24

Graph Drawing in Education

Stina Bridgeman
Hobart and William Smith Colleges

24.1 Introduction

Illustrations are a powerful explanatory tool, so one might expect a long history of the use of graph drawing in education. This history can be traced back to at least the Middle Ages, where squares of opposition (Figure 24.1) were used as pedagogical tools in logic and other fields [KMBW02]. Murdoch [Mur84] provides examples of both basic squares and more complex structures.

In mathematics, drawings of abstract graphs began to appear as illustrations in the late 18th century, 150 years after Euler's famous paper on the Königsberg bridges launched the field of graph theory [KMBW02]. Now commonplace, hand-drawn pictures of small graphs are often used as illustrations in math and computer science textbooks to describe a graph-related concept or to explain a graph algorithm—any graph theory, discrete math, or data structures text will contain many such pictures. Drawings of graphs are also used to illustrate graph-structured information, such as the topology of a computer network or a flow chart showing a program's execution.

The introduction of computers into the classroom has led to new applications of graph drawing, including algorithm animation, algorithm simulation, exercise systems, exploration systems, program visualization, and software visualization. Section 24.2 surveys these applications, with emphasis on tools specifically developed for or used in the classroom. Many

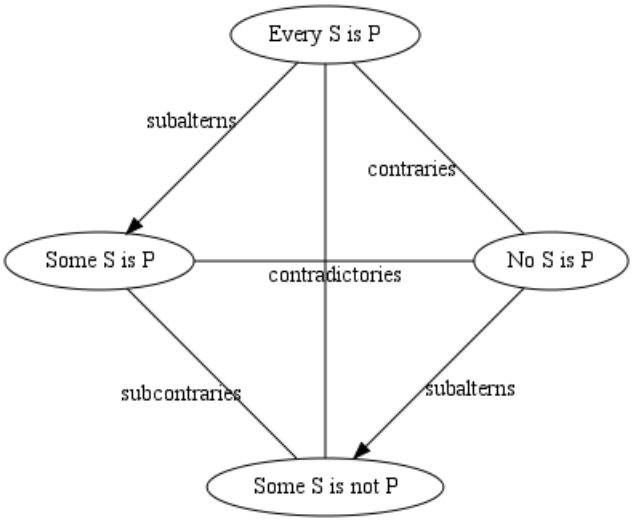


Figure 24.1 An Aristotelian square of opposition showing the relationships between the four logical forms (drawn using the `circo` algorithm from the Graphviz package [BCE+]).

of these applications place special requirements on the graph drawing algorithms used. Sections 24.3–24.6 address relevant graph drawing techniques.

24.2 Applications

24.2.1 Algorithm Animation

Algorithm animation deals with graphically illustrating the conceptual behavior of an algorithm or data structure.

Algorithm animation has been used in educational settings for many years. An early and well-known example is Baecker's 1981 video "Sorting Out Sorting" [Bae81], which animates and explains nine sorting techniques. The video illustrates how each of the sorting algorithms works by showing how bars of varying heights are gradually rearranged into increasing order, then makes an effective point about running time by showing a "race" between all of the algorithms.

In the classroom, instructor-prepared animations can be used as demonstrations during class to help explain a new concept—an animated version of the explanatory illustration. Animations used in class can be made available for students to pause, step, and replay so they can absorb the material at their own pace. Algorithm animation can also be used to engage students in the learning process—creating their own animations can deepen students' understanding of concepts, and incorporating the creation of animations into assignments can add interest to what might otherwise be a dry algorithm implementation task.

Animations of graph algorithms naturally make use of a drawing of a graph, using annotations, changing colors, or other visual effects to show the progression of the algorithm. Animations of data structure manipulations, such as inserting or removing elements from a binary tree, may also utilize a drawing of a graph or a tree. Support for automatic graph drawing frees the animation designer from having to specify the details of how the graph is drawn in each step, allowing her to focus on expressing the concept being illustrated.

Example Systems

Balsa BALSA [BS84, BS85] is one of the classic algorithm animation systems. It is a general-purpose system, designed for animating any kind of algorithm. BALSA introduced the idea of "interesting events," key points in the program where the visualization must be updated. Animations are created by implementing one or more graphical views and then augmenting the program code with calls to update those views when interesting events occur. Views are often created from scratch, though it is possible to create a reusable library of standard views. Graph layout algorithms are not provided, but can be implemented as part of a view. BALSA has been used to illustrate concepts in both mathematics and computer science courses, and for research in algorithm design and analysis.

Tango Tango [Sta90a, Sta90b] is another classic general-purpose algorithm animation system. Tango also utilizes the idea of interesting events, but provides a framework to aid in defining views. Four kinds of elements are provided as building blocks for animation scenes: basic graphical objects (shapes and text), locations of objects, transitions (movement, size, and color changes, etc.), and paths specifying how the transitions occur. Creating an animation involves three steps: defining a series of "animation scenes" (which may be a static view or an animated step), annotating the program with interesting events, and specifying the mapping of interesting events to animation scenes. Of note is Tango's support for (and emphasis on) smooth transitions between view states—many algorithm animation systems simply present a series of snapshots.

Samba Samba [Sta97] was designed to make it as easy as possible for students to create their own algorithm animations. Samba is a front-end for Polka [SK93], the successor to Tango; it reads in a command script and generates the animation from that script. Samba commands are deliberately kept simple; basic commands allow the creation of graphical objects (such as circles and lines) and the modification of existing objects (such as by moving them or changing their color). Animations are created by augmenting the program to be animated with instructions to output the Samba script. An advantage of Samba is that it does not require the animator to implement separate graphical views and link them to the code.

JAWAA JAWAA [PR98] is a web-based system for animating data structures. Animations are specified by writing a script in JAWAA's command language—unlike many algorithm animation systems, the algorithm being animated does not need to be implemented. Graphs and trees can be drawn using user-specified node positions, or can be drawn automatically using one of three built-in layout algorithms (circular layout, Tunkelang's force-directed layout [Tun94], and tree layout).

Swan Swan [SHY96] was designed specifically for visualizing graph algorithms and their related data structures. Animations are created by augmenting a C/C++ program with commands to build a graph representing the data structure to be visualized, specify visual parameters such as the shape and color of the node, and draw the graph. This means that the visualizations created are not tied to the physical representation of data structures in the program and can instead represent a conceptual view. Swan includes special "layout components" which perform automatic layout for specific types of data structures such as linked lists, arrays, trees, and general graphs. Layout components for general graphs include circular layout, Kamada and Kawai's force-directed layout [KK89], and a Sugiyama-style hierarchical layout.

24.2.2 Algorithm Simulation

Animations can only be viewed; in an algorithm simulation, a student experimenting with data structures or an instructor providing on-the-fly demonstrations in class can modify the data structure being animated or even carry out the algorithm's steps by hand.

Example Systems

Matrix Matrix [KM02] provides both algorithm animation and algorithm simulation. It can also be used to create visualizations of students' own implementations of data structures, and to perform visual testing. Multiple levels of abstraction are supported; for example, when carrying out an algorithm involving inserting an element into a balanced binary search tree, the student can perform the entire operation manually, allow the system to insert the element into the underlying binary tree but then perform rebalancing steps herself, or allow the system to do the entire insertion. Matrix provides supports several types of data structures and includes automatic tree layout. MatrixPro [KKMS04a, KKMS04b] utilizes the Matrix platform and provides a GUI tailored for instructor use in the classroom.

24.2.3 Exercise Systems

Exercise systems present students with exercises to solve and provide feedback on the students' answers. Such systems can be used for learning and practice—students can test their knowledge of an algorithm by trying exercises, and gain further understanding as the system provides feedback about their mistakes—or for assessment and grading. Key features of exercise systems include automatic generation of problem instances and automatic feedback and assessment, allowing students to practice on as many or as few problems as they wish without burdening a faculty member or teaching assistant with excessive problem-set creation or grading duties. The problem-generation feature can also be used to create individualized problem sets to help thwart cheating.

