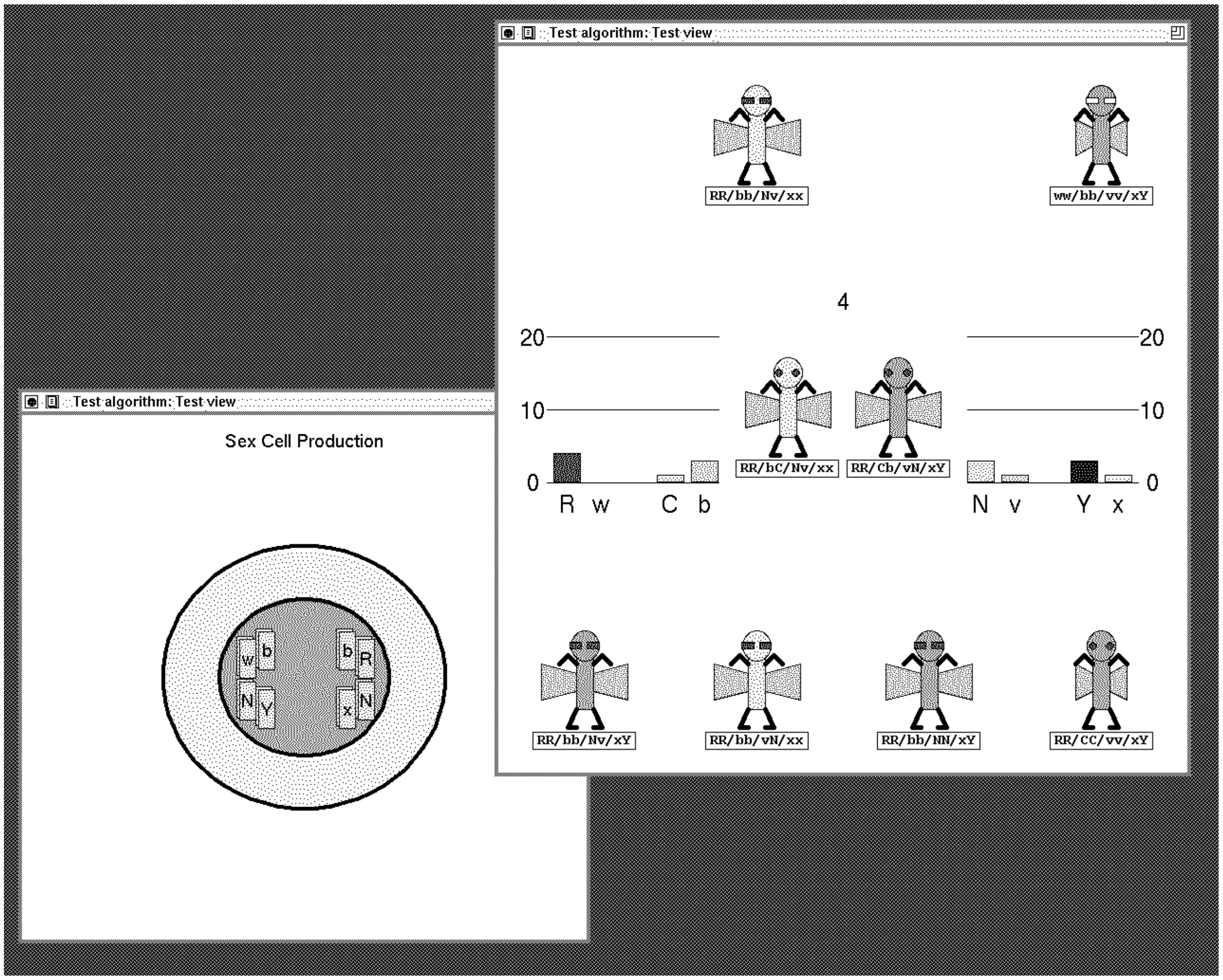
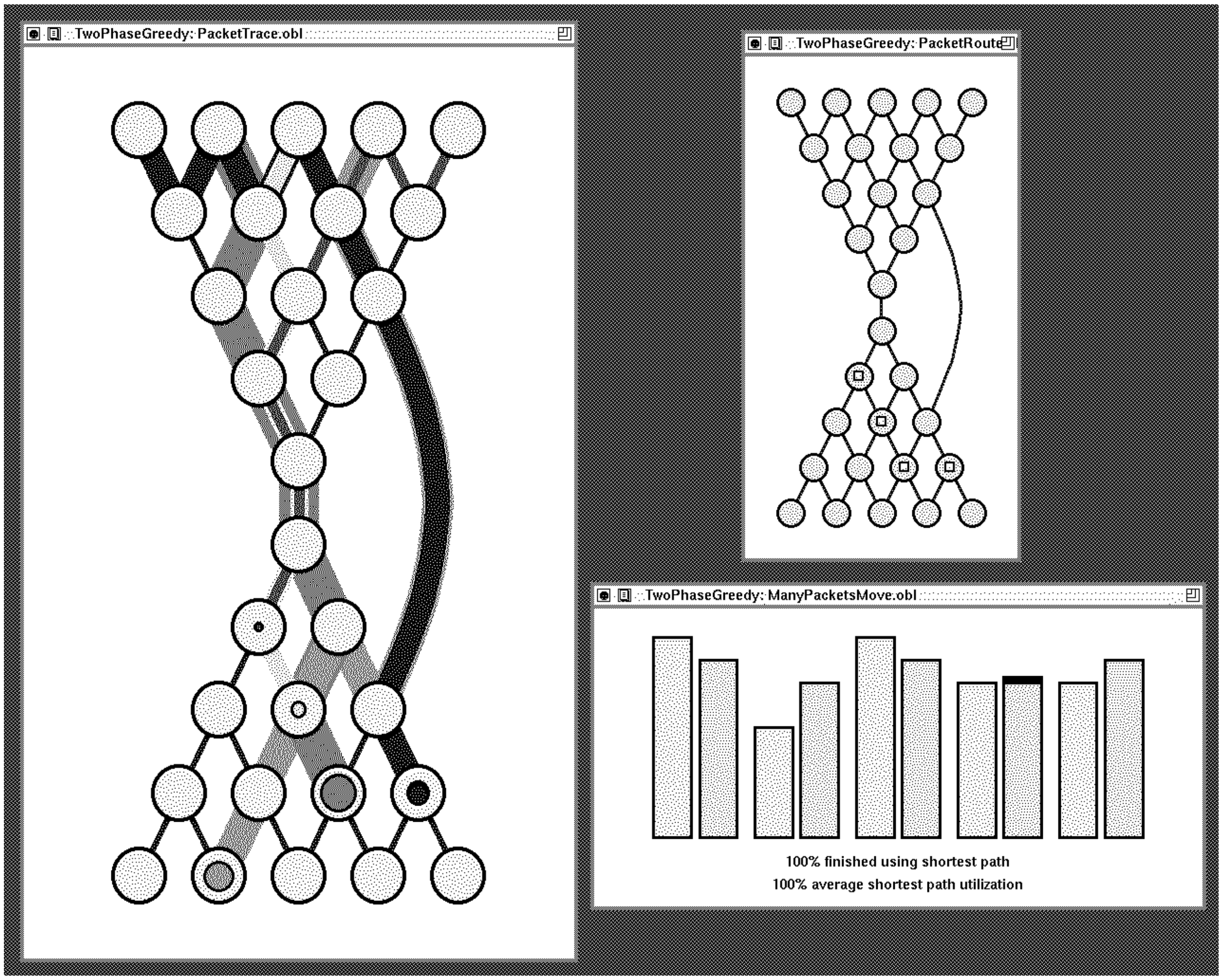

