Galant: A Graph Algorithm Animation Tool[*]

Matthias Stallmann[†] Jason Cockrell Tynan Devries Weijia Li
Alexander McCabe Yuang Ni Michael Owoc Kai Pressler-Marshall

January 6, 2017

Abstract

A host of algorithm animation programs have been developed over the years. Primarily these have been designed for classroom use and involve considerable overhead for the animator of the animations (an instructor or developer) — students are passive observers. We distinguish three primary roles: the *observer*, who simply watches an animation; the *explorer*, who is able to manipulate problem instances; and the *animator*, who designs an animation. A key feature of Galant[1] is that it simplifies the role of the animator so that students can create their own animations by adding a few visualization directives to pseudocode-like implementations of algorithms. The focus on graph algorithms has a key advantage: the objects manipulated in an animation all have the same type. The restriction to graphs need not be a limitation: graphs are ubiquitous and Galant therefore provides a framework for animations in domains beyond classic graph algorithms; examples include search trees, automata, and even sorting.

Galant is also distinguished in that it is a tool rather than a closed system. In other words, it is designed to interact easily with other software such as text editors, other graph editors, other algorithm animation tools, graph generators, Java API's, format translation filters and graph drawing programs. This interactivity significantly expands the range of Galant's applications, including, for example, as a research tool for exploring graph algorithms.

Details of this report are consistent with Version 6.0 of Galant. Source code is available at https://github.com/mfms-ncsu/galant

[*]This software was conceived by the first author and implemented in two stages. In the first (Spring 2013), authors Cockrell, Devries, McCabe and Owoc implemented a working version as a senior design project at North Carolina State University, Department of Computer Science. This version, after some enhancements by Stallmann, was used by students in a data structures and algorithms course to develop their own animations as class projects. In Summer 2015, authors Li, Ni and Pressler-Marshall added significant enhancements: better syntax and compilation, keyboard shortcuts to add/remove nodes and edges and a mechanism to run animations in real time. We thank Dr. Robert Fornaro and Margaret Heil, the instructors of the senior design course, for coordinating the project(s) and providing instruction in communication skills, respectively. Ignacio Dominguez provided valuable technical support.

[†]Matthias Stallmann, Dept. of Computer Science, NC State University, Raleigh NC, 29695-8206, mfms@ncsu.edu.

[1] Aside from being an acronym, Galant is a term for a musical style that featured a return to classical simplicity after the complexity of the late Baroque era. We hope to achieve the same in our approach to algorithm animation.

Contents

List of Figures

List of Tables

1 Background

Algorithm animation has a long history, dating back at least as far as the work of Brown and Sedgewick [10, 11] and that of Bentley and Kernighan [6] in the 1980's. The BALSA software, developed by Brown and Sedgewick, is a sophisticated system that provides several elaborate examples of animations, including various balanced search trees, Huffman trees, depth-first search, Dijkstra's algorithm and transitive closure. The Bentley-Kernighan approach is simpler: an implementation of an algorithm is annotated with output directives that trace its execution. These directives are later processed by an interpreter that converts each directive into a still picture (or modification of a previous picture). The pictures are then composed into a sequence that is navigated by the user.

In discussing algorithm animation software, we distinguish three primary roles: the *observer*, who simply watches an animation; the *explorer*, who interacts with an animation by, for example, changing the problem instance (graph); and the *animator*, who designs an animation. The latter may also be referred to as a *developer* if the process of creating animations is integrated with that of implementing the animation system.[2] An explorer needs to be an observer as well and an animator needs to be both of the others. Rössling and Freisleben [23] articulate a similar classification of roles.

We also define an algorithm animation *tool* as animation software specifically designed to interact easily with other programs such as text editors, other graph editors, other algorithm animation tools, graph generators, Java API's, format translation filters and graph drawing programs.

A hypothesis, at least partially validated (using student attitude surveys [32]) is

1. The value added — beyond lectures and textbook — for students watching an animation (observer role) is minimal.
2. If students are able to manipulate problem instances (explorer role) the gain is more significant.
3. Students who implement algorithms and design simple animations of them (animator role) are likely internalize the structure of the algorithm and come away with significant understanding of it.

We call animations that are designed for (1) above, i.e., observer focused, *demonstration* animations; those designed for (2), i.e., for both observer and explorer, are *interactive* animations; those designed for (3) are *creative* animations.

2 Related work

There is a large variety of algorithm animations available. Here we focus solely on those that offer significant mechanisms for creating new animations. These differ primarily in (i) the role of the user (primarily observer or explorer as well); and (ii) the challenges imposed on the animator. Almost all of these are systems rather than tools. For a comprehensive survey of graph algorithm animation and graph drawing as educational tools see Bridgeman [9].

Observer-oriented (passive/demonstration) systems. One general purpose animation program is ANIMAL [23, 24]; it provides the animator with a rich menu of elements common to many algorithms. Steps in the animation are linked to steps in the pseudocode. Though there are many options for creating interesting animations, it appears that these are passive.

Galles [14] is an animation tool with very sophisticated creation options. Primarily designed to be passive, it could conceivably, with a parser for a graph input format and a mechanism that allows the user to view and manipulate the input graph, be made interactive. Then it would also be a tool. It suffers, however, from the fact that the animator must navigate a complex Java-based interface.

[2] With Galant a *developer* is someone who participates in modifying the underlying Galant implementation, an activity completely independent of creating animations.

Although secondary to its main purpose as a library of data structures and algorithms, LEDA [22] offers a graph window facility that can be used to create animations of graph algorithms. The documentation gives several examples and illustrates the rich functionality of the drawing and visualization capability of graph windows. Since LEDA is a general purpose, C++-based, programming language for algorithms and data structures, it is easily augmented with extensions that are integrated seamlessly with the core API; in this case, graph windows work in concert with core graph functions and macros. Unfortunately LEDA is a commercial product with non-trivial licensing cost.

