Alcom-FT Technical Report Series
ALCOMFT-TR-01-149

# Visualization in Algorithm Engineering: Tools and Techniques[1]

*Camil Demetrescu*
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Italy
Email: demetres@dis.uniroma1.it.
URL: http://www.dis.uniroma1.it/~demetres

*Irene Finocchi*
Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza", Italy
Email: finocchi@dsi.uniroma1.it.
URL: http://www.dsi.uniroma1.it/~finocchi

*Giuseppe F. Italiano*
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Italy
Email: italiano@info.uniroma2.it
URL: http://www.info.uniroma2.it/~italiano

*Stefan Näher*
DFB IV - Informatik
Universität Trier, Germany
Email: naeher@informatik.uni-trier.de
URL: http://www.informatik.uni-trier.de/~naeher

[1] This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT), by the Italian Ministry of University and Scientific Research (Project "Algorithms for Large Data Sets: Science and Engineering" and by CNR, the Italian National Research Council under contract n. 00.00346.CT26.

# Contents

# 1. Visualization in Algorithm Engineering: Tools and Techniques

The process of implementing, debugging, testing, engineering and experimentally analyzing algorithmic codes is a complex and delicate task, fraught with many difficulties and pitfalls. In this context, traditional low-level textual debuggers or industrial-strength development environments can be of little help for algorithm engineers, who are mainly interested in high-level algorithmic ideas and not particularly in the language and platform-dependent details of actual implementations. Algorithm visualization environments provide tools for abstracting irrelevant program details and for conveying into still or animated images the high-level algorithmic behavior of a piece of software.

In this chapter we address the role of visualization in algorithm engineering. We survey the main approaches and existing tools and we discuss difficulties and relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes.

## 1.1 Introduction

There has been an increasing attention in our community towards the experimental evaluation of algorithms. This is producing several tools whose target is to offer a general-purpose workbench for the experimental validation and fine-tuning of algorithms and data structures: software repositories and libraries, collections and generators of test sets, software systems for supporting implementation and analysis are relevant examples of such an effort. In particular, in the last years there has been an increasing attention towards the design and implementation of interactive environments for developing and experimenting with algorithms, such as editors for test sets and development, debugging, and visualization tools.

In this chapter we address the role of algorithm visualization tools in algorithm engineering. According to a standard definition [45], algorithm animation is a form of high-level dynamic software visualization that uses graphics and animation techniques for portraying and monitoring the computational steps of algorithms. Systems for algorithm animation have matured significantly since, in the last decade, high-quality user interfaces have become a standard in a large number of areas.

Nevertheless, the birth of algorithm visualization can be dated back to the 60's, when Licklider did early experiments on the use of graphics for monitoring the evolution of the content of a computer memory. Knowlton was the first to address the visualization of dynamically changing data structures in his films demonstrating the Bell Lab's low-level list processing language [29]. During the 70's, the potentialities of program animation in a pedagogical setting were pointed out by several authors, and this research ended up with the realization in 1981 of the videotape Sorting Out Sorting [3], which represents a milestone in the history of algorithm animation and has been successfully used to teach sorting methods to computer science students for more than 15 years. Sorting Out Sorting is a 30-minute color film that explains nine internal sorting algorithms, illustrating both their substance and their differences in efficiency. Different graphical representations are provided for the data being sorted, and showing the programs while running on their input makes it clear at any step how such data are partially reorganized by the different algorithms. A new era begun with the 80's, when bit-mapped displays became available on workstations: researchers attempted at going beyond films and started developing interactive software visualization systems and exploring their utility not only for education, but also for software engineering and program debugging. Dozens of algorithm animation systems have been developed since then.

Thanks to the capability of conveying a large amount of information in a compact form and to the ability of human beings at processing visual information, algorithm animation systems turn out to be powerful tools also in algorithm engineering, in particular in several phases during the process of design, implementation, analysis, tuning, experimental evaluation, and presentation of algorithms. Actually, visual debugging techniques can help highlight hidden programming or conceptual errors, i.e., discover both errors due to a wrong implementation of an algorithm and, at a higher level of abstraction, errors possibly due to an incorrect design of the algorithm itself. Sometimes, algorithm animation tools can help in designing heuristics and local improvements in the code difficult to figure out theoretically, to test the correctness of algorithms on specific test sets, to discover degeneracies, i.e., special cases for which the algorithm may not produce a correct output. Their use can leverage the task of monitoring complex systems or complex programs (e.g., concurrent programs), and makes it possible also to analyze problem instances not limited in size and complexity which even long and boring handiwork would not be able to deal with. Not last, visualization is an attractive medium for algorithms researchers who want to share and disseminate their ideas.

In spite of the great research devoted in recent years to designing and developing algorithm visualization facilities, the diffusion of the use of such systems for algorithm engineering is still limited. We believe this is mostly due to the lack of fast prototyping mechanisms, i.e., to the fact that realizing an animation often requires heavy modifications of the source code at hand and therefore a great effort. Instead, the power of an algorithm animation system should be in the hands of the end-users, possibly unexperienced, rather than of professional programmers or of the developers of the visualization tool. In addition, it is very important for a software visualization tool to be able to animate not just "toy programs", but significantly complex algorithmic codes, and to test their behavior on large data sets. Unfortunately, even those systems well suited for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck, such as changes of resolution and scale, selectivity, and elision of information. Finding a solution to this kind of limitations is nowadays a challenge for algorithm visualization systems.

In this chapter we survey the main approaches and existing tools for the realization of animations of algorithms. In particular, Section 1.2 is concerned with the description of software visualization systems and libraries supporting visualization capabilities. Section 1.3 describes the two main approaches used by visualization tools: event-driven and data-driven. In Section 1.4 we discuss difficulties and present relevant examples where visualization systems helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. Conclusions and challenges for algorithm visualization research are finally listed in Section 1.5.

## 1.2 Tools for Algorithm Visualization

In this section we survey some algorithm visualization systems, discussing their main features and the different approaches introduced by each of them. We do not aim at being exhaustive, but rather we try to highlight the aspects of these systems interesting from an algorithm engineering point of view. We also describe some tools that will be used in Section 1.4 for illustrating how to prepare algorithm animations for debugging or demonstrations. For a more comprehensive description of software visualization systems we refer the interested reader to [45] and to the references therein.

Balsa [8], followed a few years later by Balsa-II [9], was the first system able to animate general-purpose algorithms and pioneered the interesting events approach, later used by many other tools. In order to realize an animation, the points of the source code which are strategically important are annotated with procedure calls that generate visualization events. At run time, events are collected by an event manager that provides to forward them to the views so as to update the displayed images. Balsa-II supports a good level of interactivity, allowing execution control by means of step-points and stop-points. In order to provide a measure of an algorithm's performance, it also supports a way to associate different costs to different events and to count the number of times each interesting event occurs, which may be interesting for profiling algorithmic codes. Zeus [11] is an evolution of Balsa-II and adds to the interesting events approach some object-oriented features: each view is created by deriving a standard base `View` class and can be provided with additional methods to handle each interesting event. Zeus also extensively uses color and sound [12] and deals with three-dimensional objects [13], thus making it possible to realize highly-customizable visualizations.

TANGO [42] (Transition-based ANimation GeneratiOn) introduced the path-transition paradigm [41] for creating smooth continuous animations. This paradigm relies on the use of four abstract data types (location, image, path, and transition) and animations are realized by handling instances of these data types by means of suitable operations defined on them. X-TANGO [43] is the X-Windows based follow-up of TANGO. Polka [44] introduces the support for the animation of concurrent programs: the programmer can assemble and present the whole animation using an explicit global clock counter as a timing system. It also includes a graphical front-end, called Samba, that is driven by a script produced as a trace of the execution.

