Report Number: WUCS-92-14

1992

Abstraction in Algorithm Animation

Kenneth C. Cox and Gruia-Catalin Roman

Abstraction of information into visual form plays a key role in the development of algorithm animations. We present a classification for abstraction as applied to algorithm animation. The classification emphasizes the expressive power of the abstraction, ranging from simple direct presentation of the program's state to complex animations intended to explain the behavior of the program. We illustrate our classification by presenting several visualizations of a shortest path algorithm.

School of Engineering & Applied Science

Abstraction in Algorithm Animation

Kenneth C. Cox
Gruia-Catalin Roman

WUCS-92-14

March 1992

This paper appeared in
Proceedings of the 1992 IEEE Workshop on Visual Languages, September 1992, pp. 18-24.

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

PROCEEDINGS OF THE 1992 IEEE WORKSHOP ON

Visual Languages

SEPT. 15 – 18, 1992 SEATTLE, WASHINGTON

SPONSORED BY THE IEEE COMPUTER SOCIETY
AND THE IEEE COMPUTER SOCIETY TASK FORCE ON MULTIMEDIA COMPUTING

IEEE COMPUTER SOCIETY PRESS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

Abstraction in Algorithm Animation

Kenneth C. Cox Gruia-Catalin Roman

Department of Computer Science
Washington University
Saint Louis, MO 63130, USA

Abstract

Abstraction of information into visual form plays a key role in the development of algorithm animations. We present a classification for abstraction as applied to algorithm animation. The classification emphasizes the expressive power of the abstraction, ranging from simple direct presentation of the program's state to complex animations intended to explain the behavior of the program. We illustrate our classification by presenting several visualizations of a shortest path algorithm.

1. Introduction

A number of researchers have published taxonomic surveys of program visualization systems and approaches. Shu [9] focused on increasing degrees of sophistication exhibited by program visualization systems, from pretty-printing to complex algorithm animations. Myers [5] used a classification along two axes: the *aspect* of the program that is illustrated (code, data, algorithm) and the *display style* (static or dynamic). Chang [3], although not providing a taxonomy, defined program visualization as the use of graphical representations to illustrate programs, data, the structure of a system, or the dynamic behavior of a system. Brown [1] proposed classifying algorithm animations along three axes: *content* (direct or synthetic representation of information about the program), *transformation* (discretely or smoothly changing images), and *persistence* (representations of the current state or of the entire execution history). Finally, Roman and Cox [7] have suggested a taxonomy based on a treatment of visualization as a mapping from programs to graphical representations, characterizing systems based on the mapping.

We continue to treat visualization as a mapping in this paper. With this approach, we can characterize visualizations according to their domain (the scope of the information that is extracted), their range (what graphical techniques are used), the means whereby the mapping is specified, and the level of abstraction performed by the mapping process.

We identify five different levels of abstraction, summarized here. To convey an immediate intuition about each level, we indicate how the abstraction might be applied to an array-sorting algorithm:

- *Direct* representations map some aspect of a program directly to a picture, with no abstraction other than the simple transformation to visual form. In sorting, we might represent each of the items to be sorted by a rectangle whose length is proportional to the item's value and whose position is a function of its array index.
- *Structural* representations conceal or encapsulate extraneous data and present the resulting simplified view of the program. We might show only the partition elements of a quicksort as each is selected and placed.
- *Synthesized* representations derive and present information that is not explicitly present in the program, but can be derived from the information that is available. We might accumulate an execution history of the sort and present it using one spatial coordinate to represent time.
- *Analytical* representations move away from the presentation of information contained in or derived from the program and attempt to capture more abstract properties of its structure or behavior. We might try to capture an invariant such as "all items preceding the index i have been correctly sorted".
- *Explanatory* representations attempt to enhance the viewer's understanding through the addition of visual "hints" that have no exact counterpart in the program or its behavior. We might illustrate each exchange in the sort by causing the visual representations of the items to smoothly trade places, thus directing the viewer's attention to the elements.