Test algorithm: Test view
Test algorithm: Test view
Sex Cell Production
w
b
N
Y
b
R
x
N
RR/bb/Nv/xx
ww/bb/vv/xY
4
20
10
0
R w C b
RR/bC/Nv/xx
RR/Cb/vN/xY
N v Y x
RR/bb/Nv/xY
RR/bb/vN/xx
RR/bb/NN/xY
RR/CC/vv/xY

Fig. 2: A Packet Routing Simulator. The screen dump here is from an animation of a *packet routing* simulation. The animation is designed to illustrate the performance of various packet routing heuristics on store-and-forward networks of arbitrary topology.

The *PacketRoute.obl* view on the upper-right shows a circle for each node of the network, a line for each inter-node connection in the network, and small squares corresponding to the current locations of the packets. At each routing step, the squares move along some link from their current node to a neighboring node.

The *PacketTrace.obl* view on the left shows the same network, but in this view, each packet leaves behind a colored "trail" as the animation progresses so you can see the complete path taken by each packet. In this view, the packets are represented by colored circles of varying sizes so that the trails of several packets are visible even if they cross the same link during the routing (see, for example, the link at the bottleneck of the hourglass).

Finally, the *ManyPacketsMove.obl* view in the lower right shows statistics about the performance of the routing heuristics. For each of the 5 packets in this simulation, a pair of columns is displayed. The height of the left column in the pair corresponds to the length of the shortest path that packet could travel to get from its specified source to its destination. The height of the right column in the pair corresponds to the number of links the packet has traveled so far. The right column of the fourth packet has a small "cap" to indicate that the packet has reached its destination. Because the two columns for this packet are the same height, we can see that the routing heuristics directed this packet along one of the shortest possible paths.


TwoPhaseGreedy: PacketTrace.obl
TwoPhaseGreedy: PacketRoute
TwoPhaseGreedy: ManyPacketsMove.obl

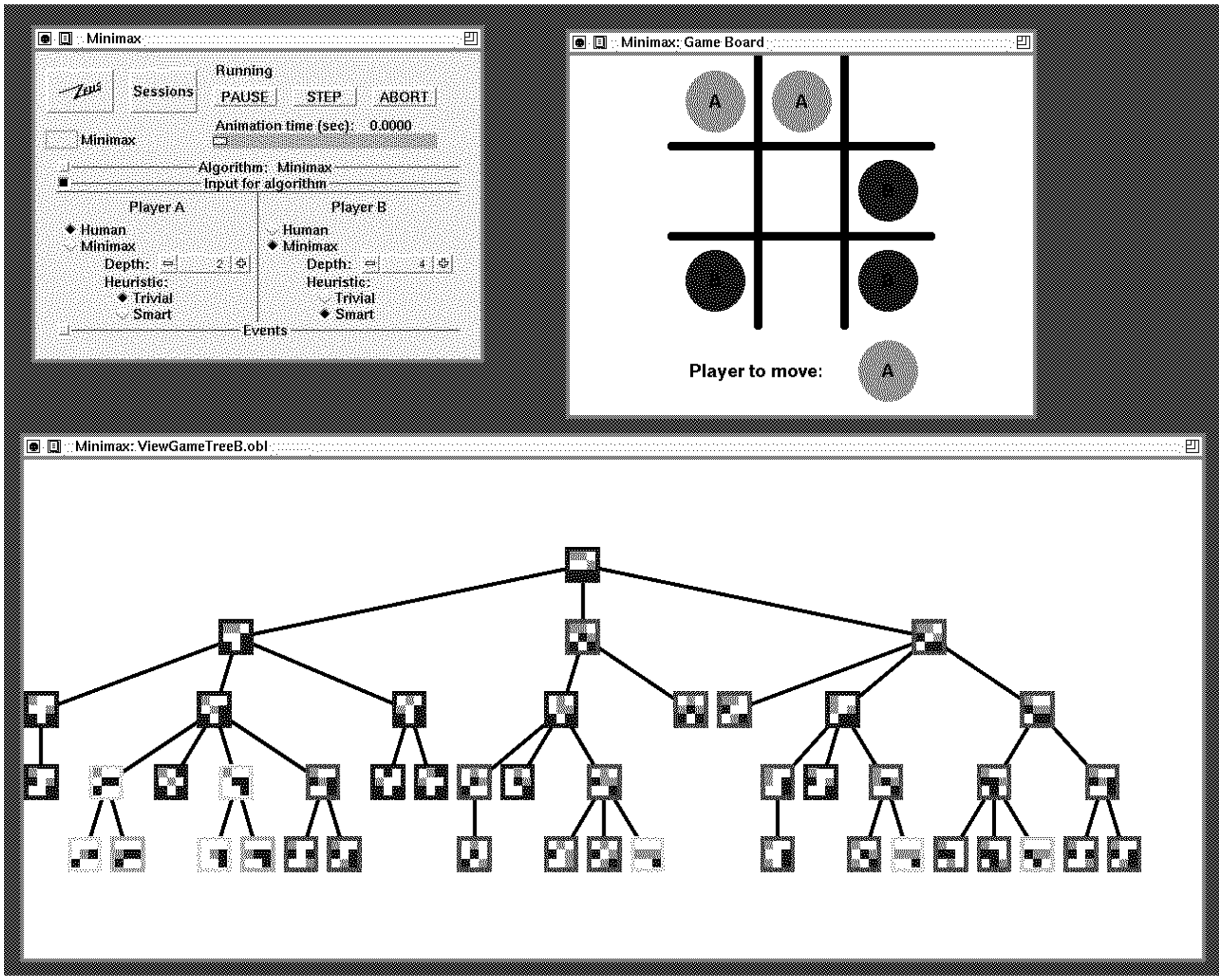
100% finished using shortest path
100% average shortest path utilization