UWPI [26] (University of Washington Program Illustrator) automatically provides high-level visualizations for the data structures of simple algorithmic codes. The visualization is automatically performed by the system thanks to an "inferencer", which analyzes the data structures in the source code, both at compile-time and at run-time, and suggests a number of possible displays for each of them. Clearly, due to the lack of a deep knowledge of the logic of the program, only the visualization of simple data structures, such as stacks or queues, can be supported.

Pavane [38, 40] marks the first paradigm shift in algorithm visualization since the introduction of interesting events. It features a declarative approach to the visualization of concurrent programs. It conceives the visualization as a mapping between the state of the computation and the image space: the transformation between a fixed set of program variables and the final image is declared by using suitable rules. This seems very important for developing visual debugging tools for languages such as Prolog and Lisp. Furthemore, the non-invasiveness of the declarative approach seems very important also in a concurrent framework, since the execution may be non-deterministic and an invasive visualization code may change the outcome of a computation.

TPM [21] (Transparent Prolog Machine) is a debugging tool for the post-mortem visualization of computer programs written in the Prolog programming language. In order to deal with the inherent complexity of Prolog programs, TPM features two distinct views: a fine-grained view to represent the program's locality and a coarse-grained view to show the full execution space via animated AND-OR trees. The overall trace structure also captures the concept of backtracking to find alternative solutions to goals. ZStep95 [32] is a reversible and animated source code stepper for LISP programs that provides a powerful mechanism for error localization. It maintains a complete history of the execution and is equipped with a fully reversible control structure: the user lets its program running until an error is found and then can go back to discover the exact point in which something went wrong. Moreover, a simple and strict connection between the execution and its graphical output is obtained by elementary clicking actions.

Debugging concurrent programs is more complicated than understanding the behavior of sequential codes: not only concurrent computations may produce vast quantities of data, but also the presence of multiple threads that communicate, compete for resources, and periodically synchronize may result in unexpected interactions and non-deterministic executions. Many tools have been therefore realized to cope with these issues. The Gthreads library [50] builds and displays a program graph as threads are forked and functions are called: the vertices represent program entities and events and the arcs temporal orderings between them. The Hence system [6] offers animated views of the program graph obtained from execution of PVM programs. Message passing views are supported by the Conch system [49]: processes appear around the outside of a ring and messages move from the sending process to the receiving one by traversing the center of the ring. This is useful to detect undelivered messages, as they remain in the center of the ring. Kiviat graphs for monitoring the CPU utilization of each processor are also supported by other systems such as ParaGraph [25] and Tapestry [33].

Leonardo [17] is an integrated environment for developing, animating, and executing general-purpose C programs. Animations are realized according to a logic-based approach, i.e., by embedding in the source code declarations that provide high-level graphical interpretations of the program's variables. As the system automatically reflects the modifications of the program state into the displayed images, a high level of automation is reached. Animations can be fully customized by means of a graphical vocabulary including basic geometric shapes as well as primitives for visualizing graphs and trees. Smoothly changing images are also supported by the system to help the viewer maintain context [19]. In addition, code written with Leonardo is completely reversible: when running code backwards, variable assignments are undone, output sent to the console disappears, graphics drawn are undrawn, and so on. The reversibility is extended to the full set of standard ANSI functions. This feature, combined with the declarative approach, makes the system well suited for visual debugging purposes. Differently from many other visualization systems, Leonardo has been widely distributed over the Internet and includes several animations of algorithms and data structures.

Computational geometry is an area where the visualization and animation of programs is a very important tool for the understanding, presentation, and debugging of algorithms, and the animation of geometric algorithms is mentioned among the strategic research directions in computational geometry [47]. It is thus not surprising that increasing attention has been devoted to algorithm visualization tools for computational geometry (see, e.g., [2, 4, 20, 27, 46]). In this chapter we focus particularly our attention on *GeoWin*, a `C++` data type which can be easily interfaced with algorithmic software libraries of great importance in algorithm engineering such as CGAL [22] and LEDA [34]. The design and implementation of GeoWin was influenced by LEDA's graph editor *GraphWin* (see [34], chapter 12). Both data types support a number of programming styles which have shown to be useful in demonstration and animation programs. Examples are the use of *result scenes* and the *event handling* approach which will be discussed in section 1.4.3. An instance *gw* of the data type *GeoWin* is an editor that maintains a collection of so-called *scenes*. Each scene in this collection has an associated *container* of geometric *objects* whose members are displayed according to a set of visual attributes (color, line width, line style, etc.). One of the scenes in the collection can be *active*. It receives the input of all editing operations and can be manipulated through the interactive interface. Both the container type and the object type have to provide a certain functionality. The container type must implement the STL list interface [35], in particular, it has to provide STL-style iterators, and for all geometric objects a small number of functions and operators (for stream input and output, basic transformations, drawing and mouse input in a LEDA window) have to be defined. Any combination of container and object type that fulfill these requirements for containers and objects, respectively, can be associated with GeoWin scene in a `gw.new_scene()` operation.

More recent trends in algorithm animation include distributed systems over the Web. JEliot [24, 31] is an automatic system for the animation of simple Java programs. After the Java code has been parsed, the user can choose the cast of variables to be visualized on the scene according to built-in graphical interpretations. The user needs to write no additional code: in other words, animation is embedded in the implementation of data type operations. The graphical presentation is based on a "theater metaphor" where the script is the algorithm, the stages are the views, the actors are program's data structures depicted as graphical objects, and the director is the user.

CATAI [14] (Concurrent Algorithms and data Types Animation over the Internet) tries to impose a little burden on the task of animating algorithms. The main philosophy behind this system is that any algorithm implemented in an object-oriented programming language (such as `C++`) should be easily animated. This should make this system easy to use, and is based on the idea that an average programmer or an algorithm developer should not invest too much time in getting an actual animation of the algorithm up and running. This is not always the case, and often animating an algorithm can be as difficult and as time consuming as implementing the algorithm itself from scratch. CATAI has an advantage over systems where the task of animating an algorithm is highly non-trivial. Producing animations almost automatically, however, can limit flexibility in creating custom graphic displays. If the user is willing to invest more time on the development of an animation, he or she can produce more sophisticated graphics capabilities, while still exploiting the features offered by the system.

JDSL [5] (Java Data Structures Library) is a library of data structures written in the Java programming language that supports the visualization of the fundamental operations on abstract data types; it is well suited for educational purposes, as students obtain a predefined visualization of their own implementation by simply implementing JDSL Java interfaces with predefined signatures.

VEGA [27] (Visualization Environment for Geometric Algorithms) is also a client/server visualization environment for geometric algorithms. It guarantees a small bandwidth communication, thus achieving good performance even in slow networks. The end-user can interactively draw, load, save, and modify graphical scenes, can animate algorithms on-line or show saved runs off-line, and can customize the visualization by specifying a suitable set of view attributes.

WAVE [20] (Web Algorithm Visualization Engine) uses a publication-driven approach to algorithm visualization over the Web and is especially well-suited for geometric algorithms. Algorithms run on a developer's remote server and their data structures are published on blackboards held by the clients. Animations are realized by writing suitable visualization handlers and by attaching them to the public data structures.

## 1.3 Two Main Approaches: Event-Driven and Data-Driven

In this section we focus on the main features of animation systems that are appealing for their deployment in algorithm engineering. From the viewpoint of the algorithmic developer, it would be highly desirable to rely on systems that offer visualizations at a very high level of abstraction. Namely, one would be more interested in visualizing the behavior of a complex data structure, such as a graph, rather than in obtaining a particular value of a given pointer. Furthermore, algorithm designers could be very interested in visualizations that are reusable and that can be created with little effort from the algorithmic source code at hand: this could be of substantial help in speeding up the time required to produce a running animation.