We have listed these levels of abstraction in increasing order of abstractive power. In practice the distinctions between the levels are fuzzy and many visualizations mix several techniques. Explanatory presentations in particular have a wide range of expressive power, in that the basic technique—the use of visual events to guide the viewer—can be applied to most visualizations; even an otherwise direct abstraction of the information can be animated to enhance the presentation and guide the viewer.

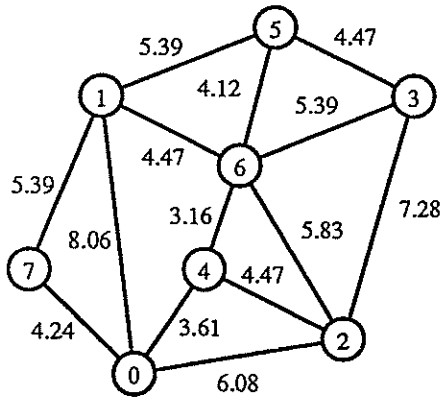

Figure 1. Undirected graph used as input to the all-pairs problem.

```
initialization
    k := 0
    forall i, j where 0 ≤ i < N ∧ 0 ≤ j < N do
      d[i, j] := W(i, j)
    end

program
    while k < N do
      forall i, j where 0 ≤ i < N ∧ 0 ≤ j < N do
        d[i, j] := min(d[i, j], d[i, k] + d[k, j])
      end
      k := k + 1
    end
```

Figure 2. Abstract code for the Floyd-Warshall all-pairs shortest path algorithm. The **forall** statement performs the indicated operation for all the indicated values of i and j in parallel.

In the remainder of this paper we illustrate each of these levels of abstraction using visualizations of the behavior of a simple parallel algorithm. Using a single algorithm permits easy contrast of the visualizations and simplifies explication of the key points of each. Further, we can explore the typical choices facing the animator during construction of a visualization. Section 2 describes the algorithm and the representation of its state that we will use. Sections 3 through 7 then discuss the visualization of this algorithm at each of the levels of abstraction.

2. Underlying Algorithm

We will use the all-pairs shortest path problem as our example. In this problem we are given an undirected graph G consisting of N nodes identified by numbers from 0 to $N-1$. Each (undirected) edge (i, j) has a positive cost given by the function $W(i,j)$. We assume that $W(i,i) = 0$ for all i and $W(i,j) = \infty$ for any i and j that are not connected by an edge. The problem is to determine the length of the shortest path between each pair of nodes. Figure 1 depicts a typical graph with distance information (in this case, the graph happens to be planar and the Euclidean distance is used).

The Floyd-Warshall algorithm solves the all-pairs problem in $O(N^3)$ serial time, or $O(N)$ time on a machine with $O(N^2)$ processors; we will visualize the parallel version. The parallel Floyd-Warshall algorithm operates on an array $d[i,j]$, where initially $d[i,j] = W(i,j)$ for all i and j. Initially each node is "unscanned". At each step the algorithm selects an unscanned node k and scans it by replacing $d[i,j]$ with $min(d[i,j]$, $d[i,k]+d[k,j])$ for all values of i and j in parallel. When all values of k have been scanned, each $d[i,j]$ contains the minimum distance from node i to node j. The correctness of the algorithm can be proved using an invariant which can be informally read as, *"At all times d[i,j] is the length of the shortest path from i to j all of whose internal nodes have been scanned"*. An abstract formulation of this algorithm appears in Figure 2. In this version, scanning of nodes is performed in numerical order with k ranging from 0 to N ($k = N$ occurs when the algorithm is complete).

In our declarative approach to visualization [6] we find it notationally-convenient to model the state as a collection of tuples. We can represent the array d by a collection of tuples of the type *dist*, where a tuple $dist(i,j,v)$ indicates that $d[i,j] = v$. The collection of tuples that represents the state will thus have one $dist(i,j,v)$ tuple for each pair of nodes i and j. We represent the variable k by a tuple of type *kvalue*; again, a tuple $kvalue(x)$ will be in our state representation exactly when $k = x$.