Intelligent tutoring systems also build up a model of the student's knowledge and understanding, and tailor the problems generated to address each student's individual weaknesses.

Incorporating automatic graph drawing into an exercise system is important if the system is to support graph- or tree-based problems, because each problem is randomly generated as needed.

Example Systems

PILOT PILOT [BGKT00] is a Web-based exercise system supporting trace-the-algorithm exercises. It supports automatic generation of exercise instances, feedback at each step of the tracing process, solution grading, and algorithm animation. PILOT's feedback and assessment mechanism is based on whether each step is consistent with correct execution of the algorithm at that point rather than simply checking if some final answer matches the correct solution. This allows PILOT to easily accommodate cases with multiple correct solutions and to provide meaningful feedback and reasonable partial credit when a single mistake is made early in the process. Several graph-based problems including minimum spanning tree, breadth-first and depth-first search, and shortest path algorithms have been implemented. Force-directed and hierarchical drawing algorithms provided by the Graph Drawing Server [BGT99] are used for graph layout.

TRAKLA2 TRAKLA2 [MKK+04] is a Web-based exercise system built on the Matrix [KM02] algorithm animation and simulation framework. Like PILOT, TRAKLA2 supports trace-the-algorithm exercises and includes automatic generation of exercise in-

stances, solution grading, and animation of model solutions. Evaluation of a student's answer is limited to comparing the student's solution to a model solution, and the student only receives notice of how many steps were correct. However, TRAKLA2 contains a number of features making it useful for coursework including storage of students' grades and submitted answers, deadlines for exercises, and the ability to control whether the same instance of an exercise may be repeatedly submitted for feedback (a practice exercise) or if it must be reset with new input data each time (a graded assignment). Exercises involving a variety of data structures and algorithms, including graph algorithms, have been implemented.

AnimalSense AnimalSense [RMS11] takes a different approach. Instead of providing an environment where students manually trace the execution of an algorithm, AnimalSense supports questions that provide evidence of successful algorithm-tracing such as "Give the sorted order" or "Give your third chosen edge." This approach allows greater latitude in the types of exercises that can be supported — it can also accommodate questions like "Provide an array which uses 4 pivots to be sorted," which go beyond simply tracing and which require deeper thinking about the functioning of an algorithm. Algorithm animation is provided to aid in solving the problem and to help reveal the cause of a mistake. Exercises involving graph algorithms, searching algorithms, and sorting algorithms have been implemented.

24.2.4 Exploration Systems

Exploration systems support experimentation with graph structures and graph theory concepts. In the classroom, exploration systems can be used in a professor-led discussion to illustrate or animate examples or algorithms, or for student exploration or experimentation.

Support for automatic graph drawing frees the experimenter from having to find a reasonable layout, and can be important in revealing the structure of the graph being studied.

Example Systems

LINK LINK [BDG+00] is designed for education and research in discrete mathematics. It consists of a library of templated C++ classes for graphs and other data structures coupled with an interactive front-end for animation and visualization. The library also contains a collection of graph algorithms commonly used as building blocks and which are often covered in their own right in graph theory and computer science courses. To aid in visualization, LINK includes several simple graph layout algorithms (place vertices randomly, on a circle, or on a grid), a spring embedder, and several algorithms suited for particular applications (e.g., illustrating the results of a depth-first search, drawing the graph as a bipartite graph, and laying out each biconnected component of the graph separately to emphasize the components) [Ber].

GraphPack GraphPack [KOD+96] is a tool designed for experimenting with graphs and graph algorithms. It supports several 3D and 2D graph layout algorithms, contains a graph viewer, and can integrate functionality from other packages such as Mathematica, Maple, and Matlab. A novel feature is its ability to extract the graph structure from a black-and-white bitmap image of a drawing.

24.2.5 Program Visualization

Program visualization deals with visualizing a program's actual execution rather than a high-level conceptual view of an algorithm. Aspects of the program being visualized can

include source code, data structures, and runtime behavior. Program visualization can be used to illustrate the functioning of an algorithm or data structure (as in algorithm animation), to gain an understanding of how the program works, to aid in debugging, and to evaluate and improve program performance.

In the classroom, program visualization can help students learn to program and debug by revealing what their programs are actually doing. This is more effective than systems which attempt to explain a bug (because explanations require understanding the underlying concept in the first place), try to guide the student to a particular way of solving the problem (ignoring other valid solutions), or are limited to a small set of toy problems [EPD92]. For more advanced students, visualizations can help explain the underlying semantics of the programming language, design patterns, and the workings of multithreaded programs [GJ05]. As with algorithm animation, instructors can also use program visualization to spice up an implementation assignment.

Automatic graph drawing is an essential component for program visualization systems which display graph-structured information because the particular graph depends on the runtime state of the program.

Example Systems

A simple form of program visualization—and one that is also suitable for algorithm animation—is to display graphical snapshots of the key data structures whenever the state of the structure changes.

GraphTree/GraphHeap Owen's GraphTree and GraphHeap subroutines [Owe86] were designed as a low-overhead animation system for illustrating binary tree and heap operations. The subroutines take the data structure to be visualized as a parameter, and are called when the animator wants to produce a graphical snapshot of the current state of the tree or heap. A simple layout algorithm is used: parent nodes are centered above their two children, with empty spaces for missing child nodes.

VisualGraph VisualGraph [LNR03] is a Java graph class which provides typical graph querying and manipulation operations, as well as visualization operations (highlighting and changing the color of edges and vertices) and related utility routines (random graph generation, graph layout using the force-directed method of Kamada [Kam89], and file I/O). Simple visualizations are created by augmenting the program code with calls to "print graph" whenever a picture of the current state of the graph is desired. VisualGraph is implemented as a front-end to an algorithm animation system—animation operations produce output in the AnimalScript language [RF01], which can then be read and displayed by a system such as JHAVÉ [NEN00].

Visualiser Naps' Visualiser class [Nap98] supports multiple data structures, including trees and graphs. It parses a string representation of the data structure to be visualized rather than working directly with particular Java objects, so it can be extended to new implementations of data structures by providing a new "to string" routine. New data structures or visualization styles can be supported by adding new Visualiser subclasses.

JDSL Visualizer The JDSL Visualizer [BBG+99] does not require users to modify their code to generate visualizations of data structures, as snapshots are automatically generated before and after data structure operations. However, data structures must be implemented to a particular API. Several linear and binary-tree-based structures are supported.

LJV The Lightweight Java Visualizer (LJV) [Ham04] uses Java's reflection mechanism to determine the structure of a Java object and is thus suitable for use with any Java program. Visualizing an object requires only adding calls to a "display object" rou-

tine when an object is to be visualized. The resulting graph structure is drawn using GraphViz [BCE+]. More advanced users or instructors setting up the tool for a course can customize the appearance of particular classes, such as to hide the internal representation of the `String` class. Of note is that because the structure is derived directly from the object itself, both correct data structures and students' incorrect ones can be visualized. The tool is also effective for demonstrating aspects of the Java language which often cause confusion, such as the pervasive but hidden use of references and the meaning of static fields.

Other systems provide visualization of data structures without requiring the program to be modified.

UWPI The University of Washington illustrating compiler (UWPI) [HWF90] analyzes program source code (written in a subset of Pascal) and automatically constructs a visualization of the data structures used in the program. UWPI attempts to infer the abstract data type of each variable from its concrete data type and usage patterns in order to determine an appropriate visualization. Supported ADTs are numbers, arrays, and digraphs; graphs are converted to directed acyclic graphs and drawn using the methods of Sugiyama, Tagawa, and Toda [STT81] and Rowe et al. [RDM+87].

jGRASP jGRASP [HCIB04] is a Java development environment combining a debugger and visualization tools. Data structures to be visualized are extracted automatically from the program; "external viewers" specify how to render a visual representation of an object of that type. This architecture allows multiple views of a single data structure to be displayed simultaneously. New external viewers can be added, so the system can be used for creating animations as well for debugging. More recent versions of jGRASP include a "Data Structure Identifier" which automatically identifies the data structure being visualized and suggests appropriate viewers [CIHJB07]. jGRASP uses FLGL, a graph drawing library based on VCJ [MB98], to produce layouts of data structures. VCJ includes Walker's algorithm [Wal90] for drawing rooted trees, Kamada and Kawai's spring embedder [KK89] for undirected graphs, and clan-based graph drawing [MCS98] for directed graphs.