Achieving simultaneously high level of abstraction and fast prototyping makes the task of developing algorithm animation systems highly non trivial. Indeed, it is possible to visualize automatically static or even running code, but at a very low level of abstraction, i.e., when the entities to be displayed and the way they change can be directly deduced from the code and the program state. For instance, the program counter tells us the next instruction, from which the line of the code to be executed can be easily recovered and highlighted in a suitable view. Also, primitive and composite data types can be mapped into canonical representations, thus displaying for free the data and the data flow. Conventional debuggers rely on this assumption but they lack capability of abstraction: they are unable to convey information about the algorithm's fundamental operations and to produce high-level synthesized views of data and of their manipulations. For example, if a graph is represented by means of an adjacency matrix, a debugger can automatically display only the matrix, but not the graph according to its usual representation with vertices and edges. Towards this aim, some extra knowledge, such as the interpretation of matrix entries, should be provided to the visualization system.

The considerations above are at the base of the distinction between *program* and *algorithm* visualization. In particular, an algorithm visualization system should be able to illustrate salient features of the algorithm, which appears to be difficult, if not impossible, with a completely automatic mechanism. The opposition *automation* versus *high-level* and *customization* possibilities makes it necessary to define a method for specifying the visualization, i.e., a suitable mechanism for binding pictures to code. In the remainder of this section, we discuss the two major solutions proposed in the literature: the event-driven and the data-driven approaches. For a comprehensive discussion of other techniques used in algorithm visualization we refer the interested reader to [10, 36, 37, 39, 45].

**Event-driven approach.** A natural approach to algorithm animation consists of annotating the algorithmic code with calls to visualization routines. The first step consists of identifying the relevant actions performed by the algorithm which are interesting for visualization purposes. Such relevant actions are usually referred to as *Interesting Events*. As an example, in a sorting algorithm the swap of two items can be considered an interesting event. The second step consists of associating each interesting event with a suitable animation scene. In our example, if we depict the values to be sorted as a sequence of sticks of different heights, the animation of a swap event might be realized by exchanging the positions of the two sticks corresponding to the values being swapped. Animation scenes can be specified by setting up suitable visualization procedures which drive the graphic system according to the actual parameters generated by the particular event. Alternatively, these visualization procedures may simply log the events in a file for a *post-mortem* visualization. The calls to the visualization routines are usually obtained by annotating the original algorithmic code in the points where the interesting events take place. This can be done either by hand or by means of specialized editors.

In addition to being simple to implement, the main benefit of the event-driven approach is that interesting events are not necessarily low-level operations (such as comparisons or memory assignments), but can be more abstract and complex operations designed by the programmer and strictly related to the algorithm being visualized (e.g., the swap in the previous example, as well as a rotate operation in the management of an AVL tree). Major drawbacks include the followings: realizing an animation may require the programmer to write several lines of additional code; the event-driven approach is invasive (even if the code is not transformed, it is augmented); the person who is in charge of realizing the animation has to know the source code quite well in order to identify all the interesting points.

**Data Driven Approach.** Data driven systems rely on the assumption that observing how the variables change provides clues to the actions performed by the algorithm. The focus is on capturing and monitoring the data modifications rather than on processing the interesting events issued by the annotated algorithmic code. Conventional debuggers can be actually viewed as data driven systems, since they provide a direct feedback of variable modifications.

Specifying an animation in a data driven system consists of providing a graphical interpretation of the *interesting data structures* of the algorithmic code. It is up to the system to ensure that the graphical interpretation reflects at any time the state of the computation of the program being animated. In the case of conventional debuggers, the interpretation is fixed and cannot be changed by the user: usually, a direct representation of the content of variables is provided. The debugger just updates the display after each change, sometimes highlighting the latest variable that has been modified by the program to help the user maintain context. In a more general scenario, an adjacency matrix used by the code may be visualized as a graph with vertices and edges, an array of numbers as a sequence of sticks of different heights, and a heap vector as a balanced tree. As the focus is only on data structures, the same graphical interpretation, and thus the same visualization code, may be reused for any algorithm which uses the same data structure. For instance, any sorting algorithm that manages to reorganize a given array of numbers may be animated with the same visualization code that displays the array as a sequence of sticks.

The main advantage of this approach over the event driven technique is that a much greater ignorance of the code is allowed: indeed, only the interpretation of the variables has to be known to animate a program. In Section 1.4.2 we will describe how we realized the animation of an algorithm in the system Leonardo with a very little knowledge of the code to be visualized. On the other hand, focusing only on data modification may sometimes limit customization possibilities making it difficult to realize animations which would be natural to express with interesting events.

We believe that a combination of the two approaches described in this section would be most effective in algorithm animation as they capture different aspects of the problem. In our own experience, we found examples where one approach is much more preferable than the other and vice-versa.

In some cases, we even found useful to use both approaches simultaneously for realizing the same animation.

## 1.4 Visualization in Algorithm Engineering

In this section we present relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes. In particular, we will consider examples where visualization can provide deep insights into the design of algorithms at the level of profiling and experimental evaluation (Section 1.4.1) and where animation has greatly simplified the task of debugging complex algorithmic code (Section 1.4.2). One of the most important aspects in algorithm engineering is the development of libraries. It is thus quite natural to try to interface visualization tools to algorithmic software libraries. Two examples of such an effort are further considered in Sections 1.4.3 and 1.4.4. In particular, we will show how demonstrations of geometric algorithms can be easily realized and interfaced with libraries (Section 1.4.3), and how fast animation prototyping can be achieved by reusing existing visualization code (Section 1.4.4).

### 1.4.1 Animation Systems and Heuristics: Max Flow

The maximum flow problem, first introduced by Berge and Ghouila-Houri in [7], is a fundamental problem in combinatorial optimization that arises in many practical applications. Examples of the maximum flow problem include determining the maximum steady-state flow of petroleum products in a pipeline network, cars in a road network, messages in a telecommunication network, and electricity in an electrical network. Given a capacitated network $G = (V, E, c)$ where $V$ is the set of nodes, $E$ is the set of edges and $c_{xy}$ is the *capacity* of edge $(x, y) \in E$, the maximum flow problem consists of computing the maximum amount of flow that can be sent from a given source node $s$ to a given sink node $t$ without exceeding the edge capacities. A flow assignment is a function $f$ on edges such that $f_{xy} \leq c_{xy}$, i.e., edge capacities are not exceeded, and for each node $v$ (bar the source $s$ and the sink $t$), $\sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw}$, i.e., the assigned incoming flows and the outgoing flows are equal. Usually, it is required to compute not only the maximum amount of flow that can be sent from the source to the sink in a given network, but also a flow assignment that achieves that amount.

Several methods for computing a maximum flow have been proposed in the literature. In particular, we mention the network simplex method proposed by Dantzig [18], the augmenting path method of Ford and Fulkerson, the blocking flow of Dinitz, and the push-relabel technique of Goldberg and Tarjan [1].

The push-relabel method, that made it possible to design the fastest algorithms for the maximum flow problem, sends flows locally on individual edges (push operation), possibly creating flow excesses at nodes, i.e., a *preflow*. A preflow is just a relaxed flow assignment such that for some nodes, called *active nodes*, the incoming flows may overcome the outgoing flows. The push-relabel algorithms work by progressively transforming the preflow into a maximum flow by dissipating excesses of flow held by active nodes that either reach the sink or return back to the source. This is done by repeatedly selecting a current active node according to some selection strategy, pushing as much exceeding flows as possible towards adjacent nodes that have a lower estimated distance from the sink paying attention not to exceed the edge capacities, and then, if the current node is still active, updating its estimated distance from the sink (relabel operation). Whenever an active node cannot reach the sink anymore as no path to the sink remains with some residual unused capacity, its distance progressively increases due to relabel operations until it gets greater than $n$: when this happens, it starts sending flows back towards the source, whose estimated distance is initially forced to $n$. This elegant solution makes it possible to deal with both sending flows to the sink and draining undeliverable excesses back to the source through exactly the same push/relabel operations. However, as we will see later, if taken "as is" this solution is not so good in practice.