Fig. 3: Hexpawn: Man versus Machine. In this screen dump, we see a hexpawn game being played between a human (as player A), and the minimax algorithm (as player B). The computer has just moved, and the resulting board position is shown in the *Game Board* view in the upper-right.

The *ViewGameTreeB.obl* view at the bottom shows the minimax computation that led to the last move by B. The root of the game tree shows the position that the board was in before B's move. The game tree was computed to a depth of four, and then the nodes were evaluated. Each node was given a value ranging from $-\infty$ to ∞, where $-\infty$ indicates a sure loss for B, and ∞ a sure win. The value assigned to each node is indicated by the color of its border: red indicates $-\infty$, yellow is 0, blue is ∞, and other values are indicated by colors between these on the spectrum.

When this screen dump was generated, there were three possible moves that B could have made from the last board position, as shown by the fact that the root of the game tree has three children. The first child was evaluated as ∞, while the other children had value $-\infty$. Therefore, the minimax algorithm chose to make the first of these moves.

The standard Zeus control panel is in the upper-left window. Notice the various parameters that the user can set for configuring how the animation will run.



Minimax

Sessions

Running

PAUSE STEP ABORT

Animation time (sec): 0.0000

Minimax

Algorithm: Minimax

Input for algorithm

Player A	Player B
[✓ Human]	Human
Minimax	[✓ Minimax]
Depth: 2	Depth: 4
Heuristic:	Heuristic:
[✓ Trivial]	Trivial
Smart	[✓ Smart]

Events

Minimax: Game Board

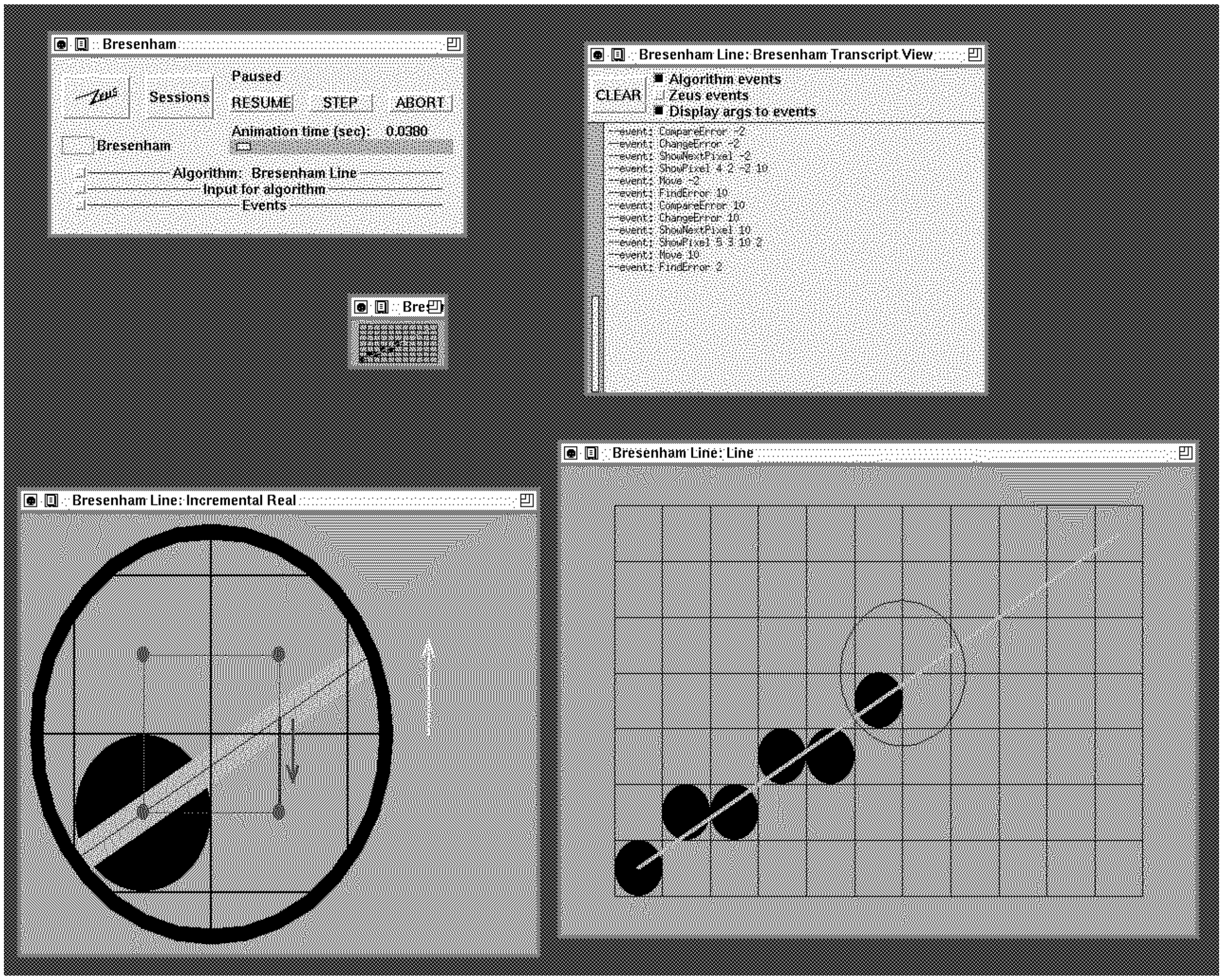
Player to move: A

Minimax: ViewGameTreeB.obl

Fig. 4: Bresenham's Line-Drawing Algorithm. This animation shows Bresenham's line-drawing algorithm, used to display a line on a raster-display.