Program visualization can include visualization of more than just data structures.

Jeliot 3 Jeliot 3 [MMSBA04] is the fourth system in a series of program visualization systems designed for beginning programmers. Jeliot displays both object structures and control flow, providing a fine-grained animations of every step of the program's execution, including the evaluation of expressions. One drawback is that Jeliot does not support the full Java language.

JIVE JIVE [GJ05] is designed for the visualization of object-oriented programs (specifically, Java) and shows objects not just as data structures but also as execution environments. JIVE's views show an object's fields and its methods, structural links between objects, and the history of the method calls made as the program runs. This approach reveals much more of how Java actually works than data structure visualization approaches, and helps the viewer more thoroughly understand what is really going on when the program is run. Streib and Soma [SS10] discuss experiences using JIVE and the contour diagrams used by JIVE in introductory programming courses.

24.2.6 Software Visualization

The field of software visualization encompasses the visualization of all aspects of a software system, including its structure, execution, and evolution over time. Graph drawing plays an important role in software visualization as many aspects of a software system can be rep-

resented using graphs, including control-flow graphs, program call graphs, class diagrams, and dependency graphs.

While there has been a great deal of work in the field of software visualization, many of the software visualization tools designed for use in the classroom focus on algorithm animation or program visualization. BlueJ, described below, is one exception.

Example Systems

BlueJ BlueJ [KQPR03] is an integrated development environment (IDE) developed for the teaching of Java programming. BlueJ emphasizes class structure and design through UML class diagrams—a class diagram is displayed in the main window when a project is opened, and it is through the diagram that students can edit, compile, and create instances of classes.

24.3 Graph Drawing for Algorithm Animation

In algorithm animation (and in many program visualization applications), an abstract view of the data structure is both sufficient and desired. In many cases, standard graph drawing algorithms are suitable for this task. The most important criteria for drawings are following familiar conventions (such as placing the root of a tree at the top or directing edges downward) and readability, properties which are easily achieved by many standard algorithms. Examples of suitable algorithms include Walker's algorithm [Wal90] for rooted trees, Sugiyama-style layout [STT81, GKNV93] for directed graphs, and force-directed methods (e.g., Kamada-Kawai [KK89] and Tunkelang [Tun94]) for general graphs.

24.3.1 A Unified Approach to Drawing Data Structures

One drawback to using standard algorithms is that different algorithms must be chosen for linked lists, trees, directed graphs, and general graphs. For applications such as visual debuggers, which need to be able to visualize any data structure (including buggy or ill-formed ones) and where the type of data structure is not known in advance, a unified approach is needed.

Since data structure graphs are directed graphs and convention often places the root of the structure at the top, a hierarchical layout is a natural layout style for drawings of data structures. The classic Sugiyama algorithm [STT81] for producing a hierarchical layout of a directed graph consists of five phases:

- Cycle removal: If the graph to be drawn is not acyclic, one or more edges must be reversed in order to remove all directed cycles.
- Layer assignment: Nodes are assigned to layers, where all nodes on the same layer will have the same y-coordinate in the final drawing. Dummy nodes are inserted as needed so that edges only connect nodes on adjacent layers.
- Crossing reduction: The nodes in each layer are rearranged so as to reduce edge crossings between layers, typically through repeated passes in which the ordering of one layer is held fixed while the nodes in an adjacent layer are rearranged. One strategy for rearranging nodes is to sort them according to the average position of the adjacent nodes in the other layer (barycenter method).
- Coordinate assignment: The nodes in each layer are assigned x-coordinates, preserving the left-to-right ordering of each layer.

- Edge routing: Edges are commonly drawn as polylines, with bends introduced by the placement of dummy nodes. However, other routing strategies (such as splines [GKNV93] and edge bundling [PNK11]) have been introduced.

Constraints can then be added to respect specialized conventions for drawing particular kinds of data structures. Waddle [Wad01] identifies three types of constraints as the most important for data structures: "same-level" constraints defining nodes which must appear on the same level, left-to-right ordering constraints between nodes or paths, and edge-orientation constraints which preference edges for reversal during cycle removal. Adapting the Sugiyama algorithm to accommodate these constraints will be discussed below.

Same-Level Constraints

Same-level constraints may result in edges connecting nodes in the same level. Traditional layer assignment prevents same-level edges, and furthermore same-level edges cannot be handled by the traditional compute-barycenters-and-sort crossing reduction method. (Sorting requires a fixed barycenter for the duration of the sort, but the barycenter of a node with same-level neighbors will change as the neighbors are rearranged during the sorting process.)

Waddle's solution is a two-tier layer assignment and crossing reduction strategy. First, same-level constraints are used to define equivalence classes of nodes that must appear on the same level and layer assignment is performed using a single proxy node in place of each equivalence class. "Virtual layers" are then created within each layer and layer assignment is repeated for each equivalence class using the virtual layers. This results in nodes involved in same-level constraints being assigned to different virtual layers. During crossing reduction, a layer containing virtual layers is sorted by applying the usual crossing reduction procedure to the virtual layers.

Finally, all nodes within a layer (regardless of virtual layer) are assigned the same y-coordinate and same-level edges are routed around intervening nodes as needed. Böhrigner and Paulisch [BN90] add an additional constraint that same-level edges must connect consecutive nodes in order to avoid the need for edge routing.

Node Ordering Constraints

Node ordering constraints specify the left-to-right ordering of pairs of nodes in the same level. (After layer assignment, path ordering constraints can be converted to node ordering constraints involving pairs of nodes and dummy nodes along the extent of the paths.) Node ordering constraints are implemented in the crossing reduction phase.

A simple strategy for respecting node ordering constraints is to proceed with sorting nodes by their barycenters, but to disallow any swaps which would violate the ordering constraints.

Waddle [Wad01] uses a different strategy: the ordering constraints are checked after the barycenters have been computed and, if a constraint is violated, a new barycenter is assigned which places the node just to the right of the rightmost node which must precede it according to the constraints. The nodes are then sorted according to their revised barycenters.

Both of these approaches are fast and result in an ordering which satisfies the constraints, but may result in a large number of avoidable crossings.

A third strategy is the "penalty graph" approach [Fin01], which produces fewer crossings at the expense of a more complex algorithm and a higher running time. In this approach, the penalty graph contains the nodes of the layer to be reordered. A directed edge (u,v) indicates that placing u to the left of v results in fewer crossings than placing v to left of u. The weight of the edge (u,v) indicates by how much the number of crossings is improved.

An ordering constraint requiring u to be to the left of v can be imposed by assigning the edge (u,v) an infinite weight. The ordering of the layer is determined by applying a heuristic to find the minimum-weight set of arcs whose removal makes the penalty graph acyclic (the minimum weighted feedback arc set problem), and then performing a topological sort of the resulting acyclic penalty graph.

Forster [For04] gives a heuristic which combines the efficiency and simplicity of the barycenter approach with the quality of the penalty graph method. First, barycenters are computed for each node. Then, for each violated constraint, the nodes involved are replaced by a single proxy node and a new barycenter is computed for the proxy node based on the combined neighbors of the original nodes. Once all of the constraints have been accommodated, the nodes and proxy nodes are sorted by their barycenters. The final sorted layer is obtained by replacing each proxy node with the ordered collection of individual nodes that were grouped together.