Two aspects of the push-relabel technique seem to be relevant with respect to the running time: (1) the selection strategy of the current active node, and (2) the way estimated distances from the sink are updated by the algorithm.

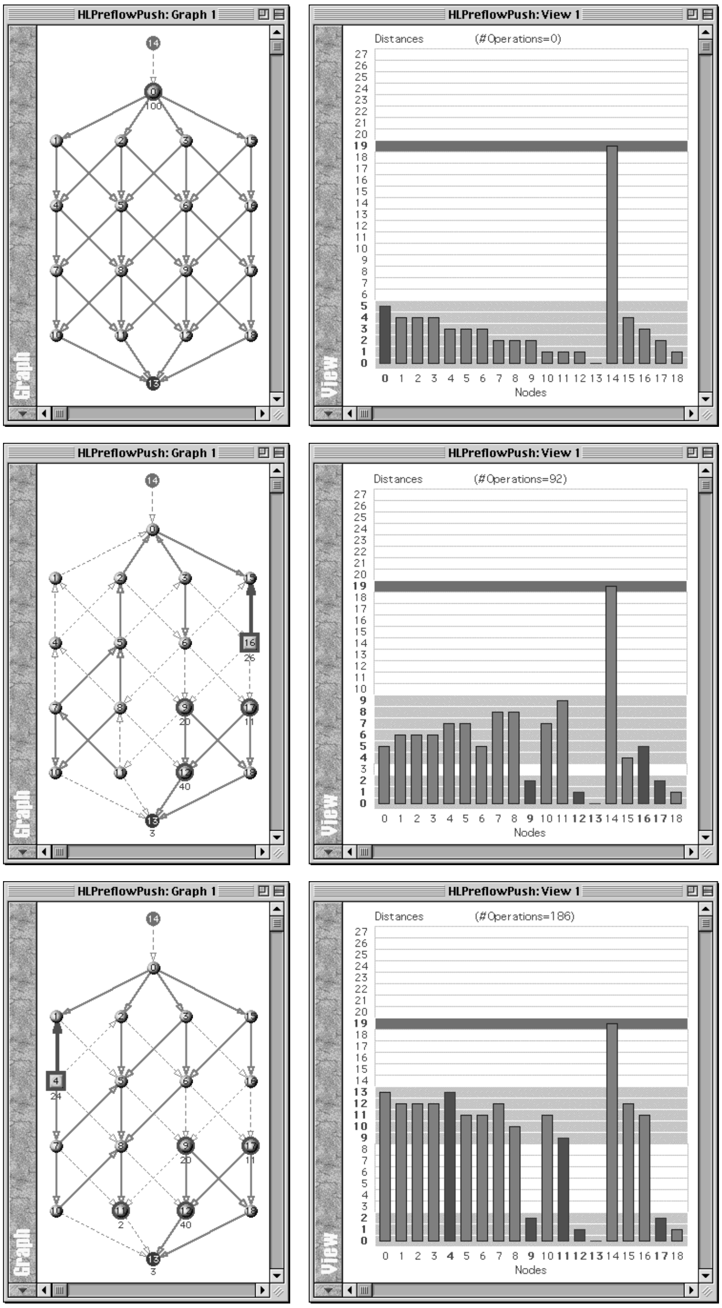
The selection strategy of the current active node has been proved to significantly affect the asymptotic worst-case running time of push-relabel algorithms [1]: as a matter of fact, if active nodes are stored in a queue, the algorithm, usually referred to as the FIFO preflow-push algorithm, takes $O(n^3)$ in the worst case; if active nodes are kept in a priority queue where each extracted node has the maximum estimated distance from the sink, the worst-case running time decreases to $O(\sqrt{m}n^2)$, which is much better for sparse graphs. The last algorithm is known as the highest-level preflow-push algorithm.

Unfortunately, regardless of the selection strategy, the push-relabel method yields in practice very slow codes if taken "ad litteram". Indeed, the way estimated distances from the sink are maintained has been proved to dramatically affect the practical performances of the push-relabel algorithms. For this reason, several additional heuristics for the problem have been proposed. Though these heuristics are irrelevant from an asymptotic point of view, the experimental study presented in [16] proves that two of them, i.e., the global relabeling and the gap heuristics, could be extremely useful in practice.

**Global Relabeling Heuristic.** Each relabel operation increases the estimated distance of the current active node from the sink to be equal to the lowest estimated distance of any adjacent node, plus one. This is done by considering only adjacent nodes joined by edges with some non-zero residual capacity, i.e., edges that can still carry some additional flows. As relabel operations are indeed local operations, the estimated distances from the sink may progressively deviate from the exact distances by losing the "big picture" of the distances: for this reason, flow excesses might not be correctly pushed right ahead towards the sink, and may follow longer paths slowing down the computation. The global relabeling heuristic consists of recomputing, say every $n$ push/relabel operations, the exact distances from the sink, and the asymptotic cost of doing so can be amortized against the previous operations. This heuristic drastically improves the practical running time of algorithms based on the push-relabel method [16].

**Gap Heuristic.** Different authors [15] have observed that, at any time during the execution of the algorithm, if there are nodes with estimated distances from the sink that are strictly greater than some distance $d$ and no other node has estimated distance $d$, then a gap in the distances has been formed and all active nodes above the gap will eventually send their flow excesses back to the source as they no more reach the sink. This is achieved by the algorithm through successive repeated increments of the estimated distances done via relabel operations until they get greater than $n$. The problem is that a huge number of such relabeling operations may be required. To avoid this, it is possible to efficiently keep track of gaps in the distances: whenever a gap occurs, the estimated distances of all nodes above the gap are immediately increased to $n$. This is usually referred to as the gap heuristic and, according to the study in [16], it is a very useful addition to the global relabeling heuristics if the highest-level active node selection strategy is applied. However, the gap heuristic does not seem to yield the same improvements under FIFO selection strategy.

The 5 snapshots $a$, $b$, $c$, $d$ and $e$ shown in Fig. 1.1 and in Fig. 1.2 have been produced by the algorithm animation system Leonardo [17] and depict the behavior of the highest-level preflow push algorithm implemented with no additional heuristics on a small network with 19 nodes and 39 edges. The animation aims at giving an empirical explanation about the utility of the gap heuristic under the highest-level selection. The example shows that this heuristic, if added to the code, could have saved about 80% of the total time spent by the algorithm to solve the problem on that instance. Both the network and a histogram of the estimated distances of nodes are shown in the snapshots: active nodes are highlighted both in the network and in the histogram and flow excesses are reported as node labels. Moreover, the edge currently selected for a push operation is highlighted as well. Notice that the source is initially assigned distance $n$ and all nodes that eventually send flows back to the source get distance greater than $n$.




(a)

Network status and distances after the initialization phase.