Whether we represent the state as a collection of variables or of tuples, we assume that the computation proceeds through a series of atomic transformations of the state. Our visualizations will examine the state after each transformation and map it to an image. In the case of the Floyd-Warshall algorithm, we treat each pass through the **while** loop as an atomic action. In other words, each atomic change consists of the parallel update of the array d (modification of the *dist* tuples) and the incrementing of the variable k (modification of the *kvalue* tuple).

3. Direct Representation

The most primitive graphical representations are obtained by mapping some aspect of the program directly to a picture. Because only very limited abstraction mechanisms are employed, it is often the case that the original information can be easily reconstructed from the graphical representation. Such a mapping may loosely be considered as a bijection between some appropriate subset of the state and the image.

Direct representations are common in existing visualization systems. Formatted layouts of the code, gauges set to indicate the values of variables, two-dimensional representations of linked lists and binary trees, and color encodings of values stored in an array are some examples of direct representations. Other forms are encountered primarily in CASE and debugging systems: flowcharting and similar graphical representations of code structure, monitoring of control flow through display of the currently-executing statement, and tracking of procedure invocation by presenting the call stack (possibly as overlapping windows, one per invocation) are all examples of direct representations.

The most common form of direct representation is the mapping of the values of variables to the attributes of graphical objects. Brown, for example, uses such a direct mapping in two visualizations of sorting algorithms using BALSA [2]. In both cases the array that is being sorted is represented in the image by a collection of objects. The index of each array element is mapped to the object's X-coordinate, while the value of the element maps to either the object's Y-coordinate (Brown's "Dots view") or its Y-size ("Sticks view").

In our example, we might choose to map the array represented by the *dist(i,j,v)* tuples to a two-dimensional array of graphical objects. The two dimensions represented by i and j would map to two spatial dimensions (say X and Y) and be obtained from the i and j components by a simple linear transform. The v component of each tuple could be represented by any of several object attributes, such as the color or size of the object. One interesting possibility is to represent v using the third dimension, obtaining the Z-coordinate or Z-size of the graphical object corresponding to *dist(i,j,v)* from a linear transform of v.

A rule which accomplishes such a direct mapping appears in Figure 3. This rule is expressed in the form used by Pavane, the algorithm animation system that we have developed [8]. A visualization is a composition of one or more mappings, with each mapping made up of one or more rules. A rule specifies a relationship between two collections of tuples. The rule in Figure 3, named *DistToBox*, maps from the state space of the computation (i.e., the collection of *dist* and *kvalue* tuples) to an entity known as the *animation space*. The animation space is a collection of tuples, each of which represents a single three-dimensional graphical object in the final image.

Pavane rules have the form

rulename ≡ variables : predicate ⇒ tuples

This is read as "For every instantiation of the *variables* such that the *predicate* is true, the corresponding *tuples* are in the output space of the rule." The *predicate* is permitted to include tests for the presence or absence of tuples; when examining the state space, this is logically equivalent to testing for the truth or falsity of a predicate over the state. In the case of the rule *DistToBox*, we attempt to find all combinations of the variables i, j, and v such that a tuple *dist(i,j,v)* exists in the state and v is finite (i.e., the distance from i to j is known). For each instantiation we generate a *box* object, which in the final image is rendered as a three-dimensional rectangular solid.

```
DistToBox ≡
    integer i,j; float v :
      dist(i, j, v), v < ∞
    ⇒
      box(corner := [XPOS(i), YPOS(j), 0.0],
        xsize := XSIZE, ysize := YSIZE,
        zsize := ZSIZE(v),
        color := ValueToColor(v), fill := true);
```

Figure 3. Direct visualization of the all-pairs algorithm. Each *dist* tuple representing a known distance is converted into a *box* object.

The *box* has a number of attributes, specified using the *name := value* syntax. The corner of the box is positioned in three-dimensional space by giving a triple *[x,y,z]* . In *DistToBox* the corner is obtained using *XPOS* and *YPOS*. These are defined by the animator; in this case both *XPOS(I)* and *YPOS(I)* are *(I–(N/2))*. The corner is thus positioned in the X-Y plane (Z = 0.0) at a location obtained by a linear transform of the i and j components of the tuple, as desired. The collection of all boxes produced by this rule will be arranged in a grid whose approximate center is the origin of the coordinate system, with the boxes spaced exactly one unit apart. *XSIZE* and *YSIZE* are each defined as the constant 0.9, which yields a small gap between the boxes.