Explorer-oriented (interactive) systems. Several online applets feature graph algorithm animation. Of these, JAVENGA [33] stands out. It is highly interactive. The drawing and editing of graphs is simple and intuitive, and graphs can be viewed in all three major representations (drawing, adjacency matrix and adjacency list). The variety of graph algorithms available is impressive: breadth-first and depth-first search, topological sort, strongly connected components, four shortest path algorithms, two minimum spanning tree algorithms, and a network flow algorithm. Animations can be run one step at a time with the option of moving backwards or continuously with an adjustable number of milliseconds per step. Javenga's main drawback is that the explorer is unable to save graphs for future sessions.

The j-Alg [21] is an impressive animation system. It is highly interactive, has a relatively easy to use interface, and has sophisticated animations for a large variety of algorithms, including graph searching, Dijkstra's algorithm, algebraic path problems (generalizations of all-pairs shortest paths and transitive closure), AVL trees, Knuth-Morris-Pratt string searching and BNF syntax diagrams. Its only drawback is that there is no readily available mechanism for outsiders to create new animations. New animations are developer-created and released periodically.

Animator-oriented (creative) systems. Edgy [7] is an applet that combines exploration, simple programming techniques (using the Snap [27] language) and a simple graph API. Since it stores graphs in `dot` format – see [18] – and programs in `XML`, it could be expanded into a tool akin to Galant. However, the programming capabilities are primitive; it would be difficult, for example, to add sophisticated data structures such as priority queues or external Java API's. From the tutorial videos for Edgy it is clear that the software is intended primarily for users who are just learning to program.

Libraries. AlgoViz [1] is a large catalog of algorithm animations, continually updated by contributors who either submit new animations or comment on existing ones. Like any large repository with many contributors, AlgoViz is difficult to monitor and maintain. The OpenDSA project [13, 25, 26] aims to create a textbook compilation of a variety of visualizations, mostly designed for observers.

Early work. Earlier animation tools/systems include GDR [30], John Stasko's Tango [31], Xtango, and SAMBA,[3] and, of course, the work of Brown and Sedgewick [10, 11] and that of Bentley and Kernighan [6]. SAMBA, and to a lesser extent GDR, is especially notable for emphasis on simplifying the creation of animations so that students can easily accomplish them. Both are also tools by our definition. All of these suffer, however, from using old technology, and, except for GDR, they require off-line creation of problem instances and have no graph-algorithm specific implementations or graph creation interfaces to offer.

3 Galant features

Galant is based on GDR, which we discuss in Section 3.1. We then give an overview of Galant – Section 3.2. The most important aspect of Galant is the ability to create animations easily, as discussed in Section 3.3. Then we give a brief overview of the user interface – Section 3.4; a more detailed description is given in Appendix B.

[3] These and Stasko's other animation tools are posted at http://www.cc.gatech.edu/gvu/ii/softvis/

3.1 GDR: Galant's predecessor

Galant is a successor to GDR [29, 30] and has much of the same functionality. The design of GDR is illustrated in Fig. 1. To the left of the dotted line are the interactions with external entities, as supported by GDR. The GDR user, when running a specific animation created and compiled externally, acts as both the editor of problem instances and as initiator of an algorithm animation with which (s)he may then interact, i.e., plays the role of explorer and of observer. Input and output take the form of a simple text-based file format that can be manipulated outside of GDR via text filters, graph editors, graph drawing applications, etc. It is this external manipulation capability that makes GDR a tool rather than a closed system.

The animator writes a C program that interacts with a graph ADT whose functions access and/or modify both the internal representation of the graph and the user's view of it. The ADT functions can be classified into one of three categories depending on the graph attributes accessed: (i) *logical* attributes — labels (and identities) of nodes and labels and endpoints of edges; (ii) *geometric* attributes — the positions of nodes and labels and inflection points of edges; and (iii) *display* attributes — highlighting of nodes and edges, making labels visible/invisible, etc.

While GDR has much to recommend it when compared with other algorithm animation software, it suffers from some serious drawbacks:

- Each animation is a separate C program that interacts with an X11 window server. Therefore GDR is not portable.
- The user interface is crude. Aside from being black and white it has no file browser, no rubber-banding of moves, non-standard keyboard shortcuts and an unappealing look and feel.
- While the API supports access to the graph itself, there is no API support for data structures commonly used in graph algorithms (stacks, queues, priority queues).

3.2 Galant overview

Fig. 2 gives an overview of Galant functionality. A graphical user interface (GUI) allows the user to edit both graphs and algorithm animations, either loaded as already existing files or newly created. At any point, the user can apply a selected animation to a selected graph for viewing. The animation program steps forward until it displays the next animation event or, if a `beginStep()` call has marked the start of a sequence of events, until it reaches the next `endStep()` call. It then pauses execution and waits for the user to decide whether to step forward, step backward, or exit. A step forward resumes execution while a step backward returns the display to a previous state. The algorithm resumes execution only when the *display state* indicated by the user's sequence of forward and backward steps ($f - b$, where f is the number of forward and b the number of backward steps) exceeds the *algorithm state*, the number of animation steps the algorithm has executed.

When editing a graph the user can create/delete nodes and edges (when in the appropriate mode) by clicking and/or moving the mouse, and can move vertices by dragging the mouse. There is also an interface for specifying labels, weights and colors for both nodes and edges. Keyboard shortcuts are available for these operations.

A preferences panel allows the user to select font size for labels and a variety of other options. Any changes to a graph are also reflected in a text (GraphML) representation of the graph, which can also be edited directly. Naturally the GraphML representation can also be created or edited externally: by a random or structured graph generator, by translation from another format, by directly editing the GraphML or by invoking a separate graph editor. Galant has a built-in force-directed drawing program (see Hu [20]) to position nodes automatically if so desired. Automatic drawing is useful when the input GraphML file does not provide position information for the nodes (and their positions are selected randomly). Other drawing programs, such as those provided by GraphViz [18] and the huge body of research carried on by the graph drawing community [16], can be used externally as well. Graphs used in Galant animations can be analyzed externally using tools such as Gephi [15].