Constraints must be considered in the correct order when creating proxy nodes or else it can become impossible to satisfy all of the constraints. The constraints to be satisfied can be represented by a constraint graph, which contains a directed edge (u,v) for each constraint of the form "u must be placed to the left of v." The next constraint to consider can be found by performing a topological sort of the constraint graph; as each node is visited, its incoming constraints are considered in reverse traversal order. The first violated constraint encountered is the next one to collapse into a proxy node. The constraint graph must be updated and the traversal restarted after each proxy node is created.

Forster's heuristic is based on the assumption that if the barycenter ordering causes vertex v to be placed to the left of u in violation of an ordering constraint, no vertices would be placed between u and v in the optimal solution with the correct ordering (u left of v). Though counterexamples can be easily found, the heuristic gives results that are nearly as good as the penalty graph approach in much less time.

Edge-Orientation Constraints

Since layer assignment requires an acyclic graph, the cycle removal phase reverses the direction of one or more edges in order to remove directed cycles. For some data structures, such as doubly-linked lists or trees where each node has both "child" and "parent" pointers, arbitrarily selecting edges for reversal may result in drawings that violate standard drawing conventions or have inconsistent edge orientations.

Waddle [Wad01] addresses the problem by tagging edges which may be reversed during cycle-breaking in the layer assignment phase. These edges will be reversed first, before untagged edges.

24.3.2 Special-Purpose Layouts

Space is a powerful visual variable, and an animation designer may choose to devise a custom layout algorithm which makes more effective use of space than a general-purpose algorithm. For example, Brown and Sedgewick [BS85] discuss the design of an animation involving binary search trees: noting that the simple recursive strategy of devoting half of the width of the current region to each of the left and right subtrees quickly leads to crowding even in trees of the size typically used in examples, they instead base the x coordinate of a node on the node's position in an in-order traversal of the tree. This ensures that each subtree has a width proportional to the number of nodes in that subtree, and also helps reinforce the organizational structure of the tree.

24.4 Graph Drawing for Program Visualization

Many program visualization applications focus on visualizing the objects in memory. These objects, along with their references to other objects, naturally form directed graphs.

Standard drawing algorithms for directed graphs can be used to produce layouts for object graphs. However, program visualization applications may have requirements that are not well-served by standard drawing algorithms. The rest of this section addresses specialized drawing techniques relevant for program visualization.

24.4.1 Complex Node Structures

Objects in programs are complex structures with multiple fields. Seeing this internal structure can be important for understanding the program's behavior, particularly in debugging applications.

The convention when drawing object structures is to show pointers or references as edges which end at distinct points inside the node. This can pose problems for standard drawing algorithms. For example, traditional crossing-reduction strategies used by Sugiyama-style layout algorithms assume that edges connect node centers and thus crossings can only occur between edges connecting different pairs of nodes. With complex nodes, edges may originate and terminate at any point within a node, and crossings can occur even when two edges are incident on the same node.

Waddle [Wad01] uses a Sugiyama-style approach for drawing object graphs, and accommodates complex nodes by using the coordinate of the edge's actual endpoint within the node instead of the node's center when computing barycenters for crossing reduction. Problems can still arise if a node contains several edges whose endpoints are vertically aligned because the adjacent nodes may end up with the same barycenter—and improper ordering of those nodes can result in edge crossings. This is addressed by assigning a secondary sort key (or "secondary barycenter") based on the vertical ordering of the endpoints. Figure 24.2 shows two ways to assign secondary barycenters.

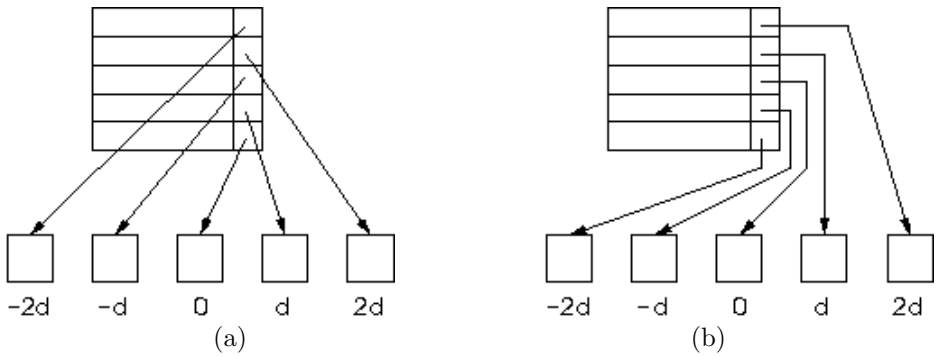


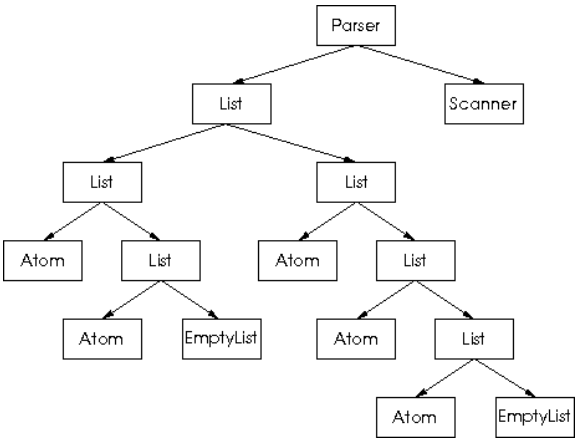
Figure 24.2 Two strategies for assigning secondary barycenters. The value of the secondary barycenters are shown below the nodes ($d > 0$). (a) Drawing with edge-node overlaps. (b) Drawing that avoids edge-node overlaps but involves additional edge routing.

24.4.2 Taking Structure into Account

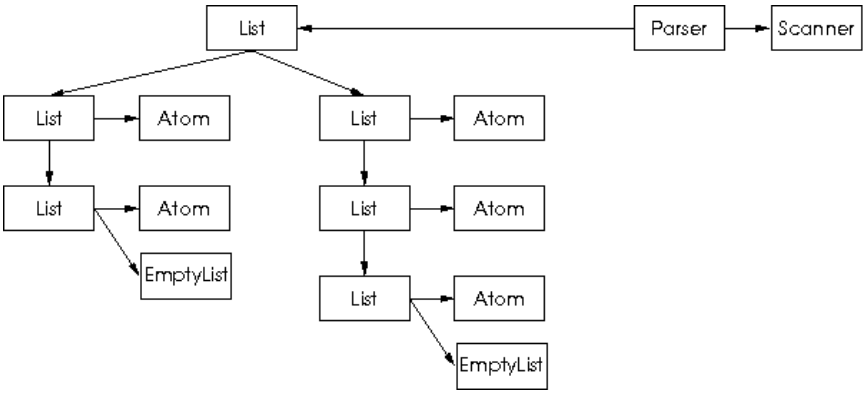
Not all of the nodes in the object graph serve the same purpose — some are part of a data structure, such as a binary tree, while others are data fields. With this in mind, Gestwicki et al. [GJG04] identify two important aesthetic criteria for drawing object graphs:

- Leaf objects, which have exactly one incoming reference and no outgoing references, should be grouped with the objects (called aggregators) that reference them.
- Recursive structures should be clustered.

Figure 24.3 illustrates the benefits of this approach.



(a)



(b)

Figure 24.3 (a) Object graph for a simple expression parser drawn using a traditional Sugiyama-style layout algorithm. (b) Drawing taking the class structure into account. Example from [GJG04].

Gestwicki et al. [GJG04] use the program's class diagram to identify the important structures. A leaf class is a class with no outgoing associations—all of its fields, including inherited fields, are either primitive types or immutable wrappers around primitive types. A recursive type is defined by a directed cycle along generalization and aggregation relationships in the class diagram—all of the classes along the cycle are part of the recursive type. The simplest case is a single class containing a field of its own type.