ZSIZE is another linear transform defined by the animator, here applied to the v component of the *dist* tuple. The height of each box is therefore proportional to v. We also visually represent v using the color of the box. Each box is filled with a color which is a linear transform of v, for example a mapping to the spectrum such that small values of v map to red and large values to blue. This transform is accomplished by *ValueToColor*, which is also defined by the animator.

Figures 5(a) and 5(b) (on the last page of this paper) show two views of this visualization, the first from "above" looking down at the array of boxes and the second from an oblique angle. By inspecting this visualization, we can immediately determine the relative values contained in all *dist* tuples, with the caveat that any tuple *dist(i,j,∞)* is represented by a gap in the array. Despite the fact that all information about the *dist* tuples is contained in this image, viewing the entire visualization (that is, the sequence of images resulting from the successive state spaces of the algorithm) does not provide a great deal of information about the behavior of the program. Addition of a direct representation of the *kvalue* tuple (for example, markers on the rows and columns of the array indicating which nodes have been

scanned) does not greatly enhance the viewer's understanding.

4. Structural Representation

More abstract representations may be obtained by concealing or encapsulating some of the details associated with the program and using a direct representation of the remaining information. The goal is to exhibit the inherent structure of the information that is visualized. Some information is lost, but the overall presentation may be enhanced through the elimination of extraneous details. Interestingly, this resembles several approaches used in computing science in which the internal details of entities such as processes or data structures are concealed and only the external behavior of the object is modeled.

A typical application of structural abstraction would be in a visualization of a computation running on a network of processes, where the (possibly complex) states of the individual processes might be reduced to a simple 'active' or 'inactive'. Many other examples can be cited. Histograms may encode the relative frequency of message occurrences by type, concealing other message details. In an operating system, colored blocks may indicate memory usage without attempting to represent the state of the memory. In all these cases, the information presented is present in the program, although possibly obscured by details. The representation simply conveys the information in a more economical way by suppressing aspects not relevant to the viewer.

Several different structural abstractions are possible with the all-pairs algorithm. We could, for example, conceal the information about the exact distance between two nodes i and j, instead representing merely whether the distance is known (i.e., not ∞). This information is actually present in the visualization of the previous section, in that no box is generated for pairs where the distance is unknown, but the decision to represent the connectivity suggests a different visual representation: We can now use a *graph* whose arcs represent the connectivity information of the d array.

Attractive layout of graphs is a difficult problem, which may be approached in several ways. In many cases, positioning the graph nodes in some regular fashion — on a grid, or on the perimeter of a circle — may be satisfactory. Another approach is to provide functions to the visualization which position each of the nodes. The image in Figure 5(c) was constructed using the latter approach, obtaining the X-Y coordinates of each node from functions provided by the animator.

Presenting all the connectivity information in the shortest-path problem can result in a confusing image, since typically the graph will rapidly become complete. We can perform a further structural abstraction by concealing more of the information; specifically, we can choose a particular node as a "focus" and show only the connectivity for that node. Such a concealing is performed in Figure 5(c), with the focus node marked with a circle to draw the viewer's attention. The nodes are also color-coded to indicate the distance from the focus. This simple visualization requires only four rules, one of which serves only to place the circle around the focus node. However, it is still not particularly effective in explaining the behavior of the algorithm.

5. Synthesized Representation.

We use the term "synthesized representation" to refer to an abstraction in which the information of interest is not directly represented in the program but can be derived from the program data. This shift of perspective often occurs when the data representations selected by the programmer come into conflict with the needs of the animator, particularly where the animator is attempting to present some of the abstract concepts of the algorithm which have no explicit representation in the program. The animator must then construct and maintain a representation that is more convenient for his needs.

BALSA's "Compare-Exchange view" of sorting may be cited as such a representation. On the surface, this visualization is simply a direct visualization which shows the sequence of comparison and exchange operations performed during the sort. However, a moment's consideration leads to the observation that this is not part of the algorithm's state—that is, something maintained by the algorithm—but rather is part of its execution history. Thus, in order to present this view, the animator must collect and maintain the historical information as part of a synthesized representation.