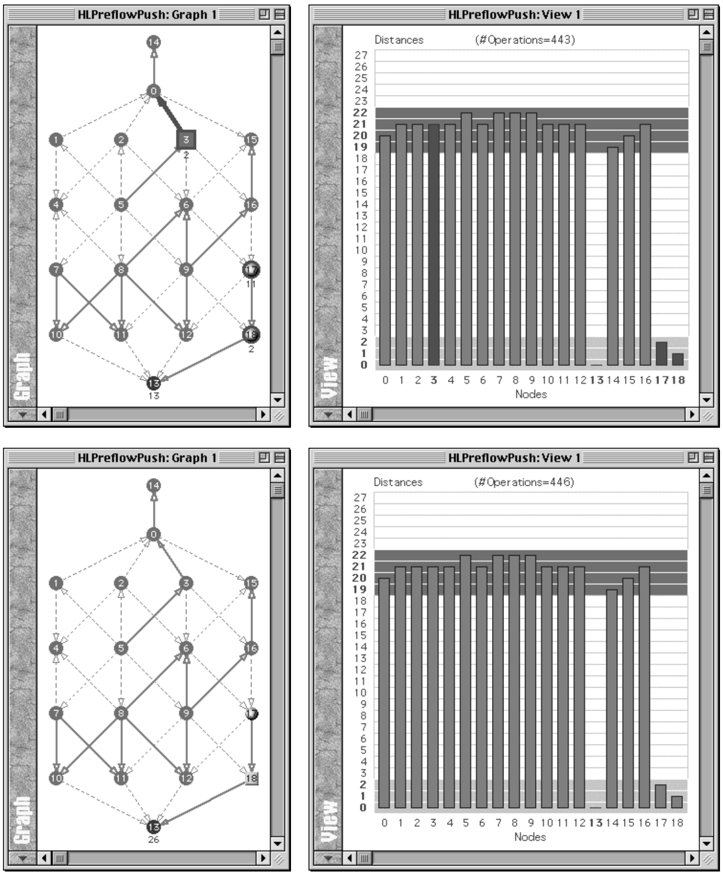
(b)

After 92 operations a gap has been formed. Nodes with distance greater than the gap no more reach the sink. Their distance should be directly increased to n through the gap heuristic.

(c)

Nodes with distance greater than the gap are being slowly relabeled step after step if the gap heuristic is not implemented.

**Fig. 1.1.** Highest-level preflow push maxflow algorithm animation in Leonardo. Snapshots a, b, c.




(d)

After 443 operations the distances of all nodes above the gap have been increased to n and their flow excesses are being drained back to the source. The gap heuristic could have saved the last 351 operations on this instance, i.e., about 80% of the total time spent by the algorithm to solve the problem.

(e)

After 446 operations the maximum flow has been determined by the algorithm and no more active nodes remain.

**Fig. 1.2.** Highest-level preflow push maxflow algorithm animation in Leonardo. Snapshots d, e.

### 1.4.2 Animation Systems and Debugging: Spring Embedding

In this section we address an important application of animation systems: debugging complex algorithmic codes. In particular, we describe our own experience with a graph layout algorithm and show how its animation was crucial for debugging an available implementation, and for discovering convergence problems due to numerical errors. The algorithm, due to Kamada and Kawai [28], is based on a force-directed paradigm, which uses a physical analogy to draw graphs: graph vertices and edges define a force system and the algorithm seeks a configuration with locally minimal energy. The embedding produced by the algorithm is also known as *spring embedding*: indeed, Kamada and Kawai's algorithm finds an embedding by introducing a force system where vertices are mutually connected by springs. More in details, the algorithm attempts to find an embedding of the graph where Euclidean distances between vertices are as close as possible to the lengths of their shortest paths. The energy of this system is thus:

$$E = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{k_{i,j}}{2} \left( dist(i,j) - L \cdot \ell(i,j) \right)^2 ,$$

where $dist(i,j)$ is the Euclidean distance between vertices $i$ and $j$, $\ell(i,j)$ is the length of a shortest path between $i$ and $j$ in the embedded graph, $L$ is the desirable length of a single edge in the drawing, and $k_{i,j}$ is the strength of the spring between vertices $i$ and $j$.

In order to find a local minimum of the energy, Kamada and Kawai make use of a two-dimensional Newton-Raphson method to look where partial derivatives are zero (or close to zero). In particular, at each step all vertices are frozen, except for one vertex which is moved to a stable point by means of an iterated process. In more details, the vertex with largest first-order partial derivatives is selected, and it is repeatedly moved towards a local minimum (based on the value of second-order partial derivatives). Those iterations terminate when first-order partial derivatives become small enough. This is a very high-level description of the algorithm, which should suffice for our goals: the interested reader is referred to [28] for the full details of the method.

We received a C implementation of this algorithm that was taken straightly out of the paper. The implementation seemed flawed with convergence problems on some graph instances: however, despite many efforts, the authors of the code were unable to track down exactly the bug. We were thus asked to try to animate their implementation, in order to gain better understanding and hopefully debugging this piece of algorithmic code. At that time, we did not know much about Kamada and Kawai's algorithm, and did not know much about the implementation either: furthermore, we did not want to invest too much time in studying in depth either the paper or the implementation.

We set up an animation in Leonardo [17]: the only information we had to retrieve from the implementation concerned the data structures used to store the graph and the positions of the vertices as progressively refined and returned by the algorithm. In particular, we had to look only at the following lines from the implementation code:

```
int n;
int G[ MAXNODES ][ MAXNODES ];
struct  { double x,y; } pos[ MAXNODES ];
```

where `G` is the adjacency matrix of the graph, `n` is the number of vertices, and `pos` contains the $x$ and $y$ coordinates of each vertex in the drawing. Our animation was set up so as to show how vertices were changing their position as the `pos` array was being updated by the algorithm, thus illustrating the intermediate drawings produced in different steps of the algorithm. We

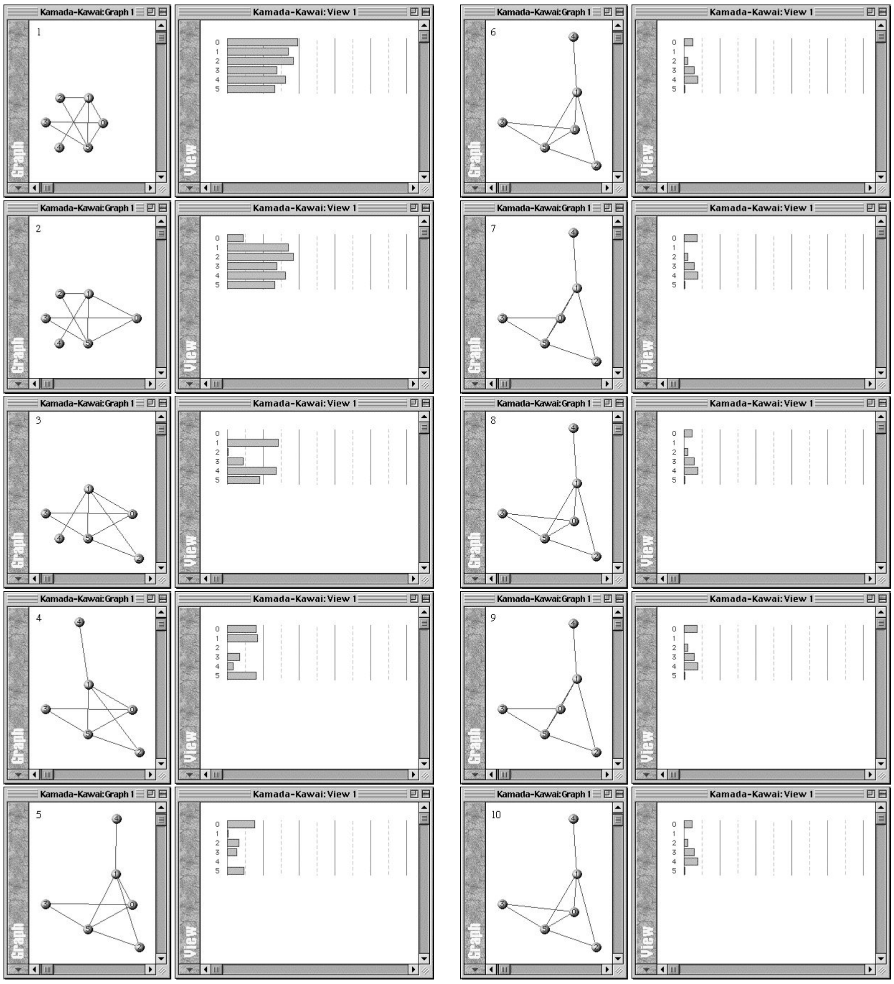



**Fig. 1.3.** Storyboard of Kamada and Kawai's algorithm animated with Leonardo.

emphasize that it was very difficult to figure out this process using only the numerical information displayed by a conventional textual debugger.