The leaf classes and recursive types identified in the class diagram can then be used to identify interesting structures in the object graph. A leaf cluster consists of an aggregator node and its leaf-class children. (Note that an aggregator node may have other children in the object graph that are not part of the leaf cluster.) A recursive cluster is a connected subgraph containing objects belonging to a single recursive type and their leaf-class children, and with at most one node with incoming edges from outside the cluster.

Once the leaf and recursive clusters have been identified, the graph is drawn in three steps:

- Draw the leaf clusters.
- Replace the leaf clusters by single nodes, and draw the recursive clusters.
- Replace the recursive clusters by single nodes, and draw the remaining structure.

In order to avoid needlessly complicating the drawing with unnecessary detail, only nodes whose type is included in the class diagram are drawn. A variety of algorithms can be used in each stage, though the drawing algorithms chosen for the last two steps must be able to take into account the area needed to draw the collapsed cluster nodes. Using different layout techniques for each cluster, such as a radial layout for leaf clusters and a hierarchical layout for recursive clusters, emphasizes the distinct nature of each type of cluster.

The advantage of deriving leaf and recursive clusters from structures in the class diagram instead of basing them solely on the object graph is that the final drawing will reflect the correct semantics of the program—it will not be dependent on the current state in the program's execution. Consider, for example, the definition of a leaf cluster—it distinguishes between nodes in the object graph which currently have no outgoing edges and those which will never have any outgoing edges.

24.4.3 Drawing Execution Environments

Many program visualization systems show objects only as containers for data, but JIVE [GJ05] aims to give a more comprehensive view of the execution of object-oriented programs by showing objects both as containers for data and as environments for execution. In the most detailed view, objects are shown with both fields and methods; each active method is shown with its parameters and local variables. This structure may be multiple levels deep as contained objects may themselves contain fields and active methods. Inheritance relationships are also shown so each object's scope is clear. JIVE's object graphs present a challenge for graph drawing, as the graphs have large nodes containing complex internal structures, nested structures, multiple types of nodes and edges, and edges which connect to internal points within nodes.

The nested structure of objects-within-objects can be represented as a tree, and the object can be drawn by creating an HV-inclusion drawing of the nesting tree. In this drawing style, child nodes are drawn as rectangles within the rectangle devoted to the parent and are either arranged in a row or stacked vertically. Garg et al. [GGJ06] give a dynamic programming algorithm for computing minimum-area HV-inclusion drawings.

The rest of the graph structure in the object graph can be drawn using an algorithm for layered drawings of weighted multigraphs [GJ05].

24.4.4 Drawing Sequence Diagrams

In addition to displaying object graphs, JIVE [GJ05] uses a sequence diagram to show the program's execution history. In a sequence diagram, each method activation is represented by a vertical bar and all of the method activation bars belonging to a single object are drawn along the same vertical line. Method calls and returns are represented by arrows drawn from one activation bar to another.

Drawing sequence diagrams can be formulated as a graph drawing problem. A sequence graph contains a node for each method activation bar and a directed edge for each method call and return; the task is to find a left-to-right ordering for the object lines which minimizes edge length, the number of edges crossing activation bars, and the number of method-call edges directed to the left. Clustering constraints may also be applied to ensure that object lines for related objects are close together.

Garg et al. [GGJ06] give a simulated annealing algorithm for finding a left-to-right ordering of object lines which respects the desired clustering and optimizes an objective function incorporating the aesthetic criteria. Each object line is assigned a unique integer value; lines belonging to the same cluster receive consecutive labels. Two object lines are selected randomly and, with a probability related to the potential improvement in the objective function and the temperature of the system, the integer labels of either the lines (if the object lines belong to the same cluster) or the clusters (if the object lines belong to different clusters) are swapped.

24.5 Graph Drawing for Software Visualization

24.5.1 Drawing UML Class Diagrams

One challenge in drawing UML class diagrams is handling the multiple types of edges—generalizations and associations—because generalizations are hierarchical and associations are not. In addition, Purchase et al. [PAC01] have identified several aesthetic criteria that are important for UML class diagrams, including orthogonality, a consistent orientation for the edges, and joined inheritance arcs instead of separate edges. The traditional aesthetic criteria of few crossings and bends are also important.

Two-Pass Approach

Seemann [See97] prioritizes showing the different types of relationships over the other aesthetic criteria. A two-pass strategy is used: first the inheritance hierarchies are drawn using a variation of the Sugiyama algorithm, and then the association edges are drawn with an orthogonal style.

In the first phase of the algorithm, a modified Sugiyama layout is applied to just the generalization edges and their incident vertices. The initial layer assignment is adjusted to reduce the span of association edges: if a node has an association with a node in a lower layer, and moving the node to the lower layer does not violate the desired direction of any generalization edges, the node is moved. In addition, the crossing reduction stage is modified to attempt to place nodes with association edges between them next to each other in the layer.

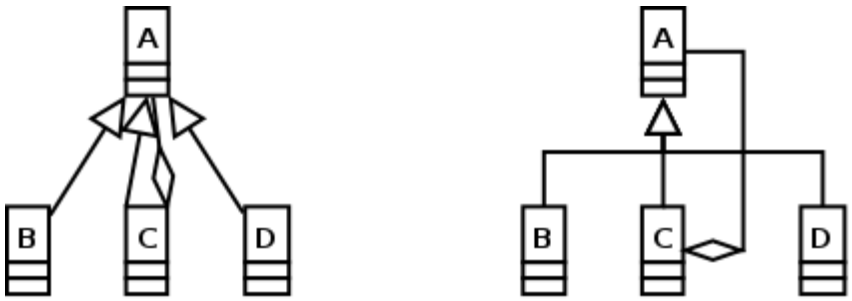
Next, the remaining nodes are placed into levels. New nodes are added incrementally; in each pass, nodes which have not yet been placed but which are adjacent to nodes which have been placed are added. Let v be an already-placed node and S be the set of to-be-placed nodes adjacent to v. If $|S| \leq 2$ and the already-placed nodes to either side of v are not adjacent to v, the nodes of S can be placed to the right and left of v in v's layer. If there is not enough room to place the nodes of S next to v—either because S is too large, or v is already connected to the nodes next to it—the nodes of S are placed on a sublayer above or below v's layer. Once all of the nodes have been placed, the sublayers are used to further reduce crossings and bends due to assocation edges connecting non-consecutive nodes on a layer.

Finally, node sizes are computed, edges are routed, and x coordinates are calculated. Node sizes are based on the information that must be displayed inside the node. Generalization edges are drawn as straight lines, with connection ports evenly spaced along the bottom or top of a node. Association edges are drawn with an orthogonal style; to route edges connecting nodes in different layers, dummy nodes representing bends in the edge are added on one side of the nodes being connected. These dummy nodes are constrained to stay vertically aligned when x coordinates are assigned. Connection ports for association edges are evenly spaced along the left or right side of a node.

Integrated Approach

Gutwenger et al. [GJK$^+$03] give a more complex drawing algorithm which respects all of the aesthetic criteria identified by Purchase et al. [PAC01] and additionally ensures that all generalization edges within the same class hierarchy are oriented in the same direction, generalization edges in different hierarchies do not cross, and hierarchies do not contain each other. The algorithm follows the topology-shape-metrics approach [DETT99]: first the graph is planarized, then the bends and angles are fixed, and finally edge lengths are computed.

Because the convention that inheritance arcs are drawn joined can result in additional crossings (Figure 24.4), the planarization phase begins with a preprocessing step which adds a new vertex for each join point. Consistency of direction of edges within a hierarchy is achieved by computing an upward planar representation for each class hierarchy, and separation of different hierarchies is achieved by treating each hierarchy as a cluster and applying a cluster planarization algorithm.



(a) (b)

Figure 24.4 (a) A planar embedding. (b) With the same embedding, joining the generalization edges results in a crossing.