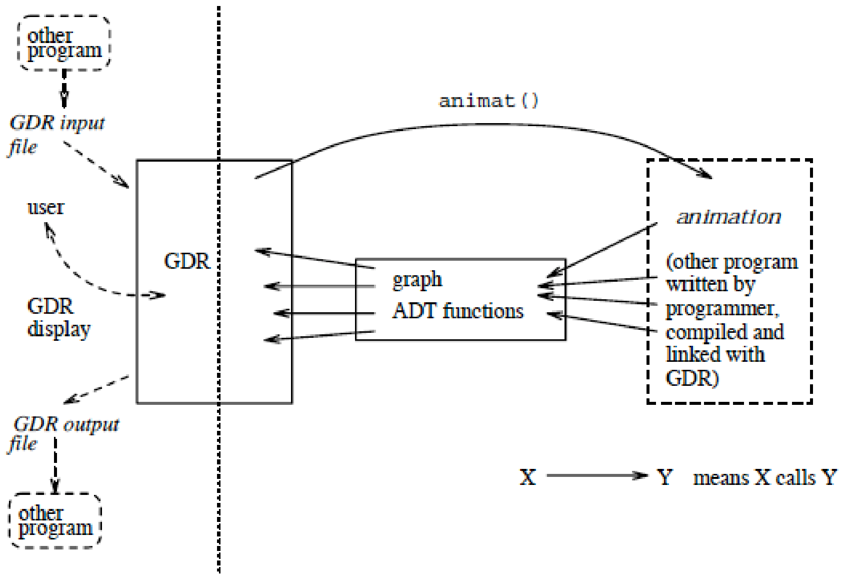


Figure 1: GDR design.

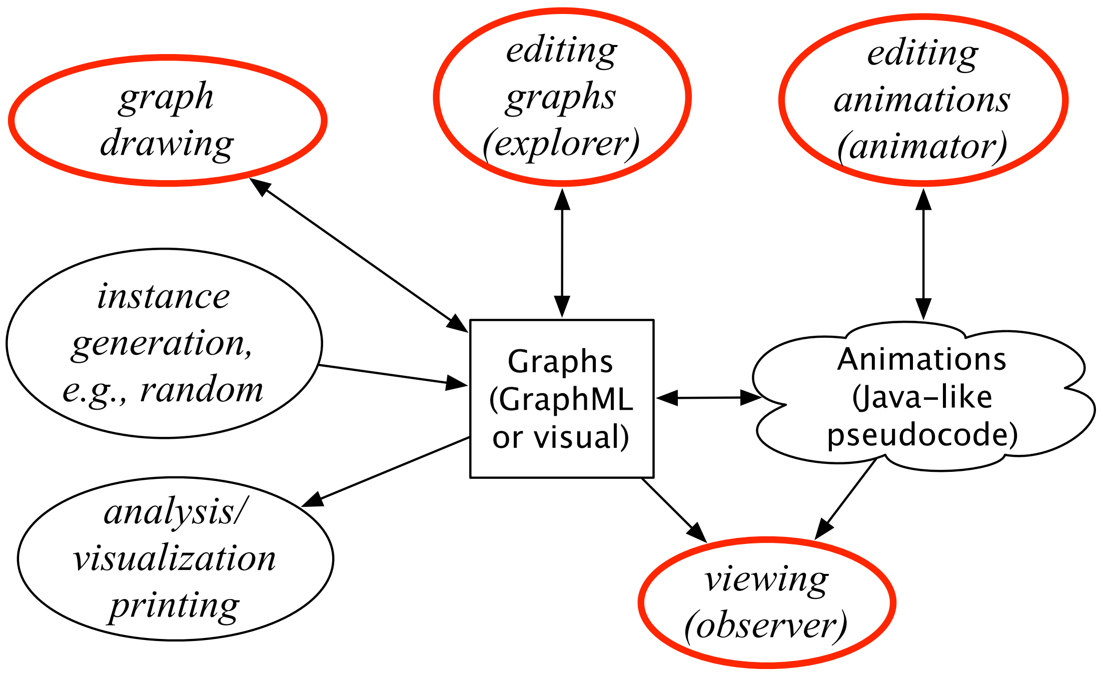

graph drawing
editing graphs (explorer)
editing animations (animator)
instance generation, e.g., random
Graphs (GraphML or visual)
Animations (Java-like pseudocode)
analysis/ visualization printing
viewing (observer)

Figure 2: Illustration of Galant design and functionality. Arrows indicate data flow. All functions, shown as ovals can be performed easily outside of Galant. The ones with thick red borders are part of current Galant functionality.

Editing/compiling an algorithm animation is just like performing the same operations on a Java program. The compiler is essentially a Java compiler that preprocesses the algorithm code and compiles it, importing the relevant modules. The preprocessor converts traversals of incidence lists and other lists of nodes or edges into the corresponding, more obscure, Java code. It also shields the Galant programmer from such syntactic circumlocutions as declaring `static` methods. Almost all Galant functions are available as both methods using object-oriented notation, e.g., `v.highlight()`, and as procedures, e.g., `highlight(v)`.

Because the functionality of the Galant editor is limited, it is usually more convenient to use an external program editor, reserving the Galant editor to make minor changes in response to compile-time or runtime errors.

We use GraphML [17] as our graph representation because it is flexible, it can easily be extended, and parsers, viewers and translation tools are becoming more common. Because GraphML is specialized XML, parsers for the latter, based on the Document Object Model (DOM) can be used. These are available for many programming languages. Translators to other formats are also available or can easily be constructed. For example, the GraphViz [18] download provides one; unfortunately it preserves only the connectivity information. However, there is straightforward mapping between the GraphML attributes we use (positions of nodes and colors, etc., of nodes and edges) and the corresponding ones in GraphViz format. We have written conversion scripts among the following formats: GraphML, gml [19], sgf (used for layered graphs in crossing minimization), and dot (GraphViz [18]).