In order to produce the animation of this algorithm with Leonardo, we embedded into the source code the following lines:

```
/**
   Graph( Out 1 );
   Node( Out N, 1) For N: InRange( N, 0, n-1 );
   Arc( U, V, 1 ) If G[ U ][ V ] != 0;
   NodePos( N, Out X, Out Y, 1 )
      Assign X = pos[ N ].x * 100
             Y = pos[ N ].y * 100;
**/
```

The goal of this visualization code is to declare a window displaying a graph with id number `1`. The vertices of the visualized graph are labeled with integers in the range $[0, n-1]$, and there is an edge `(U,V)` if and only if the corresponding entry in the adjacency matrix is non-zero. The coordinates `(x,y)` of vertex `N` of graph `1` are proportional to `pos[N].x` and `pos[N].y` respectively. The animation hinges upon the fact that, when the original algorithmic code is executed, any change in the variables `n`, `G`, and `pos` is automatically reflected in the displayed images.

Figure 1.3 illustrates different snapshots of the animation throughout the execution. Together with the window displaying the graph, there is another window showing the potential energy of each vertex (the visualization code for this window is not reported). As it can be easily seen from the right column in Figure 1.3, the implementation seems looping among different energy configurations while trying to position vertex 0: in particular, the animation shows that vertex 0 is oscillating between two different positions. This was much more difficult to discover without visualizing the running code, since relevant values of `pos[ 0 ].x` and `pos[ 0 ].y` were never identical in the sequence of cycling steps.

### 1.4.3 Animation Systems and Demos: Geometric Algorithms

The visual nature of geometric applications makes them a natural area for designing systems which describe relevant aspects of the algorithm behavior by using animation. Indeed, the animation of geometric algorithms is mentioned among the strategic research directions in computational geometry [47] and increasing attention has been put towards designing algorithm visualization tools for computational geometry (see, e.g., [2, 4, 27, 46]).

In this section we show how to use the GeoWin data type introduced in Section 1.2, which was designed to be easily interfaced with algorithmic software libraries such as CGAL [22] and LEDA [34]. In particular, we discuss two of the basic features of GeoWin.

**Result Scenes.** A *result scene* is a GeoWin scene that depends on one or more *input scenes*. The dependence is defined by a function to be called for the objects of the input scenes. The contents of the result scene is just the output of this function. Whenever the input scene is modified the output scene is recomputed. In this way, it is very easy to write programs for showing the result of an algorithm on-line while the user is modifying the input of the algorithm, for example, by moving objects around, or by inserting or deleting objects of the input scenes.

The following piece of code shows an example program using this approach. We assume that there is a function `INTERSECT` computing the intersection points (of some type `point_t`) of a given set of straight line segments (of some type `segment_t`). Then we can create a the result scene that depends on an input scene `sc_input` of points by calling `gw.new_scene(INTERSECT,` sc_input). Many demonstration programs in LEDA and CGAL are written in this way. In particular, all algorithms working on an input set of points (e.g., all kinds of Voronoi and Delaunay diagrams) can be visualized in a single elegant program.

```
void INTERSECT(const list<segment_t>&, list<point_t>&);

int main() {
  GeoWin gw("Segment Intersection");
  list<segment_t> L;
  geo_scene sc_input = gw.new_scene(L);
  geo_scene sc_ouput = gw.new_scene(INTERSECT,sc_input);
  gw.set_color(sc_output,red );
  gw.set_visible(sc_ouput,true );
  gw.edit(sc_input);
  return 0;
}
```

**Event Handling.** Every edit operation of the interactive interface of GeoWin has an associated *event*. For instance, creating a new object triggers a *new_object* event, deleting an object causes a *del_object* event, and moving an object around creates a *move_object* event. Application programs can handle these events by specifying corresponding call-back functions which are to be called whenever a certain event occurs. We show how to use event handling in the animation of a sweep line algorithm.

The program creates a special scene sc_sweep that contains a single vertical line, the sweep line, and it associates a call-back function sweep_handler with the move_object events of this scene (by calling gw.set_move_handler (sc_sweep,sweep_handler)). Now, during the interactive mode, the user can grab and move the sweep line with the mouse, and for each motion event the sweep handler function is called, with the relative distance vector of the motion. Note that the call-back function associated with move object events has a boolean return type. The result of this function is evaluated by GeoWin and controls whether the actual motion is really executed. In the sweep example we use this fact to prevent any backward motion of the sweep line.

```
void sweep_handler(GeoWin& gw, const line& sl,
                   double dx, double dy) {
  // move sweep line horizontally by dx"
  // do not allow backward motions
  if (dx > 0) {
    sweep_x += dx;
    "process all events left of sweep_x"
  }
}

int main() {
  GeoWin gw("Sweep Demo");

  list<line> sweep_line;
```

```
  sweep_line.append(line(point(0,-100), point(0,100)));

  geo_scene sc_sweep =  gw.new_scene(sl);
  gw.set_move_handler(sc_sweep, sweep_handler);
  gw.edit(sc_sweep);

  return 0;
}
```

The screenshot of Figure 1.4 shows the window of an animation that uses the above described technique for the animation of Fortune's sweep algorithm (see [23]) for computing the Voronoi Diagram of a set of points in the plane. This animation allows to drag the sweep line across the plane while watching several different structures: the constructed Delaunay triangulation, the shore line of parabolic arcs, and the circle events of the sweep.

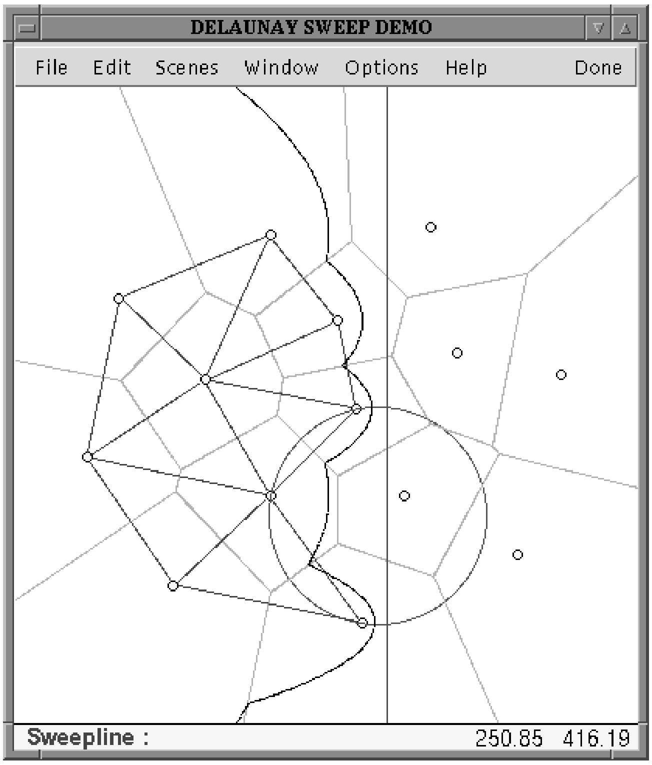



**Fig. 1.4.** Animation of Fortune's sweep algorithm with GeoWin

### 1.4.4 Animation Systems and Fast Prototyping

Many animation systems require heavy modifications to the source code at hand and, in some instances, even require writing the entire animation code in order to produce a desired algorithm visualization. Thus, a user of these systems is supposed to invest a considerable amount of time writing code for the animation but also needs to have a significant algorithmic background to understand the details of the program to be visualized. This is not desirable, especially when algorithm animation is to be used in program development and debugging. Indeed, our own experience supported the same conclusions drawn in reference [37], namely that the effort required to animate an algorithm is one of the main factors limiting the diffusion of animation techniques as a successful tool for debugging, testing and understanding computer algorithms.