In the shape phase, vertices with degree greater than four are replaced by a "cage" containing a cycle of degree-3 vertices prior to computing an orthogonal representation.

Finally, two compaction steps are used in the metrics phase. After the first compaction step, the cages are replaced by the original high-degree vertices. Because the cages may be larger than the vertices they contain, additional bends may be needed in order to route edges within the cage. The second compaction step addresses this problem and removes unnecesssary bends.

24.6 Sequences of Drawings

Both algorithm animation and program visualization often involve graphs whose structure changes over the course of the visualization. In these cases, it is important to preserve the user's *mental map* [ELMS91]—that is, to maintain a degree of layout stability so the viewer can focus on what is really going on in the algorithm or program without being distracted by the side-effects of the layout algorithm. However, many standard layout algorithms assume complete freedom over the placement of nodes.

There are many models for the user's mental map. The most rigid is the "no change" model, where existing portions of the drawing are preserved exactly (e.g. [MHT93, PT98]). Böhringer and Paulisch [BN90] limit change to nodes within a certain graph distance of those directly affected by an update. Other strategies seek to preserve absolute vertex position, but allow some movement (e.g. [LMR98]). Misue et al. [MELS95] seek more generally to preserve the shape of the drawing and give several models for the mental map based on orthogonal ordering (the relative up/down/left/right relationships between nodes), proximity (nodes near each other should stay near each other), and topology (specifically, the dual graph). Specific metrics for measuring mental map preservation are given by Lyons et al. [LMR98], Bridgeman and Tamassia [BT98], and Brandes and Wagner [BW98]. Time can also be a factor, with the idea that it is more costly to the user's mental map when long-stable portions of the drawing are changed instead of relatively new sections [BW97].

Preserving the mental map typically leads to a tradeoff with drawing quality. Algorithms which more rigidly preserve the original layout result in drawings which are less good according to traditional aesthetic criteria such as drawing area, crossing minimization, and bend minimization. Some dynamic graph drawing algorithms allow user control over the relative weight given to each goal.

An overview of dynamic graph drawing and its application in several drawing paradigms is given by Branke [Bra01]. This section will address some strategies for maintaining layout stability within the drawing paradigms most useful for data structure and program visualization.

24.6.1 Trees

A "no change" algorithm for binary trees is simple: recursively draw the left subtree in the left half of the available space and the right subtree in the right half of the available space, and center parent nodes above the drawings of their subtrees. GraphTree and GraphHeap [Owe86] use this approach. The drawback is an overly-wide drawing and wasted space if the tree is not complete or nearly complete.

Moen [Moe90] gives an algorithm for general trees which makes better use of space and does not change the drawings of subtrees not affected by updates. In addition, the algorithm can accommodate nodes with any polygonal shape—an advantage for data structures with complex nodes. The algorithm is based on computing a contour around each subtree,

which is then used to pack subtrees together as closely as possible. Contours are computed recursively. Making changes to the tree structure requires recomputing contours (only) for the subtrees containing the affected nodes.

A similar approach is used by Workman et al. [WBP04], with the drawing convention that trees are laid out horizontally (children next to parents instead of below) and parents are placed on the same level as the first child.

24.6.2 Force-Directed Layout

In the force-directed model, layout stability is most commonly achieved by incorporating additional forces into the model. Varying the strength of the stability forces provides a convenient way to balance layout stability and drawing quality.

Absolute vertex positions can be maintained by adding forces that attract nodes to their former positions [LMR98, BW97].

Relative distances between nodes can be maintained by adding springs whose natural length is the desired distance [BW97]. Stiffening the springs makes the distances more rigid. Stiffening entire subgraphs can help maintain the shape of the drawing.

Clustering can be maintained by adding attractive forces toward the center of the cluster and repulsive forces between clusters [Tam98].

It is also possible to incorporate some hard constraints. For example, Tamassia [Tam98] mentions truncating a node's movement each time forces are applied in order to keep it within the desired region. In addition, the shape of a subgraph can be preserved exactly (up to translation and rotation) by treating it as a single rigid body when computing forces.

24.6.3 Sugiyama-Style Hierarchical Layout

Within the Sugiyama framework, several basic approaches can be used: incremental techniques, in which the existing drawing is modified to accommodate the changes; constraint-based techniques, in which a new layout is computed subject to constraints meant to preserve the user's mental map; and cost-based techniques, in which stability is encouraged by assigning a cost to changes that affect the user's mental map.

Incremental Techniques

North [Nor96] describes an incremental heuristic for maintaining both geometric (position) and topological (ordering) stability in Sugiyama-style layouts. It is assumed that changes are made to the graph one at a time, so the algorithm only needs to accommodate the addition or removal of a single node or edge.

A new node is assigned to the highest possible level consistent with maintaining a downward orientation for edges, and existing nodes are shifted to lower levels as needed. New level assignments are determined by depth-first search. Nodes are moved downward one level at a time. At each step, the node is shifted into its correct horizontal position in the level according to the median of its neighbors' positions. Finally, a linear program is used to assign horizontal coordinates to the nodes. An additional cost is introduced to penalize moving nodes to new positions.

Constraint-Based Techniques

Böhringer and Paulisch [BN90] maintain layout stability by adding constraints to maintain the level assignment of nodes and the ordering of nodes within a level. When a node or edge is added or removed, constraints are weakened (more likely to be deactivated

in the case of contradictory constraints) or removed in the vicinity of the changes. They define "vicinity" in terms of graph distance, but other notions (such as Euclidean distance in the drawing) could be used.

Waddle's algorithm for drawing data structures [Wad01] also handles layout stability by adding constraints. He focuses on maintaining the relative ordering of subgraphs rather than fixing the layer assignment, adding

- node-ordering constraints between root nodes in the top level,
- edge-ordering constraints between edges incident on root nodes, and
- edge-ordering constraints between downward edges.

New elements added to the graph are initially unconstrained; constraints which are rendered invalid by the removal of elements are updated or deleted. Böhringer and Paulisch's [BN90] scheme of weakening constraints in the vicinity of changes could also be applied.

Section 24.3.1 outlines how the basic Sugiyama algorithm can be modified to accommodate these constraints.

Cost-Based Techniques

North and Woodhull [NW02] reduce the layer assignment and coordinate assignment phases to integer linear programs. Layout stability can be incorporated by adding terms to the objective functions to penalize movement to a different layer (layer assignment) or a different position (coordinate assignment). An advantage of this approach is that the tradeoff between drawing quality and layout stability can be managed by adjusting the cost of movement.

In the crossing reduction phase, median sort and transposition sort are used to reduce crossings. Only new or modified nodes and edges and edges incident on new or modified nodes are considered to be movable during sorting.

24.6.4 Offline Dynamic Graph Drawing

In "canned" animations, both the graph and the sequence of changes being made to the graph are known in advance. In this situation, an offline drawing algorithm—which takes into account future graph states when producing a layout—can be used to increase layout stability.

Force-Directed Layout

Erten et al. [EHK+04] combine the individual snapshot graphs into a single aggregrate graph, adding edges between corresponding vertices in different snapshots. At a minimum, between-snapshot edges should be added between corresponding vertices in consecutive snapshots. Global layout stability can be increased by adding edges between more distant snapshots.

The aggregate graph is drawn using the Kamada-Kawai algorithm [KK89], modified so that there are no repulsive forces between vertices in different snapshots. The balance between layout stability and readability can be controlled by adding weights to the between-snapshot edges. To accommodate weights, the Kamada-Kawai forces are modified to use the ideal distance between vertices (based on the weights of edges between them) instead of the graph distance between them.

Foresighted Layout

Diehl, Görg, and Kerren [DGK01] give a more general strategy which they call "foresighted layout." In the simplest case, the individual snapshot graphs are combined to create a supergraph containing every node and every edge present in at least one individual graph. A layout is then computed for the supergraph, and each individual graph is drawn using the subset of the supergraph layout information.