3.3 For the animator

When it comes to creating animations, Galant offers the most significant advantages over GDR and, a fortiori, over other algorithm animation software. Among these are:

- The API interface is simpler, due, in part, to the fact that the underlying language is Java rather than C.
- Each node and edge has both a weight and a label. Conversion of a weight to a number is automatic while labels are kept as text. The programmer can choose the appropriate attribute, which makes the implementation more transparent and devoid of explicit conversions.
- Most data structures are built in: stacks, queues, lists and priority queues of both edges and vertices. Priority queues implicitly use the weight attribute of the node/edge in question. The weight attribute is also used for sorting.
- An algorithm initially designed for directed graphs can usually be applied to undirected graphs (and get the desired interpretation) with no change in the implementation. This is useful, for example, when implementing Dijkstra's algorithm.
- The interface that allows an explorer to edit graph instances can also be used to edit, compile, and run algorithm implementations. While initial creation and major edits are usually more convenient via a standard program editor offline, an algorithm window in Galant can be used to view the algorithm and make corrections in response to compile or runtime errors.

The philosophy behind the API design is that it should be usable by someone familiar with graph algorithms but only a rudimentary knowledge of Java (or any other programming language). The fact that Galant code resembles the pseudocode used in one of the most popular algorithm design and analysis texts, that of Cormen et al. [12], attests to the fact that we have succeeded.

A key advantage of the API design, not present in, for example, BALSA or Edgy, is that it sits directly on top of Java. This allows the animator to develop arbitrarily complex algorithms using other Java class API's and ones devised by the animator. More importantly, it allows Galant to offer significant new functionality provided by developers with only a modicum of Java training: *sets* of nodes and edges (in addition to the stacks and queues already built in)[4] or significant infrastructure for algorithms in a specific domain, as was done for crossing minimization in layered graphs.

[4] The most recent release has sets built-in – these were easily added by the developers.

3.4 The Galant user interface

Two windows appear when Galant is started: a *graph window* shows the current graph and a *text window* shows editable text. Depending on the currently selected tab in the text window, the text can be either a GraphML description of the current graph or the source code of an algorithm.

The user interface is designed for all three roles. The observer (or an instructor demonstrating an algorithm) does as follows: (a) loads a graph using the file browser; (b) loads an algorithm; (c) pushes the "Compile and Run" button; and (d) uses the controls underneath the graph window or arrow keys to step through the algorithm forward or backward as desired.

A typical explorer might edit or create a graph using the graph window and then follow steps (b)–(d), repeating steps (a) and (d) to try out different graphs. Saving graphs for later use is also an option. In addition, the explorer can use the graph's tab in the text window to fine tune the placement of nodes or apply the provided graph drawing method – see Hu [20] – to adjust node placement.

An animator can load and edit an existing algorithm or create one from scratch using an appropriate tab in the text window. Compilation and execution is accomplished via the buttons at the bottom. In fact, the code of an animation is essentially a Java program with (a) predefined types for nodes and edges; (b) an API that interacts with the graph and with intended animation effects; (c) a set of built-in data structures for convenience; and (d) a set of macros that allows the program to traverse, for example, all incident edges of a node without invoking templated Java constructs. Line numbers of errors reported by the compiler are those of the code displayed in the text window. Runtime errors are reported in the same way. In both cases, due to the imports and macro translations, the error messages may not be immediately intelligible, [5] but the line numbers *are* correctly identified.

4 Future work

Perhaps the most promising direction that our work on Galant can take is that of developing research applications. The benefits are twofold. First, Galant has the potential for providing a rich environment for the exploration of new graph algorithms or developing hypotheses about the behavior of existing ones. Second, as Galant is augmented with the infrastructure for specific research domains (in the form of additional Java classes), some of the resulting functionality can be migrated into its core. Or the core will be enhanced to accommodate new capabilities. The former has happened with research on crossing minimization heuristics (e.g., [28]). The latter continues to happen – new functionality is added with each research project.

Because the node and edge states both guide the logic of the algorithm and how the nodes/edges are displayed, the nomenclature has become awkward. A better way to handle the situation is to add initial declarations that specify color, thickness, and, in case of nodes, fill color, for each logical state. A logical state would have a name, e.g., inTree, and be automatically provided with a setter (Boolean argument), e.g., setInTree, and a logical test, e.g., isInTree.

A key challenge confronting any developer of algorithm animation software is that of accessibility to blind users. Previous work addressed this via a combination *earcons*[6], spoken navigation and haptic interfaces (see [3, 4, 5]). The resulting algorithm animations were developed for demonstration and exploration rather than simplified creation of animations. In theory any graph navigation tool can be extended, with appropriate auditory signals for steps in an animation, to an algorithm animation tool. The most promising recent example, due to its simplicity, is GSK [2]. Earcons can be added to substitute for state changes of nodes or edges.

A user study testing the hypothesis that student creation of animations promotes enhanced learning raises several nontrivial questions. Are we going to measure ability to navigate specific algorithms? Or a broader understanding of graphs and graph algorithms? Can we make a fair comparison that

[5] Exception handling is continuously improving, however.

[6] *Earcons* are sounds that signal specific events, such as the arrival of email. The term was coined by Blattner et al. [8].

takes into account the extra effort expended by students to create and debug animations? Why incur the overhead of designing an experiment that is very likely to validate the obvious? Namely: in order to create a compelling animation, an animator must first decide what aspects of a graph are important at each step of an algorithm and then how best to highlight these. This two-stage process requires a longer and more intense involvement with an algorithm than mere exploration of an existing animation.

There are various implementation issues with and useful enhancements to the current version of Galant. These will be addressed in future releases. As new animations for teaching and research are created, other issues and desired enhancements will undoubtedly arise. The current implementation should be transparent and flexible enough to effect the necessary modifications — the most challenging aspect of creating enhancements has been and continues to be the design decisions involved.