Although many systems provide for forms of synthesized representations (if only in a limited fashion), few animations fully exploit the possibilities. This situation may be partly attributed to the fact that few systems have the ability to construct arbitrary mappings from programs to pictures. The animator typically has to do significant extra work in order to maintain the representation, for example by creating data structures whose contents are examined and updated by the procedures that produce the graphics.

Our rule-based approach provides a conceptually-similar mechanism which is somewhat simpler to use. Each rule may examine not only its input space, but also the previous versions of its input and output spaces, i.e., the versions resulting from the last application of the mapping. Using this mechanism, historical information about the behavior of the computation can be collected, maintained, and examined.

Returning to the all-pairs program, the animator might decide that representing the actual *paths* rather than the *distances* would be useful. Note that path information is not explicitly present in the algorithm. However, we can recover it from the information in the array d. At each point in the algorithm, the edge from i to j is on a (current) shortest path from h to j precisely when both $d[h,i]$ and $d[h,j]$ are finite and $d[h,j] = d[h,i] + W(i,j)$. This makes recovery of the path information from the state space a simple matter; we

need only write a single rule to find all h, i, and j that satisfy the above relation.

Figure 5(d) shows the resulting visualization. Both the path edges (drawn with thick lines) and the underlying graph edges (thinner lines) are displayed. The display of the edges is helpful to the viewer but still does not lead to an intuitive understanding of the algorithm.

6. Analytical Representation

Analytical representations attempt to capture abstract program properties, such as those used in correctness proofs. The goal of such abstractions is to de-emphasize the operational mechanics of program execution and focus on issues that are important in analytical thinking about the program. The approach is rooted in the notion that properties that are important in reasoning formally about programs should, when translated into visual form, also help the viewer understand the program's behavior.

Although many researchers have exploited program properties in constructing visualizations, few explicitly recognize the importance of such properties. Our own work on Pavane is the only research of which we are aware in which program properties play a central role in the design of visualizations. In part, this may be due to our focus on the visualization of concurrent processes, a domain where formal reasoning about program properties is crucial to the understanding of algorithms.

We can illustrate the analytical approach using the all-pairs correctness property previously mentioned: "At all times $d[i,j]$ is the length of the shortest path from i to j all of whose internal nodes have been scanned". We can visualize this by exhibiting the shortest paths (as in the previous visualization) and marking the scanned nodes in some appropriate manner, say by coloring scanned nodes green and unscanned nodes red.

Because this visualization uses the node color to represent the status, no representation of the node's distance is contained in the final image. We could have avoided this by retaining color for distance and using some other distinction for status. However, color is not particularly effective at encoding quantitative information such as the distances in our example. Inspecting Figure 5(d), we see that although some idea about the relative distances is conveyed ("blue is farther than yellow"), the quantitative information about these distances is lost.

If we want to effectively show the distance information, we should choose an attribute that is more amenable to quantitative analysis. Among such attributes are the dimensions and position of the graphical object. This leads us back to the approach of our first visualization in Figure 9(b), where the height of the boxes in the Z-dimension was used to indicate the distance between the nodes. As in that visualization, if the distance to a node is unknown we can simply not produce the corresponding object. Unfortunately, this conflicts with the desire to show the status of the nodes, since a node may have any combination of scanned/unscanned status and known/unknown distance.

One solution is shown in Figure 5(e). Here, each node is represented by either one or two objects. The colored circle in the planar representation of the graph represents the node's scanned/unscanned status, while distance information is represented by the Z-coordinates of the spheres "above" the graph. The edges of the graph are shown in the planar representation, while the path edges are shown in the three-dimensional representation. The "focus" node is marked with circles, as before; since this node is always at a distance of 0 from itself, its sphere and circle overlap. This could have been avoided with a simple adjustment of the rules (adding a constant value to the Z-coordinates of the spheres, for example), but the effect is not unpleasant and, more importantly, slightly enhances the presentation by emphasizing the special role of the focus node.