A drawback to this approach is that the supergraph can be quite large if the graph structure changes significantly over the course of the animation, leading to wasted space in the individual layouts. To save space, a reduced version of the supergraph in which nodes and edges with disjoint "live times" are grouped together is used instead. (Since elements with disjoint live times do not occur in the same snapshot graph, they can occupy the same position in different snapshots.)

As a final step, layout adjustment strategies can be used to improve the quality of each snapshot layout at the expense of layout stability [DG02].

24.6.5 Smooth Animation

When viewing a series of drawings, animation can be used to help the viewer see and understand what has changed from one drawing to the next.

A simple scheme is to move each vertex along a straight line between its starting and ending positions. However, this can lead to very poor animations which confuse rather than reveal the structure of the changes, particularly in cases where part or all of the drawing has been rotated or flipped.

With this problem in mind, Friedrich and Eades [FE02] identify several properties of a good animation:

- Uniform motion—groups of vertices with similar relative positions at the beginning and the end should move together.
- Separation—vertices with different motion paths should not be too close together.
- Rigid motion—movements should be consistent with 2D projections of the motion of 3D rigid objects, to exploit human perceptual strengths.
- No misleading layouts—unfortunate overlaps, such as a vertex lying on an edge, can lead to incorrect conclusions about the graph's structure.
- Short motion paths—vertices should travel as short a route as possible, to make the motion easier to follow.

They give a four-step algorithm for animation designed to satisfy these properties:

- fade out vertices and edges not present in the end drawing,
- apply a rigid transformation (composed of translation, rotation, scaling, flipping, and/or shearing) to the entire graph to move the elements of the graph as close as possible to their positions in the end drawing,
- complete the movement of vertices to their final positions, and
- fade in vertices and edges not present in the beginning drawing.

The transformation is chosen to minimize the sum of the squared distances between the transformed nodes and their positions in the end drawing. To reduce the effect of outliers, a (weighted) centroid can be included in the node set, or several random subsets of nodes can be chosen and the best transformation used. Once the transformation has been found, smooth animation paths for the nodes can be computed by extracting the rotational part

of the transformation using polar matrix decomposition, then simultaneously interpolating the angle of rotation and the entries of the non-rotational part of the transformation matrix. Rotation around the center of the drawing can be achieved by incorporating a translation to the origin and back into the transformation matrix before the decomposition is done.

Simple linear interpolation can be used to move the transformed nodes to their final positions, but a more pleasing result can be obtained with a force-directed approach where nodes repel each other but are attracted to their final positions instead of their neighbors.

If the drawings contain subgraphs which move in different ways, the animation can be improved by applying the middle steps separately to each distinct subgraph. Friedrich and Houle [FH02] suggest two strategies for clustering nodes into groups with common transformations—k-means and eliminating edges in the Delaunay triangulation to merge triangles with sufficiently similar transformations—and note that both strategies can produce good results though they also have limitations.

References

[Bae81] R. M. Baecker. Sorting out sorting, 1981. 30 minute color sound film, Dynamic Graphics Project, University of Toronto, excerpted and reprinted in SIGGRAPH Video Review 7, 1983.

[BBG+99] Ryan S. Baker, Michael Boilen, Michael T. Goodrich, Roberto Tamassia, and B. Aaron Stibel. Testers and visualizers for teaching data structures. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '99, pages 261–265, New York, NY, USA, 1999. ACM.

[BCE+] A. Bilgin, D. Caldwell, J. Ellson, E. Gansner, Y. Hu, and S. North. GraphViz. http://www.graphviz.org/.

[BDG+00] J. Berry, N. Dean, M. K. Goldberg, G. E. Shannon, and S. Skiena. LINK: a system for graph computation. *Software: Practice and Experience*, 30(11):1285–1302, 2000.

[Ber] J. Berry. LINK online manual. http://dimacs.rutgers.edu/ berryj/manual/.

[BGKT00] S. Bridgeman, M. T. Goodrich, S. G. Kobourov, and R. Tamassia. PILOT: an interactive tool for learning and grading. In *SIGCSE 2000*, pages 139–143, March 2000.

[BGT99] Stina Bridgeman, Ashim Garg, and Roberto Tamassia. A graph drawing and translation service on the World Wide Web. *Internat. J. Comput. Geom. Appl.*, 9(4–5):419–446, 1999.

[BN90] K. Bohringer and F. Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proc. ACM Conf. on Human Factors in Computing Systems*, pages 43–51, 1990.

[Bra01] Jürgen Branke. Dynamic graph drawing. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs*, volume 2025 of *Lecture Notes in Computer Science*, pages 228–246. Springer Berlin / Heidelberg, 2001.

[BS84] Marc H. Brown and Robert Sedgewick. A system for algorithm animation. *SIGGRAPH Comput. Graph.*, 18:177–186, January 1984.

[BS85] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Softw.*, 2(1):28–39, January 1985.

[BT98] Stina Bridgeman and Roberto Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In *Journal of Graph Algorithms and Applications*, pages 57–71. Springer-Verlag, 1998.

[BW97] Ulrik Brandes and Dorothea Wagner. A bayesian paradigm for dynamic graph layout. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 236–247. Springer-Verlag, 1997.

[BW98] Ulrik Brandes and Dorothea Wagner. Dynamic grid embedding with few bends and changes. In *Proceedings of the 9th International Symposium on Algorithms and Computation*, ISAAC '98, pages 89–98. Springer-Verlag, 1998.

[CIHJB07] James H. Cross II, T. Dean Hendrix, Jhilmil Jain, and Larry A. Barowski. Dynamic object viewers for data structures. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 4–8, New York, NY, USA, 2007. ACM.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[DG02] Stephan Diehl and Carsten Görg. Graphs, they are changing. In *Revised Papers from the 10th International Symposium on Graph Drawing*, GD '02, pages 23–30. Springer-Verlag, 2002.

[DGK01] Stephan Diehl, Carsten Görg, and Andreas Kerren. Preserving the mental map using foresighted layout. In *In Proceedings of Joint Eurographics IEEE TCVG Symposium on Visualization VisSym'01*, pages 175–184. Springer Verlag, 2001.

[EHK+04] Cesim Erten, Philip Harding, Stephen Kobourov, Kevin Wampler, and Gary Yee. GraphAEL: Graph animations with evolving layouts. In Giuseppe Liotta, editor, *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, pages 98–110. Springer Berlin / Heidelberg, 2004.

[ELMS91] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.

[EPD92] Marc Eisenstadt, Blaine A. Price, and John Domingue. Software visualization as a pedagogical tool. *Instructional Science*, 21:335–364, 1992.

[FE02] Carsten Friedrich and Peter Eades. Graph drawing in motion. *Journal of Graph Algorithms and Applications*, 6:2002, 2002.

[FH02] Carsten Friedrich and Michael Houle. Graph drawing in motion II. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 122–125. Springer Berlin / Heidelberg, 2002.

[Fin01] I. Finocchi. Layered drawings of graphs with crossing constraints. In *COCOON '01*, pages 357–367, 2001.

[For04] M. Forster. A fast and simple heuristic for constrained two-level crossing reduction. In *GD '04*, pages 206–216, 2004.

[GGJ06] Ashim Garg, Paul V. Gestwicki, and Bharat Jayaraman. Interactive program visualization and graph drawing. In M. Sethumadhavan, editor, *Discrete Mathematics and Its Applications*, pages 36–52. Narosa Publishing House Pvt. Ltd., 2006.

[GJ05] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of JIVE. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 95–104, New York, NY, USA, 2005. ACM.

[GJG04] P. V. Gestwicki, B. Jayaraman, and A. Garg. From class diagrams to object diagrams: A systematic approach. Technical Report 2004-21, University at Buffalo, State University of New York, December 2004.

[GJK+03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 179–188, New York, NY, USA, 2003. ACM.

[GKNV93] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19:214–230, 1993.