References

[1] AlgoViz, *The Algorithm Visualization Portal.* algoviz.org. 5

[2] S. Balik, S. Mealin, M. Stallmann, and R. Rodman, *GSK: Universally Accessible Graph SKetching*, in SIGCSE, 2013. 9

[3] N. Baloian and W. Luther, *Visualization for the minds eye*, in Software Visualization, no. 2269 in LNCS, 2002, pp. 354–367. 9

[4] N. Baloian, W. Luther, and T. Putzer, *Algorithm explanation using multimodal interfaces*, in Proc. XXV Int. Conf. of the Chilean Comp. Soc., 2005. 9

[5] D. Bennett, *Effects of navigation and position on task when presenting diagrams to blind people using sound*, in Diagrams 2002, vol. 2317 of LNAI, 2002, pp. 161–175. 9

[6] J. L. Bentley and B. W. Kernighan, *A system for algorithm animation: Tutorial and user manual*, Computing Science Technical Report 132, AT&T Bell Laboratories, Jan. 1987. 4, 5

[7] S. Bird, M. Nazečić-Andrlon, and J. Gallina, *Edgy.* http://snapapps.github.io/. 5

[8] M. Blattner, D. Sumikawa, and R. Greenberg, *Earcons and icons: Their structure and common design principles*, Human Computer Interaction, 4 (1989), pp. 11–44. 9

[9] S. Bridgeman, *Graph drawing in education*, in Handbook of Graph Drawing and Visualization, R. Tamassia, ed., CCRC Press, 2013, ch. 24. 4

[10] M. H. Brown, *Exploring algorithms with Balsa II*, Computer, 21 (1988). 4, 5

[11] M. H. Brown and R. Sedgewick, *Techniques for algorithm animation*, IEEE Software, 2 (1985), pp. 28–39. 4, 5

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, third ed., 2009. 8, 13

[13] E. Fouh, M. Sun, and C. Shaffer, *OpenDSA: A creative commons active-eBook.* Poster presentation., SIGCSE, 2012. 5

[14] D. Galles, *Data structure visualizations.* cs.usfca.edu/∼galles/visualization/Algorithms.html. 4

[15] Gephi, *The open GraphViz platform.* gephi.org. 6

[16] *Internalional symposium on graph drawing.* www.graphdrawing.org, 1992–2013. 6

[17] *The GraphML file format.* graphml.graphdrawing.org. 8

[18] GraphViz, *Graph Visualization Software*. www.graphviz.org. 5, 6, 8

[19] M. Himsolt, *Gml: A portable graph file format*, tech. rep., Universität Passau, 1999. 8

[20] Y. Hu, *Efficient, high-quality force-directed graph drawing*, The Mathematica Journal, 10 (2006). 6, 9, 22

[21] *j-Algo: The algorithm visualization tool*. j-algo.binaervarianz.de. 5

[22] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999. 5

[23] G. Rössling and B. Freisleben, *ANIMAL: a system for supporting multiple roles in algorithm animation*, J. Visual Languages and Computing, 13 (2002), pp. 341–354. 4

[24] G. Rössling, M. Schüler, and B. Freisleben, *The ANIMAL algorithm animation tool*, in ITCSE 2000, 2002. www.algoanim.info/AnimalAV/. 4

[25] C. Shaffer, V. Karavirta, A. Korhonen, and T. Naps, *OpenDSA: Beginning a community hypertextbook project*, in Proceedings of 11th Koli Calling International Conference on Computing Education Research, 2011, pp. 112–117. 5

[26] C. Shaffer, T. Naps, and E. Fouh, *Interactive textbooks for computer science education*, in Proceedings of the Sixth Program Visualization Workshop, 2011, pp. 97–103. 5

[27] *Snap*. http://snap.berkeley.edu/. 5

[28] M. Stallmann, *A heuristic for bottleneck crossing minimization and its performance on general crossing minimization: Hypothesis and experimental study*, Journal on Experimental Algorithmics, 17 (2012). 9

[29] M. Stallmann, R. Cleaveland, and P. Hebbar, *GDR: A visualization tool for graph algorithms*, Tech. Rep. 91-27, Department of Computer Science, North Carolina State University, Raleigh NC 27695-8206, Oct. 1991. 6

[30] ——, *GDR: A visualization tool for graph algorithms*, in Computational Support for Discrete Mathematics, N. Dean and G. Shannon, eds., no. 15 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, March 1992, pp. 17–28. 5, 6

[31] J. T. Stasko, *Tango: A framework and system for algorithm animation*, Computer, 23 (1990), pp. 27–39. 5

[32] ——, *Using student-built algorithm animations as learning aids*, in SIGCSE, 1997, pp. 25–29. 4

[33] B. Thanasis, *Javenga*. users.uom.gr/∼thanasis/JAVENGA.html. 5

A Animation examples

We now illustrate the capabilities of Galant by showing several examples.

A.1 Dijkstra's algorithm

One of the simplest algorithms we have implemented is Dijkstra's algorithm for the single source shortest paths problem. Fig. 3 the implementation of the animation of Dijkstra's algorithm. At every step the nodes already in the shortest paths tree are *marked* (gray shading) and the nodes that have been encountered (but are not in the tree) are *highlighted* (thick red boundary); these are often referred to as *frontier* nodes. Selected *edges* (thick red) represent the current shortest paths to all encountered nodes; they are the edges of a shortest paths tree when the algorithm is done. The same algorithm animation works for both directed and undirected graphs, as illustrated in Figs. 4 and 5. The user can toggle between the directed and undirected versions of a graph via push of the appropriate button. The functions *beginStep* and *endStep* define the points at which the exploration of the algorithm stops its forward or backward motion. In their absence, any state change (mark, highlight, change in weight, etc.) constitutes a step, which, in some cases, can force the user to do excessive stepping to move past uninteresting state changes.