7. Explanatory Representation

Sophisticated visualizations go beyond presenting simple representations of the program state and use a variety of visual techniques to illustrate program behavior. These visual events often have no counterparts in the computation being depicted by the animation. They are added for the sake of improving the aesthetic quality of the presentation, out of the desire to communicate the implications of a particular computational event, and in order to focus the viewer's attention. In essence, the animator takes the liberty of adding events to the representation of the program in order to "tell the story" better.

As mentioned previously, sorting provides a simple example of such abstraction. Imagine a sorting algorithm in which the array that is being sorted is mapped to a row of rectangular objects. The exchange of two array elements can be illustrated by a corresponding swap of the rectangles representing those elements. This draws the viewer's attention to the objects and ensures that the exchanges are properly perceived, but the images generated during the animated swap do not in any way correspond to the state of the computation. When the elements with indices 3 and 5 are exchanged, there is no point at which either is stored in the array at index 4, but the rectangles in the animation will pass through the X-coordinate corresponding to index 4 during the animation. This point is actually quite significant, as a poor choice of animation activities could lead a naive viewer to assume a corresponding activity in the program.

TANGO [10] provides sophisticated facilities for the production of explanatory animations of algorithms. TANGO uses the "path-transition" paradigm for visualization, which simultaneously maps program data to graphical objects and program actions to animation actions involving the objects. The animation actions are defined using *paths*, arbitrary sequences of coordinates which are interpreted as points in some attribute space (image coordinates, object visibility, and so on). The animation actions may be composed using several operators to generate various effects.

Animation in Pavane is accomplished by using time-dependent attributes for the tuples in the animation space. Times are measured in terms of frames, i.e., the number of images which are produced. Several functions are provided which produce values as a function of time, e.g., the *ramp* function produces a linear interpolation between two values.

In our all-pairs problem, we might want to animate the changes in the spheres that represent the path information. By considering the behavior of the algorithm, we can isolate several different cases: a sphere may be added to the image (when the distance to the node is first found), a sphere may change color (as a result of scanning), a sphere may change its height (when the distance to the node is reduced), or a sphere may remain unchanged. Interestingly, we can prove from the program and the rules that produce the spheres that a sphere cannot simultaneously change color and height (that is, when node k is scanned, none of the $d[i,k]$ change); this simplifies the development of rules for the animation.

A frame from one of the transitions generated by an explanatory visualization is shown in Figure 5(f). The second node from the right has just been scanned, as shown by the color change. As a result, the distance to the rightmost node is now known and the corresponding sphere is added to the graph. The sphere at the right is increasing in radius as it moves upward to its final position. Naturally, the print medium limits our ability to convey this type of abstraction; film or video tape is much more appropriate.

8. Conclusions

Selection of a proper abstraction plays a key role in the effective visual communication of information. Several areas of the computing field can be identified in which use of an effective abstraction is essential. Chief among these is visual exploration and monitoring, where a viewer examines and possibly interacts with a visual representation of an executing process; the use of a readily-understood abstraction of the information about the process is essential. The word "process" here is carefully chosen, because another major area in which visual abstractions play a critical role is in the monitoring of "real-world" processes such as industrial plant behavior. This ties in with the development of reliable visual human-machine interfaces, whether to physical processes or to information structures such as operating systems and databases.

In pedagogical settings, the presentation and explanation of a program can be greatly enhanced by a visualization of its behavior. Indeed, viewing an appropriately-chosen animation can provide significant intuition about the behavior of a program, and even serve as an aid in debugging. Here, as in the areas mentioned previously, abstraction may be used to filter out extraneous information, to draw the viewer's attention to important areas, and to efficiently convey large amounts of information.

Despite the importance of abstraction, few researchers have systematically examined its application. By presenting a classification of levels of abstraction and giving appropriate examples, we hope that we have brought the attention of the community to this area and suggested useful areas of exploration.

Acknowledgments

This work was supported in part by the National Science Foundation under the Grant CCR-9015677. The Government has certain rights in this material.