The *Line* view in the lower-right shows the ideal line (in green) and very large pixels. The *Incremental Real* view in the bottom-left shows the part of the line currently being processed, and with more detail. The red and yellow arrows indicate the local variables that the algorithm maintains in order to determine where the next pixel should be drawn—to the east or northeast. The small view in the middle shows the rasterized line with smaller pixels.

The window in the upper-left is the Zeus control panel, and the window in the upper-right is the *Transcript* view, provided automatically by Zeus. The *Transcript* view prints each event that goes from the algorithm to the views.



```
--event: CompareError -2
--event: ChangeError -2
--event: ShowNextPixel -2
--event: ShowPixel 4 2 -2 10
--event: Move -2
--event: FindError 10
--event: CompareError 10
--event: ChangeError 10
--event: ShowNextPixel 10
--event: ShowPixel 5 3 10 2
--event: Move 10
--event: FindError 2
```

Fig. 5: The k-Shortest Paths Algorithm. The k-shortest paths algorithm finds the shortest path (and the 2nd shortest, the 3rd shortest, . . . , and the kth shortest) from a starting vertex to all other vertices, in a directed graph with all positive edge weights. In this example, $k = 3$ and the starting vertex is a.

The example graph contains five vertices; the edges of the graph are shown in heavy white lines, and the weight of each edge is the Euclidean distance between the edge's endpoints. The snapshot shows the 3 shortest paths from vertex a to all other vertices. The concentric rings at each vertex represent the termini of the 3 shortest paths to that vertex; the shortest path terminates at the innermost ring, the second-shortest path terminates at the middle ring, and the third-shortest path terminates at the outer ring. The source of an edge indicates, both by color and by starting position, each of the k-shortest paths.

Perhaps the easiest way to read the picture is from the end of the path back to the source. For example, to find the second shortest path from a to b, we first note that the second shortest path into b comes from the middle ring of c, that is, the second shortest path into c (the green arc). To get to the middle ring of c we follow the center ring from d (the red arc). And to get to the center ring of d, we just follow the edge from a (the red arc). Thus, the path is a-d-c-b. Similarly, the third shortest path from a to b uses the outer (blue) arc from c. To get there, we follow the inner (red) arc from e, which came from d, which came from a. Thus, the whole path is a-d-e-c-b.

The lighter colored rings and edges depict computations still in progress; in the picture above, we are in the process of settling on the second-shortest path to e.

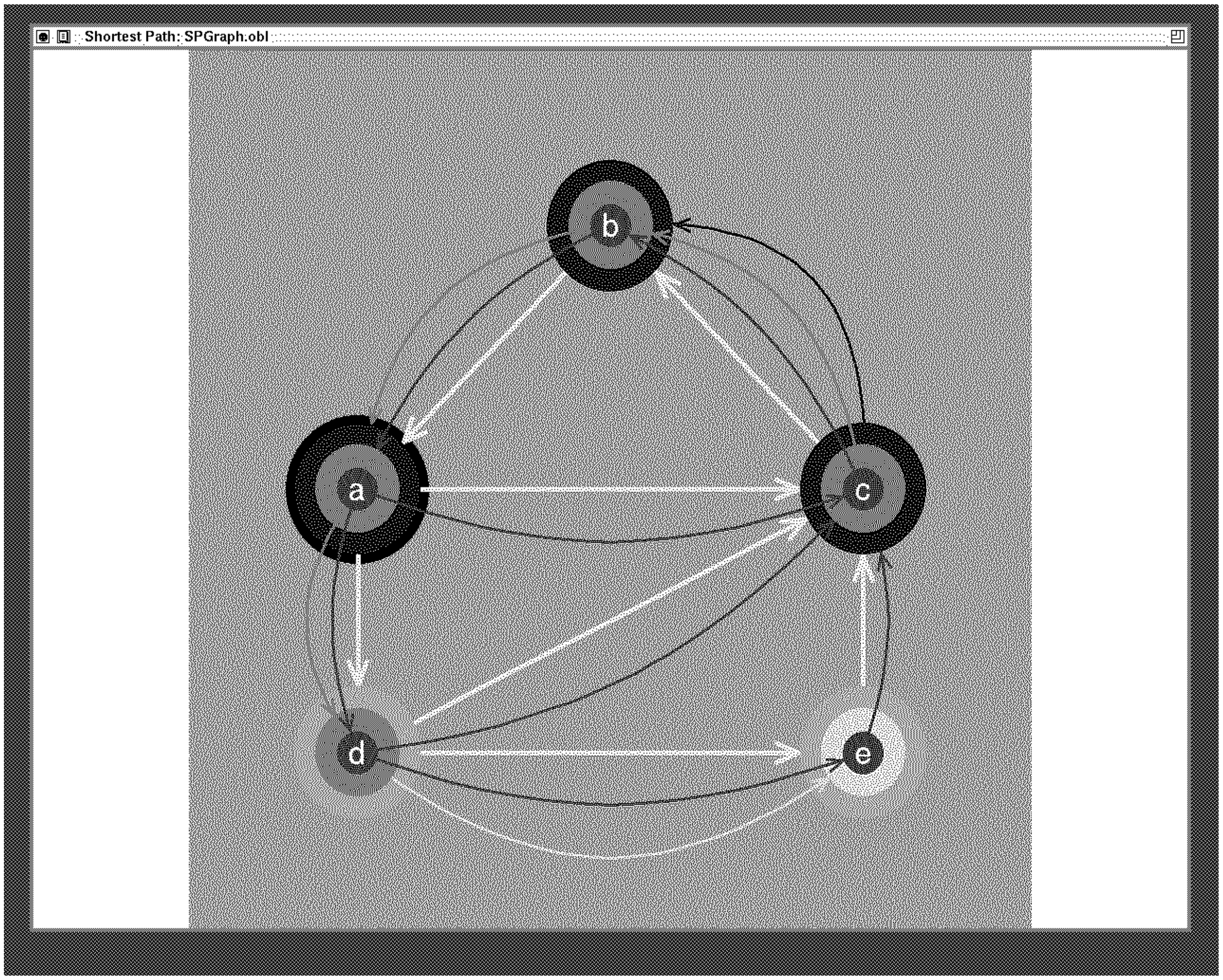
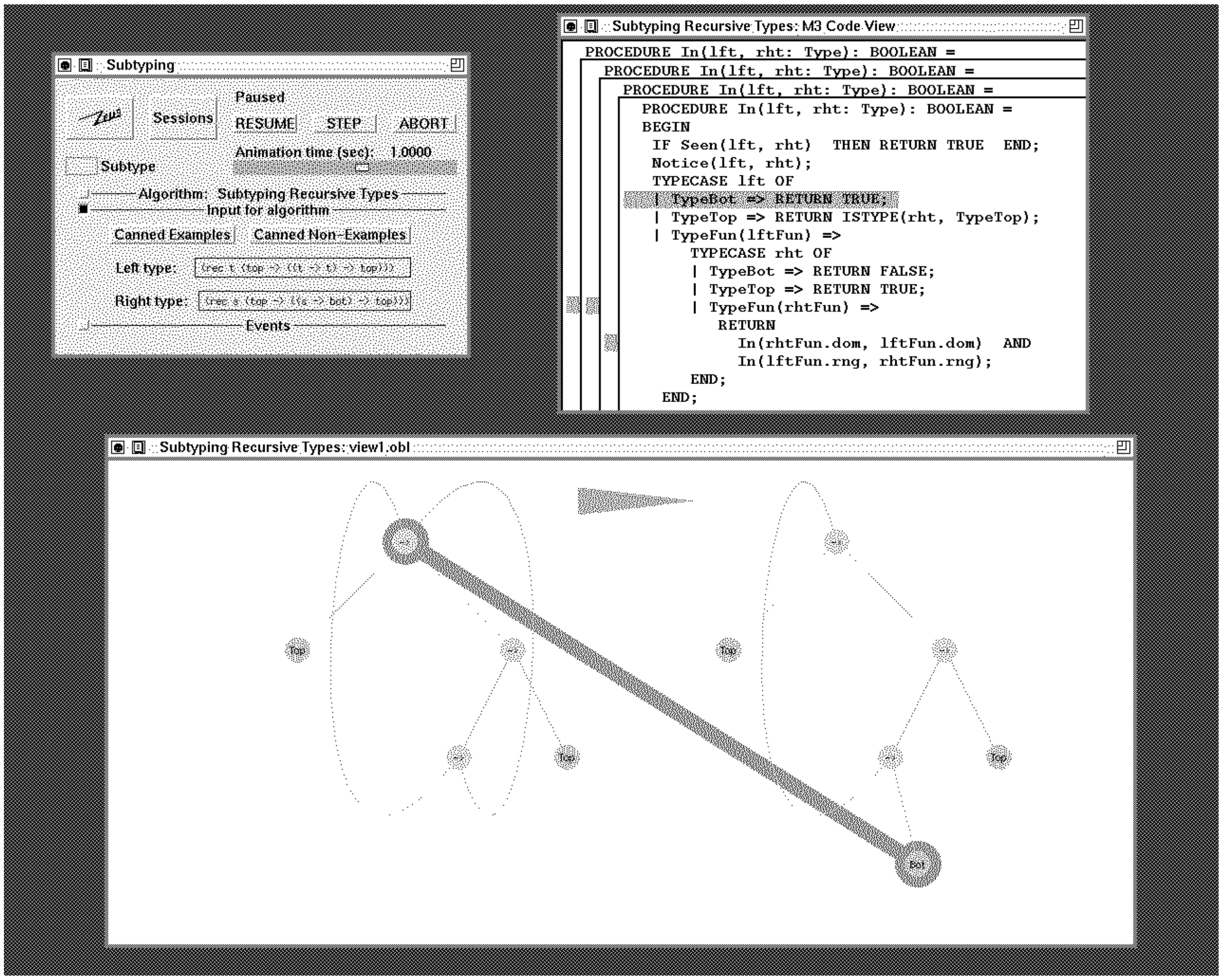