[Ham04] J. Hamer. Visualising Java data structures as graphs. In *CRPIT '30: Proceedings of the 6th Conference on Australian Computing Education*, pages 125–129. Australian Computer Society, Inc., 2004.

[HCIB04] T. Dean Hendrix, James H. Cross II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight IDE. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 387–391, New York, NY, USA, 2004. ACM.

[HWF90] Robert R. Henry, Kenneth M. Whaley, and Bruce Forstall. The University of Washington illustrating compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–233, New York, NY, USA, 1990. ACM.

[Kam89] T. Kamada. *Visualizing Abstract Objects and Relations.* World Scientific Series in Computer Science, 1989.

[KK89] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.*, 31:7–15, 1989.

[KKMS04a] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke. MatrixPro—a tool for demonstrating data structures and algorithms ex tempore. In *Proc. IEEE Int. Conf. on Advanced Learning Technologies*, pages 892–893, 2004.

[KKMS04b] Ville Karavirta, Ari Korhonen, Lauri Malmi, and Kimmo Stlnacke. MatrixPro – a tool for on-the-fly demonstration of data structures and algorithms. In *Proceedings of the Third Program Visualization Workshop*, pages 26–33. Department of Computer Science, University of Warwick, UK, July 2004.

[KM02] Ari Korhonen and Lauri Malmi. Matrix: concept animation and algorithm simulation system. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI, pages 109–114, New York, NY, USA, 2002. ACM.

[KMBW02] E. Kruja, J. Marks, A. Blair, and R. C. Waters. A short note on the history of graph drawing. In *GD '01*, pages 272–286. Springer-Verlag, 2002.

[KOD+96] M. S. Krishnamoorthy, F. Oxaal, U. Dogrusoz, D. Pape, A. Robayo, R. Koyanagi, Y. Hsu, D. Hollinger, and A. Hashimi. GraphPack: Design and features. In P. Eades and K. Zhang, editors, *Software visualization*, pages 83–99. World Scientific, 1996.

[KQPR03] Michael Kolling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[LMR98] Kelly A. Lyons, Henk Meijer, and David Rappaport. Algorithms for cluster busting in anchored graph drawing. *J. Graph Algorithms Appl.*, 2(1):1–24, 1998.

[LNR03] J. Lucas, T. L. Naps, and G. Rößling. VisualGraph—a graph class designed for both undergraduate students and educators. In *SIGCSE 2003*, pages 167–171, February 2003.

[MB98] Carolyn McCreary and Larry Barowski. VGJ: Visualizing graphs through java. In Sue Whitesides, editor, *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 454–455. Springer, 1998.

[MCS98] Carolyn McCreary, Richard Chapman, and Fwu-Shan Shieh. Using graph parsing for automatic graph drawing. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, pages 545–561, 1998.

[MELS95] K. Misue, Peter Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *J. Visual Lang. Comput.*, 6(2):183–210, 1995.

[MHT93] K. Miriyala, S. W. Hornick, and R. Tamassia. An incremental approach to aesthetic graph layout. In *Proc. Internat. Workshop on Computer-Aided Software Engineering*, 1993.

[MKK+04] Lauri Malmi, Ville Karavirta, Ari Korhonen, Jussi Nikander, Otto Seppl, and Panu Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. In *Informatics in Education*, page 048, 2004.

[MMSBA04] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 373–376, New York, NY, USA, 2004. ACM.

[Moe90] S. Moen. Drawing dynamic trees. *IEEE Softw.*, 7:21–28, 1990.

[Mur84] J. E. Murdoch. *Album of Science: Antiquity and the Middle Ages*. Charles Scribner's Sons, New York, 1984.

[Nap98] Thomas L. Naps. A Java visualiser class: incorporating algorithm visualisations into students' programs. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education*, ITiCSE '98, pages 181–184, New York, NY, USA, 1998. ACM.

[NEN00] T. L. Naps, J. R. Eagan, and L. L. Norton. JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. In *SIGCSE '00: Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, page 109–113, Austin, Texas, 2000. ACM Press.

[Nor96] S. North. Incremental layout in DynaDAG. In F. J. Brandenburg, editor, *Graph Drawing (Proc. GD '95)*, volume 1027 of *Lecture Notes Comput. Sci.*, pages 409–418. Springer-Verlag, 1996.

[NW02] Stephen C. North and Gordon Woodhull. Online hierarchical graph drawing. In *Revised Papers from the 9th International Symposium on Graph Drawing*, GD '01, pages 232–246. Springer-Verlag, 2002.

[Owe86] G. S. Owen. Teaching of tree data structures using microcomputer graphics. In *SIGCSE '86: Proceedings of the 17th SIGCSE Technical Symposium on Computer Science Education*, pages 67–72. ACM Press, 1986.

[PAC01] Helen C. Purchase, Jo-Anne Allder, and David A. Carrington. User preference of graph layout aesthetics: A UML study. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 5–18. Springer-Verlag, 2001.

[PNK11] Sergey Pupyrev, Lev Nachmanson, and Michael Kaufmann. Improving layered graph layouts with edge bundling. In *Proceedings of the 18th International Conference on Graph Drawing*, GD'10, pages 329–340. Springer-Verlag, 2011.

[PR98] W. C. Pierson and S. H. Rodger. Web-based animation of data structures using JAWAA. In *SIGCSE '98*, pages 267–271, 1998.

[PT98] A. Papakostas and I. G. Tollis. Interactive orthogonal graph drawing. *IEEE Trans. Comput.*, C-47(11):1297–1309, 1998.

[RDM+87] L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Softw. – Pract. Exp.*, 17(1):61–76, 1987.

[RF01] G. Rößling and B. Freisleben. Program visualization using AnimalScript. In *Proceedings of the First Program Visualization Workshop, PVW'00*, page 41–52, Porvoo, Finland, 2001. University of Joensuu Press, University of Joensuu Press.

[RMS11] Guido Rößling, Mihail Mihaylov, and Jerome Saltmarsh. AnimalSense: combining automated exercise evaluations with algorithm animations. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ITiCSE '11, page 298–302, New York, NY, USA, 2011. ACM.

[See97] Jochen Seemann. Extending the sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. Di Battista, editor, *Graph Drawing (Proc. GD '97)*, volume 1353 of *Lecture Notes Comput. Sci.*, pages 415–424. Springer-Verlag, 1997.

[SHY96] C. A. Shaffer, L. S. Heath, and J. Yang. Using the Swan data structure visualization system for computer science eduction. In *SIGCSE '96*, pages 140–144, February 1996.

[SK93] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *J. Parallel Distrib. Comput.*, 18:258–264, June 1993.

[SS10] James T. Streib and Takako Soma. Using contour diagrams and JIVE to illustrate object-oriented semantics in the Java programming language. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 510–514, New York, NY, USA, 2010. ACM.

[Sta90a] J. T. Stasko. Simplifying algorithm animation with tango. In *Proc. IEEE Workshop on Visual Languages*, pages 1–6, 1990.

[Sta90b] J. T. Stasko. Tango: a framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[Sta97] John T. Stasko. Using student-built algorithm animations as learning aids. In *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '97, pages 25–29, New York, NY, USA, 1997. ACM.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[Tam98] R. Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):89–122, 1998.

[Tun94] D. Tunkelang. A practical approach to drawing undirected graphs. Technical Report CMU-CS-94-161, School Comput. Sci., Carnegie Mellon University, June 1994.

[Wad01] V. E. Waddle. Graph layout for displaying data structures. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, pages 241–252. Springer-Verlag, 2001.

[Wal90] J. Q. Walker II. A node-positioning algorithm for general trees. *Softw. – Pract. Exp.*, 20(7):685–705, 1990.

[WBP04] David Workman, Margaret Bernard, and Steven Pothoven. An incremental editor for dynamic hierarchical drawing of trees. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science—ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 986–995. Springer-Verlag, 2004.