References

[1] Brown, M. H., "Perspectives on Algorithm Animation," *CHI'88 Human Factors in Computing Systems*, Washington, DC, USA, pp. 33-38, 1988.

[2] Brown, M. H., "Exploring Algorithms using Balsa-II," *IEEE Computer*, vol. 21, no. 5, pp. 14-36, 1988.

[3] Chang, S.-K., *Visual Languages and Visual Programming*, Plenum Press, New York, NY, 1990.

[4] McCleary, G. F., "An Effective Graphic "Vocabulary"," *IEEE Computer Graphics and Applications*, vol. 3, no. 2, pp. 46-53, 1983.

[5] Myers, B. A., "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97-123, 1990.

[6] Roman, G.-C., and Cox, K. C., "A Declarative Approach to Visualizing Concurrent Computations," *Computer*, vol. 22, no. 10, pp. 25-36, 1989.

[7] Roman, G.-C., and Cox, K. C., "Program Visualization: The Art of Mapping Programs to Pictures," *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, 1992, pp. 412-420 (invited paper).

[8] Roman, G.-C., Cox, K. C., Wilcox, C. D., and Plun, J. Y., "Pavane: a system for declarative visualization of concurrent computations," to appear in *Journal of Visual Languages and Computing*.

[9] Shu, N. C., *Visual Programming*, Van Nostrand Reinhold Company, New York, NY, 1988.

[10] Stasko, J., "The path-transition paradigm: a practical methodology for adding animation to program interfaces," *Journal of Visual Languages and Computing*, vol. 1, no. 3, pp. 213-236, 1990.

[11] Tufte, E. R., *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.

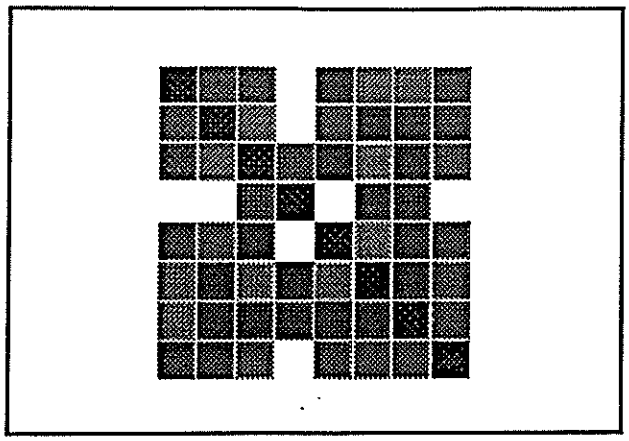

Figure 5(a). Direct abstraction in which distances are mapped to boxes arranged in the X-Y plane. The color of each square is a function of the distance between the two nodes, with shorter distances mapped to darker shades.

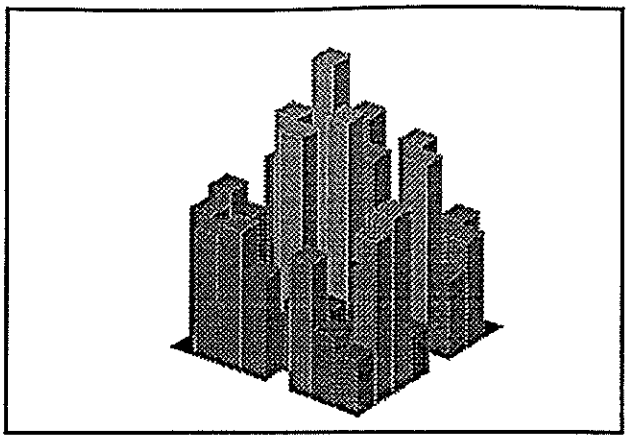

Figure 5(b). Oblique view of the boxes depicted in Figure 5(a). The height of each box is proporitional to the value stored in the corresponding *dist* tuple.