Shortest Path: SPGraph.obl
b
a
c
d
e

Fig. 6: Checking for Recursive Subtypes. Simple types can be represented as trees with type operators at the nodes and basic types at the leaves. Recursive types can be similarly represented as graphs, with loops expressing recursion. This animation illustrates an algorithm that checks whether two recursive types are in a subtype relation.

The algorithm works by traversing both types simultaneously with a two-headed cursor (the cyan bar) while remembering the pairs of nodes already visited (not shown) in order to stop looping. The direction of subtyping swaps every time the cursor descends the left branch of a function space node (arrow), since the function space type operator is anti-monotonic in its left argument. The cyan triangle represents the current direction of subtyping. The recursion stack is apparent in the *M3 Code* view in the upper-right. The standard Zeus control panel is in the upper-left window. Note the input parameters that the user can manipulate.


Subtyping
Zeus
Sessions
Paused
RESUME
STEP
ABORT
Animation time (sec): 1.0000
Subtype
Algorithm: Subtyping Recursive Types
Input for algorithm
Canned Examples
Canned Non-Examples
Events
Subtyping Recursive Types: M3 Code View
Subtyping Recursive Types: view1.obl
->
Top
->
->
Top
Top
->
->
->
Top
Bot

Left type: (rec t (top -> ((t -> t) -> top)))