The macro *for_outgoing(*v, e, w*)* creates a loop whose body is executed once for each edge leading out of v; in the body, e refers to the current edge and w to the other endpoint (any other variable names can be used). In an undirected graph the term *outgoing* applies to all incident edges.[7]

The difference between what the algorithm does on a directed versus an undirected graph is evident in the figures. The edge *from* node 3 to node 2 in the directed graph becomes an edge *between* the two nodes in the undirected form of the same graph. Thus, in the undirected version, when node 2 is added to the tree it also causes the distance from the source, node 0, to node 3 to be updated, via the path through node 2. These snapshots come from the executions of the *same algorithm* on the *same graph*. The only difference is that the explorer toggled from the directed to the undirected interpretation of the graph.

The array `chosenEdge` is required in order to control the highlighting. Galant provides for seamless indexing of arrays with node id's: the function `nodeIds` simply returns the largest node id plus one (so that an array can be allocated correctly) and `id(v)` returns the id of v, to be used as an index. Node id's, therefore, need not be contiguous starting at 0; in general, they might not be because of deletions or when graphs come from external sources.

A.2 Kruskal's algorithm

Another simple algorithm implementation is that of Kruskal's algorithm for finding a minimum spanning tree (or forest) in a graph. Fig. 6 shows the implemented animation of Kruskal's algorithm. Additional Galant features illustrated here are:

- Use of the keyword `function` to declare a function: this avoids the syntactic complications of Java method declarations. In the case of `FIND_SET`, for example, you would normally have to say
  `static Node FIND_SET( Node x )`
  and would get error messages about non-static methods in a static context if you omitted the keyword `static`.
- No need to use the Java keyword `void` to designate a function with no return type. This is implicit if the return type is omitted as with `INIT_SET`, `LINK` and `UNION`.
- Implicit use of weights to sort the edges: weights are created by the explorer when editing a graph. The `sort` functions translates automatically into the relevant Java incantation.
- The ability to write messages during execution, as accomplished by the `display` calls.

[7] Also provided are *for_incoming* and *for_adjacent*; the latter applies to all incident edges, even for directed graphs.

```
algorithm {
    NodePriorityQueue pq = new NodePriorityQueue();
    Edge [] chosenEdge = new Edge[nodeIds()];
    beginStep();
    for_nodes(node) {
        setWeight(node, INFINITY);
        pq.add(node);
    }
    Node v = getStartNode();
    setWeight(v, 0);
    endStep();

    while ( ! pq.isEmpty() ) {
        v = pq.removeMin();
        mark(v);        // nodes are marked when visited
        unhighlight(v); // and highlighted when on the frontier
        for_outgoing ( v, e, w ) {
            if ( ! marked(w) )  {
                if ( ! highlighted(w) ) highlight(w);
                double distance = weight(v) + weight(e);
                if ( distance < weight(w) ) {
                    beginStep();
                    highlight(e);
                    Edge previous_chosen = chosenEdge[id(w)];
                    if (previous_chosen != null )
                        unhighlight(previous_chosen);
                    pq.decreaseKey(w, distance);
                    chosenEdge[id(w)] = e;
                    endStep();
                }
            } // end, neighbor not visited (not in tree); do nothing if node
              // is already in tree
        } // end, adjacency list traversal
    } // stop when priority queue is empty
} // end, algorithm
```

Figure 3: The implementation of the Dijkstra' algorithm animation.

Two steps in the execution of the animation are shown in Fig. 7. The two endpoints of an edge are marked when that edge is considered for inclusion in the spanning tree. If the endpoints are not in the same tree, the edge is added and highlighted and the cost of the current tree is displayed in the message. Otherwise the message reports that the endpoints are already in the same tree.

A.3 Depth-first search (directed graphs)

Fig. 8 illustrates an animation of depth-first search. The code and definitions follow those of Cormen et al. [12]. Tree edges are highlighted (selected) and non-tree edges are labeled as **B**ack edges, **F**orward edges or **C**ross edges. White nodes, not yet visited, are neither highlighted (selected) nor marked; Gray nodes, visited but the visit is not finished, are highlighted only; and black nodes, visit is completed, are both highlighted and marked. Labels on nodes indicate the discovery and finish times, separated by a slash.

This particular algorithm, unlike our implementation of Dijkstra's, is intended for a directed graph. We would need to write a different algorithm/animation for undirected graphs. When a graph is undirected an edge is effectively directed both ways; hence tree edges would get relabeled as back edges when they are encountered the second time. The usual trick is to either mark edges if they

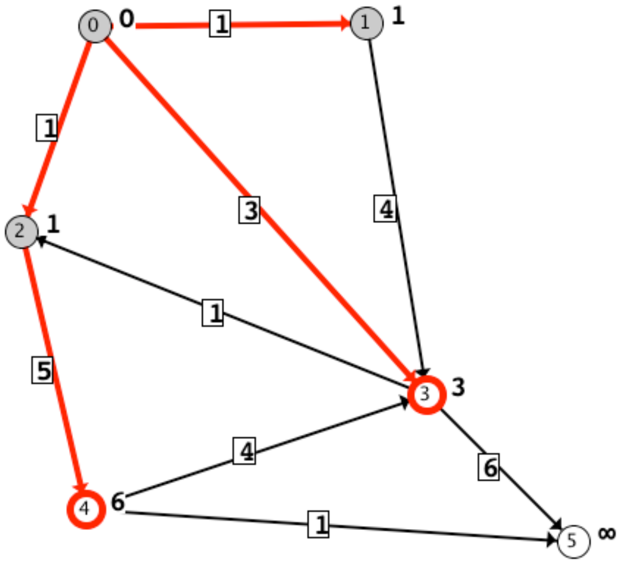


Figure 4: Dijkstra's algorithm on a directed graph.