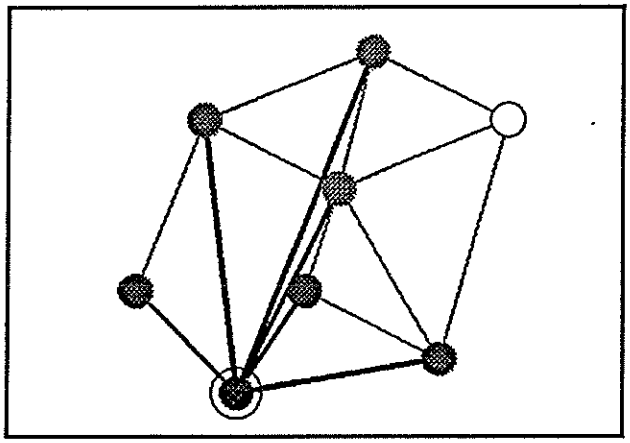

Figure 5(c). Structural abstraction in which the focus node (circled) is connected by a line to those nodes for which the distance from the focus is known. The edges of the original graph are also shown.

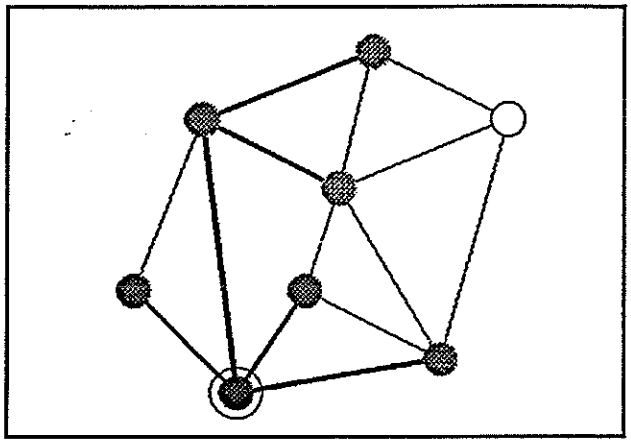

Figure 5(d). Synthesized abstraction in which actual shortest paths are extracted from the state and presented (thick lines). This path information is not explicitly maintained by the program.

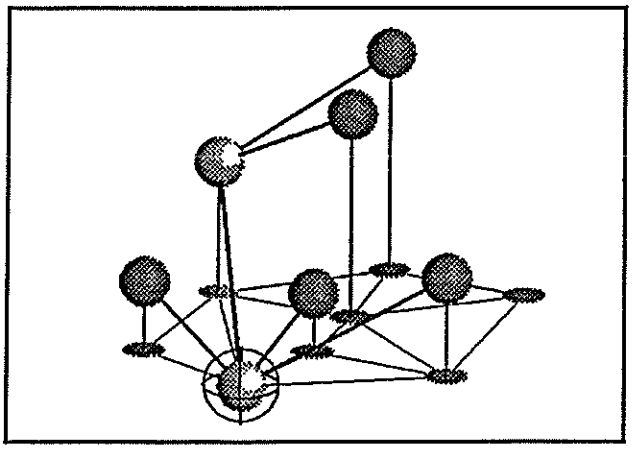

Figure 5(e). Analytical abstraction which attempts to convey the invariant *"d[i,j] is the length of the currently-known shortest path from i to j all of whose internal nodes have been scanned"*.

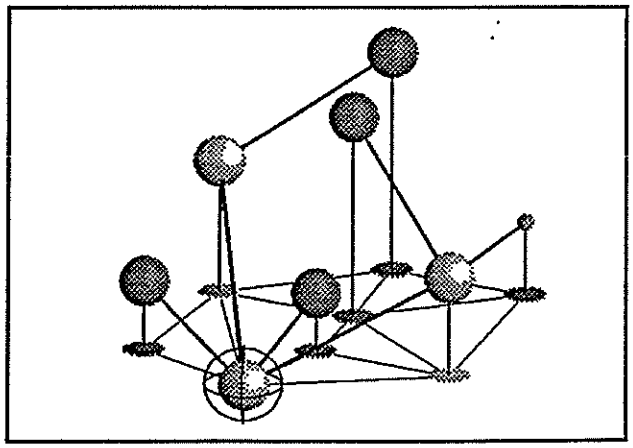

Figure 5(f). Explanatory abstraction; this is a frame from the transition following Figure 5(e). The sphere at the right is being added to the image following the scanning of the sphere second from the right.