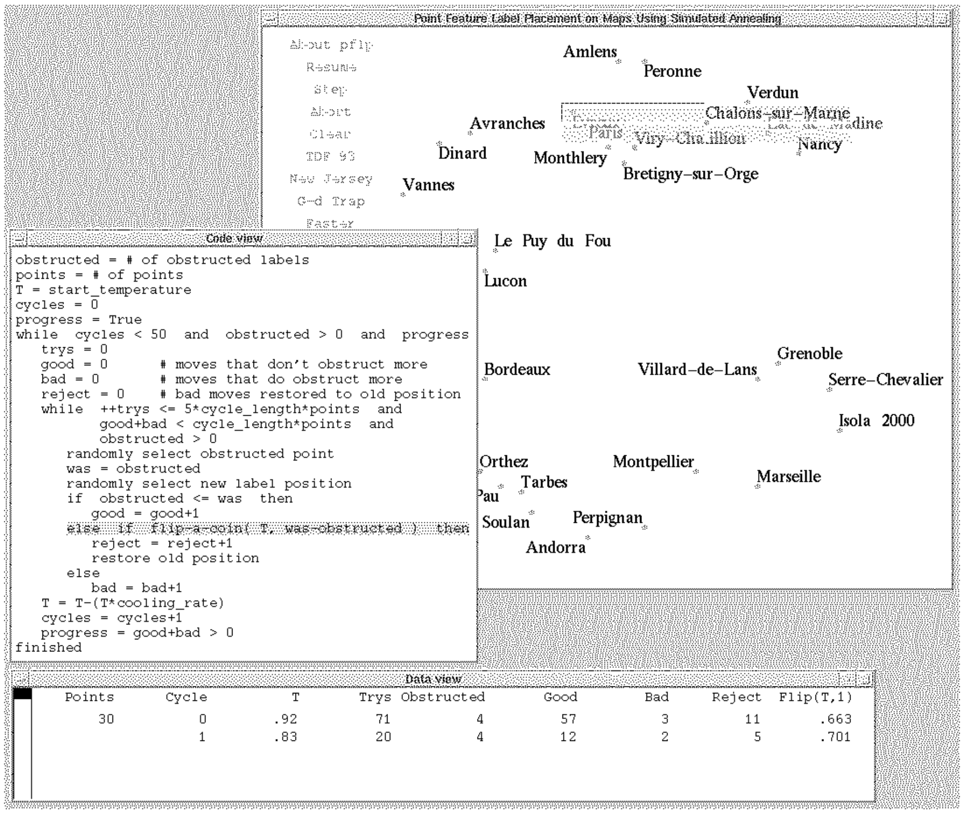
Right type: (rec s (top -> ((s -> bot) -> top)))

```
PROCEDURE In(lft, rht: Type): BOOLEAN =
  PROCEDURE In(lft, rht: Type): BOOLEAN =
    PROCEDURE In(lft, rht: Type): BOOLEAN =
      PROCEDURE In(lft, rht: Type): BOOLEAN =
      BEGIN
       IF Seen(lft, rht)  THEN RETURN TRUE  END;
       Notice(lft, rht);
       TYPECASE lft OF
        | TypeBot => RETURN TRUE;
        | TypeTop => RETURN ISTYPE(rht, TypeTop);
        | TypeFun(lftFun) =>
            TYPECASE rht OF
             | TypeBot => RETURN FALSE;
             | TypeTop => RETURN TRUE;
             | TypeFun(rhtFun) =>
                RETURN
                  In(rhtFun.dom, lftFun.dom)  AND
                  In(lftFun.rng, rhtFun.rng);
            END;
         END;
```

Fig. 7: Map Labeling. These two screen dumps show an animation of using simulated annealing to label point features on a map.

In the top snapshot, the four transparent yellow rectangles show possible label positions, and they move about the map as the algorithm processes conflicting labels (shown in red). The corresponding lines in the code view are highlighted appropriately. At this moment, the algorithm is attempting to improve the map by repositioning the label “Chalons-sur-Marne” from left to right. Since this move makes the map worse by adding a conflict with “Lac de Madine”, the program is “flipping a coin” to decide if the move should be rejected. This has a probability of .701 as shown in the Data view window.

The bottom snapshot shows the map labeled with no conflicts at the end of the run. The fine grain view of algorithm execution shown in the top snapshot has been replaced by graphs showing the number of unobstructed labels after each repositioning attempt. Here, a previous run with the standard cooling schedule (top graph) is compared with the less efficient current run that used an accelerated cooling schedule (bottom graph). Vertical yellow bars denote the end of each temperature cycle.



```
obstructed = # of obstructed labels
points = # of points
T = start_temperature
cycles = 0
progress = True
while  cycles < 50  and  obstructed > 0  and  progress
   trys = 0
   good = 0       # moves that don't obstruct more
   bad = 0        # moves that do obstruct more
   reject = 0     # bad moves restored to old position
   while  ++trys <= 5*cycle_length*points  and
          good+bad < cycle_length*points  and
          obstructed > 0
      randomly select obstructed point
      was = obstructed
      randomly select new label position
      if  obstructed <= was  then
         good = good+1
      else  if  flip-a-coin( T, was-obstructed )  then
         reject = reject+1
         restore old position
      else
         bad = bad+1
   T = T-(T*cooling_rate)
   cycles = cycles+1
   progress = good+bad > 0
finished
```

```
     Points       Cycle          T       Trys  Obstructed       Good       Bad     Reject  Flip(T,1)
         30           0        .92         71           4         57         3         11       .663
                      1        .83         20           4         12         2          5       .701
```

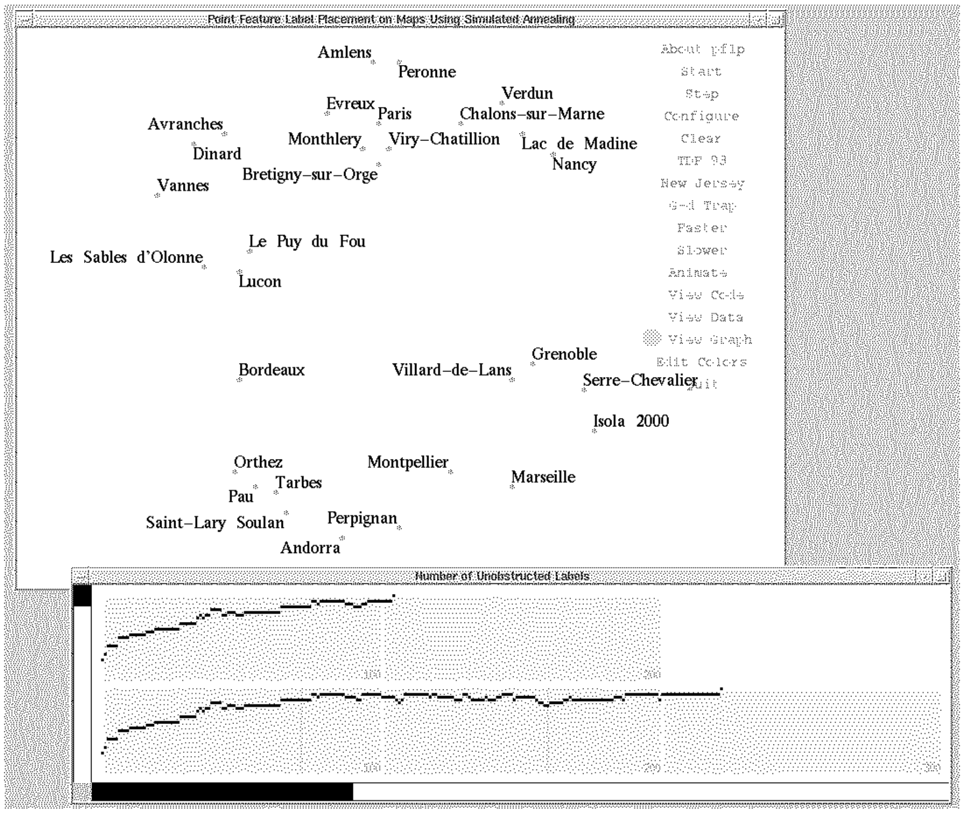


Fig. 8: Wheeler's Block-Sort Lossless Data Compression. This screen dump is from an animation of Wheeler's Block-Sort Lossless Data Compression algorithm[4].