In this section we address the issue of fast prototyping in algorithm animation and we show how this can be achieved by a deep use of *reusability*: in many cases, in the area of algorithm animation reusability is not considered at all, and very often the animation is so heavily embedded in the algorithm itself that not much of it can be reused in other animations. To achieve this, we need to enforce reusability in a strong sense: if the user produces a given *animated data type* (e.g., a stack, a tree, or a graph), then all its instances in any context (local scope, global scope, different programs) must show some standard graphical behavior with no additional effort at all. Of course, when multiple instances of different data types are animated for different goals, a basic graphical result may be poor without an additional, application-specific coordination effort that by its own nature seems not (and perhaps could never be) reusable. A successful approach is to offer different levels of sophistication: non-sophisticated animations should be basically obtained for free. If one wants a more sophisticated animation, for instance by exploiting some coordination among different data types for the algorithm at hand, then some additional effort should be required.

We now exemplify how fast prototyping and reusability can be addressed in an animation system by taking the example of CATAI [14]. In particular, we describe the general steps that must be followed for preparing an animation in CATAI and at the same time illustrate them through a working example: the animation of Kruskal's algorithm for computing a minimum spanning tree (MST) of a graph [30].

Kruskal's algorithm first sorts all the edges by increasing cost, and then grows a forest that will finally converge to a minimum spanning tree. Initially, the forest consists of $n$ singleton nodes (i.e., the vertices in the graph). At each step, the algorithm analyzes one edge at the time, in increasing order of their cost. If the endpoints of the edge under examination are already connected in the current forest, this edge is discarded. Otherwise, the endpoints are in two different trees: the edge is inserted into the forest (i.e., it will be a minimum spanning tree edge), and the two trees are merged into one. For efficiency issues, the trees are maintained as set union data types [48]. We refer to LEDA's implementation of Kruskal's algorithm [34], which make use of the class `partition` to implement set union data types.

While building an algorithm animation, the first decision to be taken is which data types are to be animated. In the example at hand, for instance, it seems natural to visualize the graph being explored; additionally, we could also choose to animate the underlying partition given by the set union data types. Once this has been decided, the process of developing an animation can be broken in three different steps.

**Animation libraries.** A crucial module which provides the basic tools for animation in CATAI, e.g., the graphical vocabulary, is given by the animation libraries. CATAI supplies animation libraries for most textbook algorithms: these libraries are totally independent from the data structures being animated and can be easily reused. In our example of minimum spanning trees, CATAI contains already animation libraries to represent graph objects, and thus this task is empty.

**Animated data types.** Once animation libraries are available, we need to revise the implementation of the original data types to support some animation capabilities. We call *animated classes* the classes that implement data types with support for animation: CATAI offers a specialized `C++` library to assist in the development of animated classes. The principal component of this library is the `Animator` class, which provides animation server communication primitives and binding mechanisms between a data type and the related animation library. An animated class can be derived from the original non-animated class and from the `Animator` class. These primitives map data type operators to their animated counterparts.

In our running example of minimum spanning tree, the non-animated algorithm uses the LEDA graph and partition data types. The LEDA graph class uses a single object which acts as a container to hold nodes and edges. To obtain the animated class, we derive the class `animgraph` from the LEDA `graph` class and from the `Animator` class. The methods that we wish to animate are those which change the graph: adding, removing and modifying edges or vertices. Apart from these methods, we could also add some extra methods for animation purposes.

**Animated algorithm.** We are now ready to show how to animate the implementation of Kruskal's algorithm at hand. Starting from the original code, we replace the standard graph and partition with their animated counterparts. Next, we add some animation-specific code to highlight the behavior of the algorithm.

**Original algorithm**

```
...
G = new graph();
...
list<edge> MST::KRUSKAL(graph &G){
  node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      P->union_blocks(v,w);
    }
  }
  return T;
}
```

**Animated algorithm**

```
...
G = new animgraph(sockd);
...
list<edge> MST::KRUSKAL(animgraph &G){
  anim_node_partition P(G);
  list<edge> L = G.all_edges();
  list<edge> T;

  L.sort(CMP_EDGES);
  edge e;
  forall(e,L) {
    color_edge(e, GREEN);
    node v = source(e);
    node w = target(e);
    if (! P->same_block(v,w)) {
      T.append(e);
      color_edge(e, BLUE);
      color_node(v, BLUE);
      color_node(w, BLUE);
      P->union_blocks(v,w);
    }
    else color_edge(e, RED);
  }
  return T;
}
```

For instance, we can choose to color *red* the edge which we are currently considering. If this edge will be included in the minimum spanning tree, then we will color it *blue*, and otherwise we will color it *red*. Endpoints of *blue* edges are colored *blue*, so that a forest of *blue* trees is visualized throughout the execution of the algorithm. This *blue* forest will converge to a minimum spanning tree. The resulting algorithm is proposed as a method of a container object, i.e., an MST class, and the public interface of this object will report the services (methods) that can be requested by the end-user. One snapshot of the actual animation is contained in Figure 1.5.

## 1.5 Conclusions and Further Directions

In this chapter we have addressed the role of visualization in algorithm engineering. We have surveyed the main approaches and existing tools and discussed difficulties and relevant examples where visualization systems have helped developers gain insight about algorithms, test implementation weaknesses, and tune suitable heuristics for improving the practical performances of algorithmic codes.

We believe that this can have a high impact in the way we design, debug, engineer, and teach algorithms. Yet, it seems that its potential has not been fully delivered. Citing verbatim from the foreword of [45] by Jim Foley: "My only disappointment with the field is that software visualization has not yet had major impact on the way we teach algorithms and programming or the way in which we debug our programs and systems. While I continue to believe

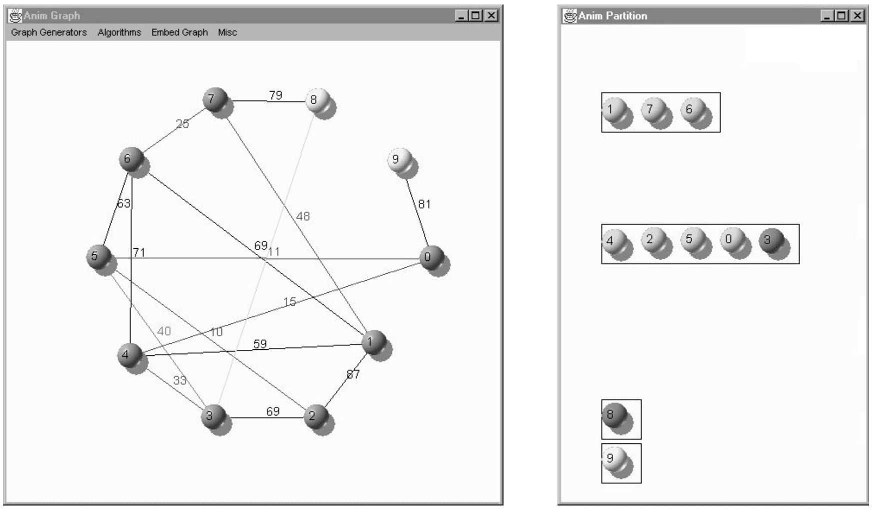



**Fig. 1.5.** Snapshot of the animation of Kruskal with CATAI: edges *(2,5)* *(0,5)*, *(4,0)*, *(6,7)*,*(4,3)* and *(1,7)* have been examined and colored blue together with their endpoints, edge *(5,3)* has been colored red, and the edge *(3,8)* is currently examined and colored green. The state of the partition is shown to the right: we have grown two blue trees (one containing vertices *0,2,3,4,5* and the other containing vertices *1,6,7*). Vertices *8* and *9* are still in singleton trees.

in the promise and potential of software visualization, it is at the same time the case that software visualization has not yet had the impact that many have predicted and hoped for."