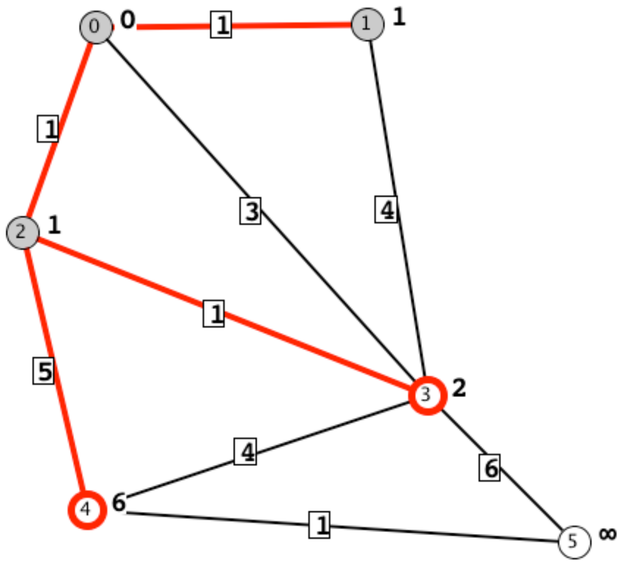
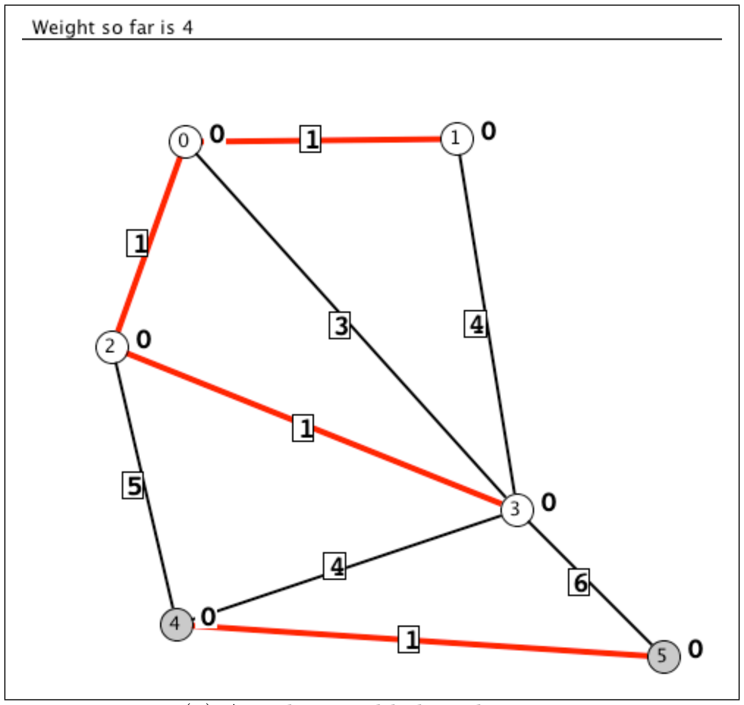


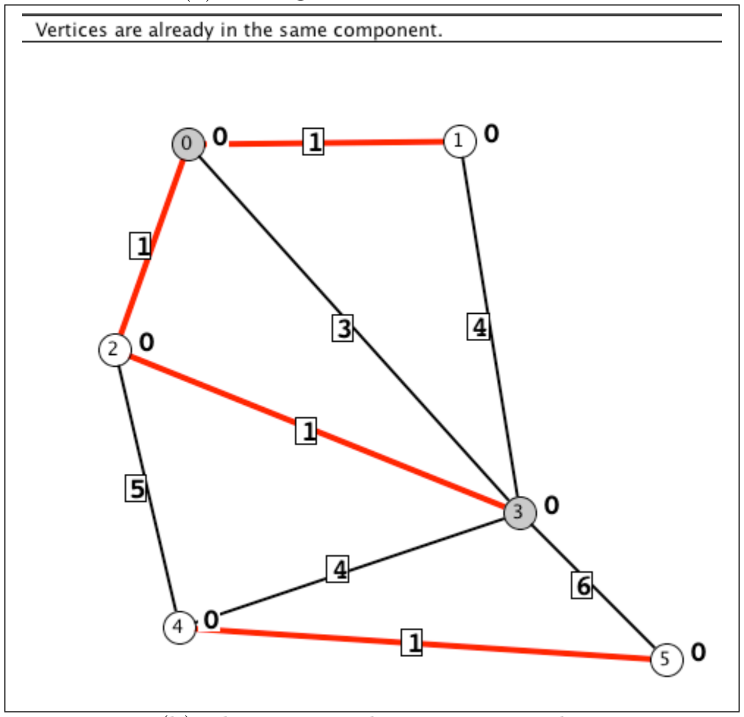
Figure 5: Dijkstra's algorithm on the same graph, undirected.

```
// standard disjoint set untilities; not doing union by rank or path
// compression; efficiency is not an issue

Node [] parent;

function INIT_SET(Node x) { parent[id(x)] = x; }

function LINK(Node x, Node y) { parent[id(x)] = y; }

function Node FIND_SET(Node x) {
    if (x != parent[id(x)])
        parent[id(x)] = FIND_SET(parent[id(x)]);
    return parent[id(x)];
}

function UNION(Node x, Node y) { LINK(FIND_SET(x), FIND_SET(y)); }

algorithm {
    parent= new Node[nodeIds()];
    for_nodes(u) {
        INIT_SET(u);
    }

    EdgeList edgeList = getEdges();
    sort(edgeList);

    // MST is only relevant for undirected graphs
    setDirected(false);

    int totalWeight = 0;
    for ( Edge e: edgeList ) {
        beginStep();
        Node h = source(e);
        Node t = target(e);
        // show e's endpoints as it's being considered
        // marking is used for display purposes only
        mark(h); mark(t);
        endStep();

        beginStep();
        // if the vertices aren't part of the same set
        if ( FIND_SET(h) != FIND_SET(t) ) {
            // add the edge to the MST and highlight it
            highlight(e);
            UNION(h, t);
            totalWeight += e.getWeight();
            display( "Weight so far is " + totalWeight );
        }
        else {
            display( "Vertices are already in the same component." );
        }
        endStep();

        beginStep(); unMark(h); unMark(t); endStep();
    }
    display( "MST has total weight " + totalWeight );
}
```

Figure 6: The implementation of Kruskal's algorithm animation.