The *Compress.obl* view at the left illustrates the compression phase. The string to be compressed, `bandana`, is shown at the top. In the middle, we see a matrix whose rows contain all the possible cyclic permutations of the string, in sorted lexicographic order. The original string is highlighted in pink, and the row's index is circled. All places in the original string that contain two instances of the same substring (e.g., `an`) will results in the last column of this matrix containing two adjacent (or nearly adjacent) copies of the substring's first character. Below the matrix we see the results of encoding the string from the last column of the matrix (i.e., `ndbanaa`) to take advantage of this property. This string is preceded by copy of the complete alphabet (`abnd`), and then each character of the string is replaced by the number of distinct characters between it and the preceding occurrence of the same character. The resulting sequence of counts (`1123310`) is likely to contain a disproportionate number of zeroes and other small numbers, so it can be compressed very effectively using a standard Huffman compression algorithm.

The *Decompress.obl* view at the top-right illustrates the decompression phase. The algorithm first decodes the sequence of counts, giving back the last column of the matrix. This column is then sorted, producing a copy of the original matrix's first column, and displayed to the right of the other column. The two adjacent columns produce all the digraphs in the original string. The algorithm then uses color to distinguish the multiple copies of same letter. For example, the first "a" in each column is red, the second is green, and the third is blue. Finally, the algorithm reassembles the original string by overlapping matching characters with matching colors. The remembered row index from the original matrix tells us where to start this process. The screen dump shows the reassembly process after it has reassembled just the `a` and the `b`.

The *WhyDecompressWorks.obl* view at the bottom-right illustrates the crux of the decompression phase: that the original string will be reassembled by overlapping matching characters with matching colors. This happens as long as multiple instances of each letter maintain their same relative order. For instance, the three `a`'s in both the left and right matrices appear in the order red-green-blue, from top to bottom.

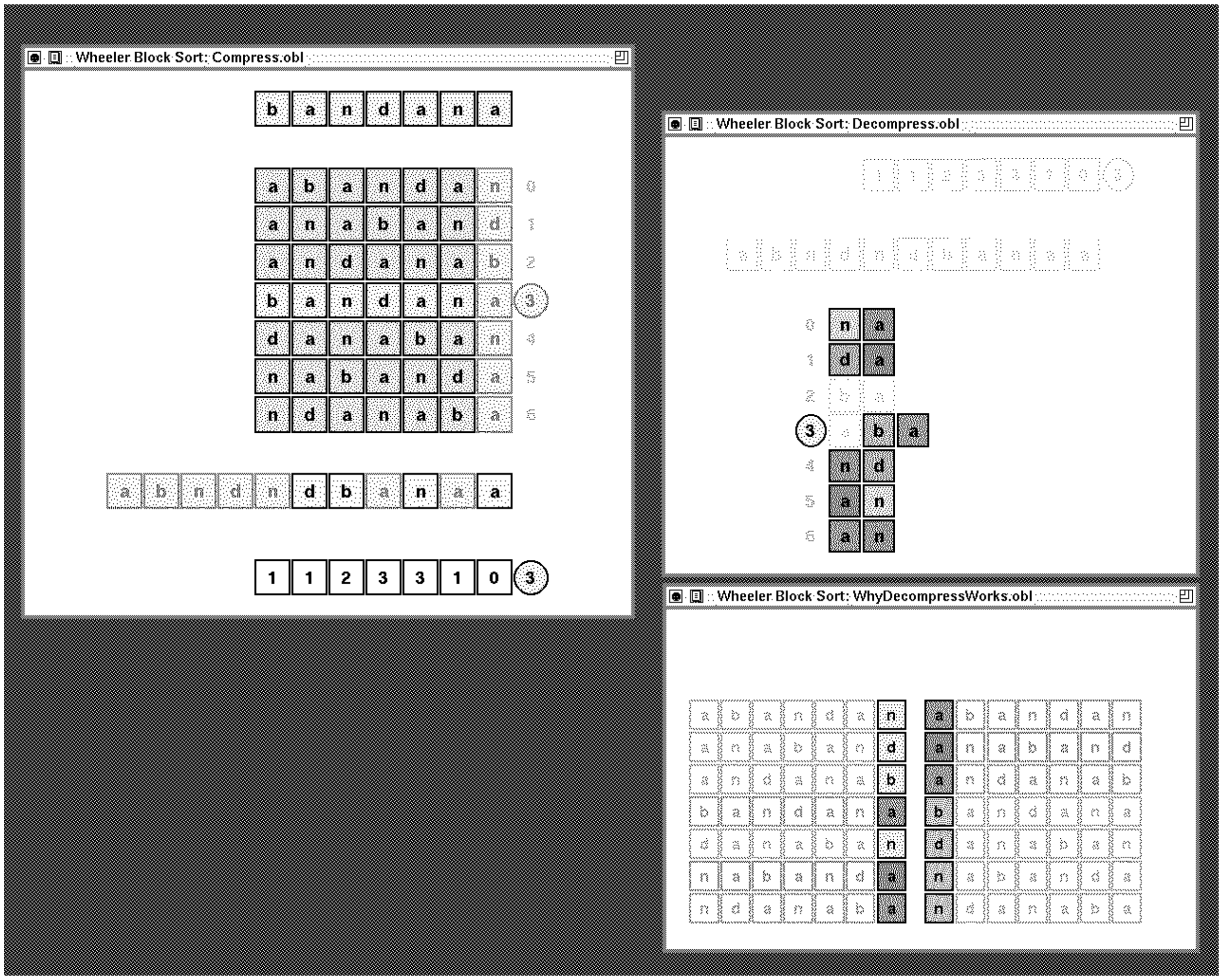

Wheeler Block Sort: Compress.obl
b a n d a n a
a b a n d a n 0
a n a b a n d 1
a n d a n a b 2
b a n d a n a 3
d a n a b a n 4
n a b a n d a 5
n d a n a b a 6
a b n d n d b a n a a
1 1 2 3 3 1 0 3
Wheeler Block Sort: Decompress.obl
Wheeler Block Sort: WhyDecompressWorks.obl