There are many challenges that the area of algorithm animation is currently facing. First of all, the real power of an algorithm animation system should be in the hands of the final user, possibly unexperienced, rather than of a professional programmer or of the developer of the tool. For instance, instructors may greatly benefit from fast and easy methods for tailoring animations to their specific educational needs, while they might be discouraged from using systems that are difficult to install or heavily dependent from particular software/hardware platforms. In addition to being easy to use, a software visualization tool should be able to animate significantly complex algorithmic codes without requiring a lot of effort. This seems particularly important for future developments of visual debuggers. Finally, visualizing the execution of algorithms on large data sets seems worthy of further investigation. Currently, even systems designed for large information spaces often lack advanced navigation techniques and methods to alleviate the screen bottleneck, such as changes of resolution and scale, selectivity, and elision of information.

# References


1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1993.
2. A. Amenta, T. Munzner, S. Levy, and M. Philips. Geomview: A system for geometric visualization. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages C12–C13, New York, NY, USA, June 1995. ACM Press.
3. R. Baecker. Sorting Out Sorting. In *SIGGRAPH video review.* Morgan Kaufmann Publishers, 1983. 30 minute color sound film.
4. J.E. Baker, I. Cruz, G. Liotta, and R. Tamassia. Animating Geometric Algorithms over the Web. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages C3–C4, 1996.
5. R.S. Baker, M. Boilen, M.T. Goodrich, R. Tamassia, and B. Stibel. Testers and visualizers for teaching data structures. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 31, 1999.
6. A. Begeulin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing'91*, pages 435–444, 1991.
7. C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks.* Wiley, 1962.
8. M.H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.
9. M.H. Brown. Exploring algorithms using balsa-ii. *Computer*, 21(5):14–36, 1988.
10. M.H. Brown. Perspectives on Algorithm Animation. In *Proceedings of the ACM SIGCHI'88 Conference on Human Factors in Computing Systems*, pages 33–38, 1988.
11. M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 7-th IEEE Workshop on Visual Languages*, pages 4–9, 1991.
12. M.H. Brown and J. Hershberger. Color and Sound in Algorithm Animation. *Computer*, 25:52–63, 1992.
13. M.H. Brown and M. Najork. Algorithm Animation Using 3D Interactive Graphics. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 93–100, 1993.
14. G. Cattaneo, U. Ferraro, G.F. Italiano, and V. Scarano. Cooperative Algorithm and Data Types Animation over the Net. In *Proc. XV IFIP World Computer Congress, Invited Lecture*, pages 63–80, 1998. System Home Page: `http://isis.dia.unisa.it/catai/`.
15. B.V. Cherkassky. *A Fast Algorithm for Computing Maximum Flow in a Network.* In A.V. Karzanov editor, *Collected Papers, Issue 3: Combinatorial Methods for Flow Problems,* pages 90-96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Translations, Vol. 158, pp. 23–30. AMS, Providence, RI, 1994.

16. B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.
17. P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible Execution and Visualization of Programs with LEONARDO. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. Leonardo is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.
18. G.B. Dantzig. *Application of the Simplex Method to a Transportation Problem.* In T.C. Hoopmans editor, *Activity Analysis and Production and Allocation*, Wiley, New York, 1951.
19. C. Demetrescu and I. Finocchi. Smooth animation of algorithms in a declarative framework. *Journal of Visual Languages and Computing*, 2001. To appear in the special issue devoted to selected papers from the 15th IEEE Symposium on Visual Languages (VL'99).
20. C. Demetrescu, I. Finocchi, and G. Liotta. Visualizing algorithms over the Web with the publication-driven approach. In *Proc. of the 4-th Workshop on Algorithm Engineering (WAE'00), Saarbrücken, Germany. September 5-8*, 2000.
21. M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine: An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):1–66, 1988.
22. A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schnherr. The CGAL kernel: A basis for geometric computation. In *Applied Computational Geometry: Towards Geometric Engineering Proceedings (WACG'96), Philadelphia*, pages 191–202, 1996.
23. S.J. Fortune. A sweepline algorithm for Voronoi diagrams. In *2nd ACM Symp. Computational Geometry, New York*, pages 313–322, 1986.
24. J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of User Algorithms on the Web. In *Proceedings of the 13th IEEE International Symposium on Visual Languages (VL'97)*, pages 360–367, 1997.
25. M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):23–39, 1991.
26. R.R. Henry, K.M. Whaley, and B. Forstall. The University of Washington Program Illustrator. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.
27. Ch.A. Hipke and S. Schuierer. VEGA: A user centered approach to the distributed visualization of geometric algorithms. In *Proceedings of the 7-th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 110–117, 1999.
28. Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.
29. K. Knowlton. Bell telephone laboratories low-level linked list language, 1966. 16-minute black and white film, Murray Hill, N.J.
30. J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.
31. S.P. Lahtinen, E. Sutinen, and J. Tarhio. Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.
32. H. Lieberman and C. Fry. Zstep95: A Reversible, Animated Source Code Stepper. In : [45], pages 277–292.
33. A. Malony and D. Reed. Visualizing parallel computer system performance. In M. Simmons, R. Koskela, and I.s Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization.* ACM, 1990.

34. K. Mehlhorn and S. Naher. *LEDA: A Platform of Combinatorial and Geometric Computing.* Cambrige University Press, ISBN 0-521-56329-1, 1999.
35. D.R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison Wesley, 1996.
36. B.A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.
37. B.A. Price, R.M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.
38. G.C. Roman and K.C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22:25–36, 1989.
39. G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26:11–24, 1993.
40. G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.
41. J.T. Stasko. The Path-Transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.
42. J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.
43. J.T. Stasko. Animating Algorithms with X-TANGO. *SIGACT News*, 23(2):67–71, 1992.
44. J.T. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.
45. J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, Cambridge, MA, 1997.
46. A. Tal and D. Dobkin. Visualization of Geometric Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):194–204, 1995.
47. Roberto Tamassia, Pankaj K. Agarwal, Nancy Amato, Danny Z. Chen, David Dobkin, Scot Drysdale, Steven Fortune, Michael T. Goodrich, John Hershberger, Joseph O'Rourke, Franco P. Preparata, Joerg-Rudiger Sack, Subhash Suri, Ioannis Tollis, Jeffrey S. Vitter, and Sue Whitesides. Strategic directions in computational geometry. *ACM Computing Surveys*, 28(4):591–606, December 1996.
48. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. Assoc. Mach.*, 31:245–281, 1984.
49. B. Topol, J. Stasko, and V. Sunderam. Integrating visualization support into distributed computing systems. In *Proceedings of the 15-th International Conference on Distributed Computing Systems*, pages 19–26, 1995.
50. Q. Zhao and J. Stasko. Visualizing the execution of threads-based parallel pprograms. Technical Report GIT-GVU-95/01, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, 1995.