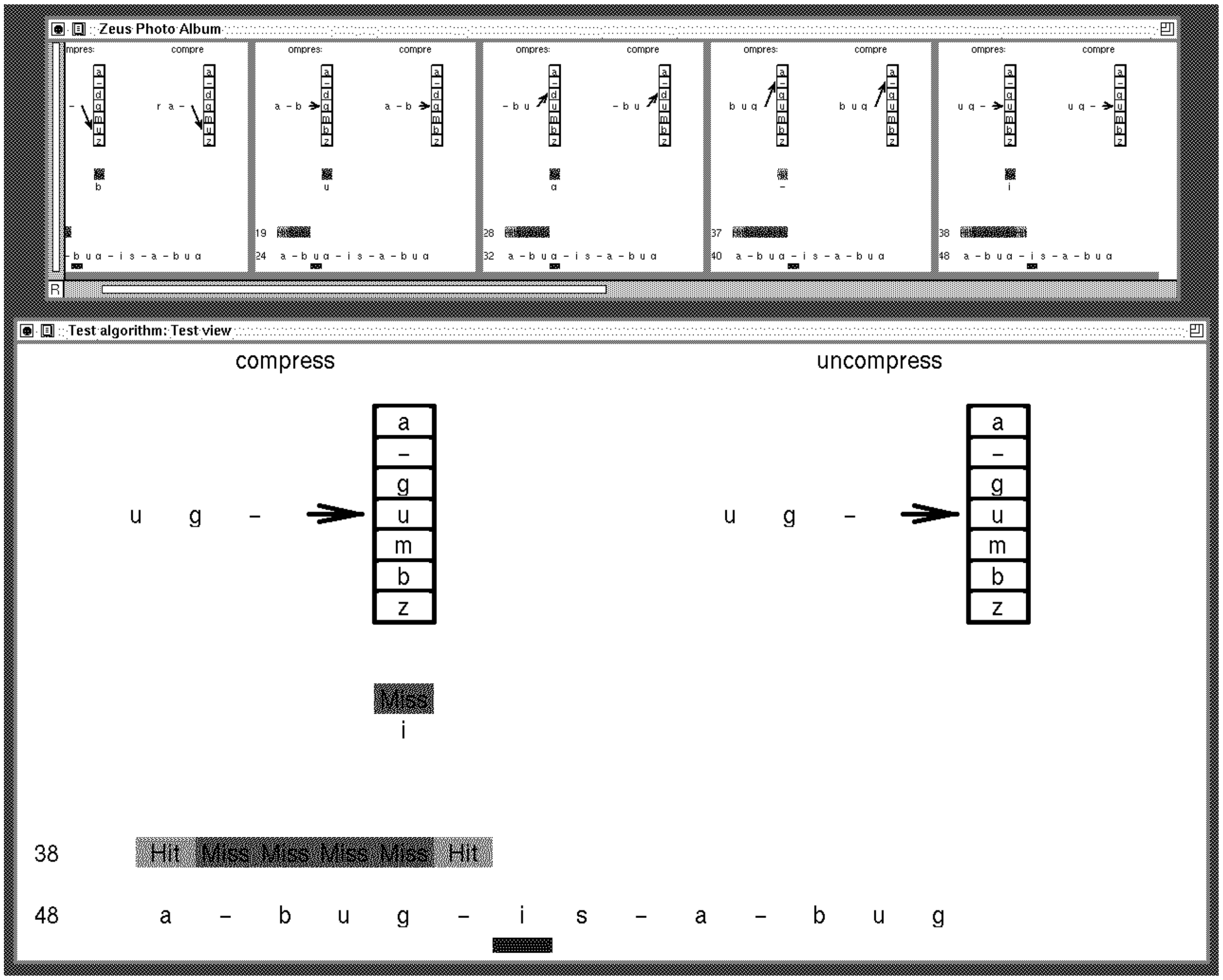
Fig. 9: Wheeler's "Revenge". This screen dump shows another lossless data compression algorithm. The *Zeus Photo Album* view at the top records a graphical history of the other view at various key stages in the algorithm's progress.

The compressor (shown in the left in the large view) runs an automaton that guesses what the next input character is. If it guesses correctly, the compressor outputs a "hit" marker; otherwise, it outputs a "miss" marker along with the input character that caused the miss. The bottom of the view shows the progress of the algorithm: In this snapshot, the characters `a-bug-` have already been processed. There were hits on the initial "a" and on the second hyphen. An uncompressed version of the string would require 48 bits: 8 bits for each of the 6 characters. The compressed version requires only 38 bits: each hit requires 1 bit and each miss 9 bits, 1 bit for the marker and 8 bits for the character itself.

The decompressor (shown in the right) reads the compressed input while also running the *same* automaton as the compressor runs. If the marker in the compressed input is a "hit," the decompressor gets the uncompressed character from its own automaton. Otherwise, the decompressor finds the uncompressed character on the compressed input stream, right after the "miss" marker.

In order to "guess" the next character, the automaton hashes the last three characters of the input stream to index an array. In the snapshot, the automation is using the characters `ug-`; the hash function returns a 3, indicating a "guess" of a `u` for the next character. However, this "guess" is wrong, and a "miss" marker and the character `i` will be put into the compressed output.

What makes this compression algorithm work is that the automaton adapts itself to the input stream by changing the contents of the array in the case of a miss. For example, in this snapshot, the `u` in the array will be replaced by the `i`. This increases the likelihood of hits, assuming that the input is more likely to see another `g-i` triple than another `g-u` triple. The *Zeus Photo Album* view shows the array adapting after each miss.


Zeus Photo Album
Test algorithm: Test view
compress
uncompress
a
–
g
u
m
b
z
u g –
Miss
i
38
Hit Miss Miss Miss Miss Hit
48
a – b u g – i s – a – b u g