Weight so far is 4

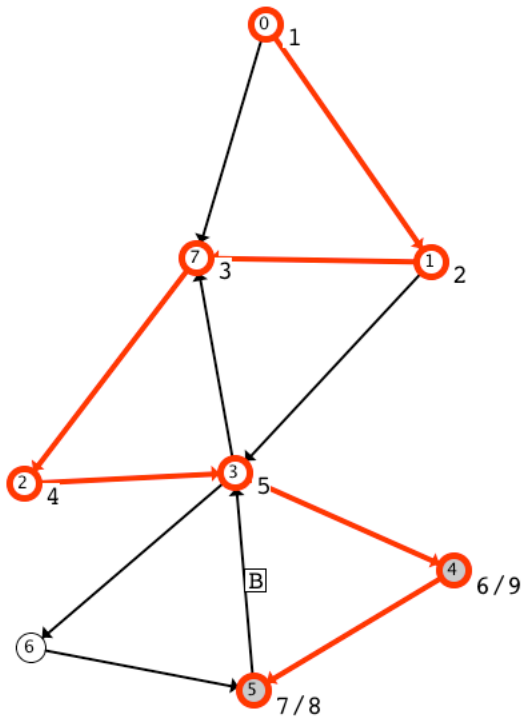
(a) An edge is added to the tree.


Vertices are already in the same component.

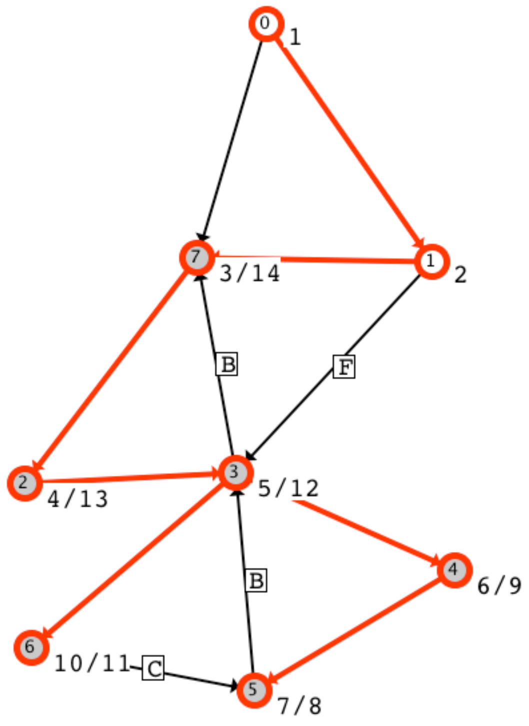
(b) The current edge creates a cycle.

Figure 7: Two steps in Kruskal's algorithm. A message at the top left of the window describes the state of the algorithm.

```
int time;

int [] discovery;
int [] finish;

function visit( Node v ) {
    time = time + 1;
    discovery[id(v)] = time;
    beginStep();
    label(v, discovery[id(v)] );
    select(v);
    endStep();
    for_outgoing( v, e, w ) {
        beginStep();
        if ( ! selected(w) ) {
            select(e);
            visit(w);
        }
        else if ( finish[id(w)] == 0 ) {
            label(e, "B");
        }
        else if ( finish[id(w)]
                  > discovery[id(v)] ) {
            label(e, "F");
        }
        else {
            label(e, "C");
        }
        endStep();
    }
    time = time + 1;
    finish[id(v)] = time;
    beginStep();
    mark(v);
    label(v, discovery[id(v)]
             + "/" + finish[id(v)]);
    endStep();
}

algorithm {
    time = 0;
    discovery = new int[nodeIds()];
    finish = new int[nodeIds()];
    for_nodes( u ) {
        if ( ! selected(u) ) {
            visit( u );
        }
    }
}
```



After first non-tree edge is labeled.



After all but one non-tree edges have been labeled.

Figure 8: Implementation of a depth-first search animation with an illustration of the graph panel during execution.

have already been visited or to keep track of the parent of a node when it is visited.

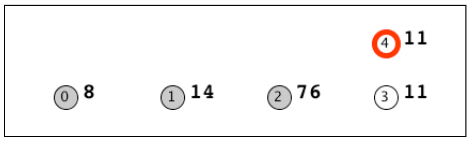
A.4 Insertion sort

Fig. 9 illustrates an animation of insertion sort that uses node movement and suggests that, with some creativity, other sorting algorithms might be animated as well – this has already been done for bubble sort and merge sort. At the beginning (not shown) the code: (i) puts nodes, evenly spaced, on a horizontal line; (ii) creates arrays `nodes`, `xCoord` and `A`, holding the nodes, their horizontal positions and their weights (i.e., the array to be sorted), respectively; and (iii) adds a new node `toInsert` for the element to be inserted at each outer-loop iteration.

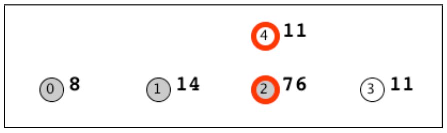
The rest of the code is a classic insertion sort implementation: nodes behave as if they were array positions; those in the already sorted part of the array are marked; a node that is being compared with `x` is highlighted. To ensure this one-to-one correspondence between comparisons and highlighting, there is a break in the middle of the main loop. The first animation step highlights the two nodes being compared. If the comparison signals that the loop should not be continued a `break` statement causes an exit. The second animation step then (i) copies the weight of `nodes[j]` (a.k.a. `A[j]`) to `nodes[j+1]`; (ii) unhighlights `nodes[j]` to indicate that `A[j]` is no longer involved in a comparison; (iii) unmarks `nodes[j]` to signal that `A[j]` is in transition; and (iv) marks `nodes[j+1]` to indicate that `A[j+1]` has its final value for this iteration of the outer loop.

The insertion sort animation begins with the declaration `movesNodes()`. This prevents the user from moving nodes during algorithm execution. Ordinarily, users *are* allowed to change node positions during execution to, for example, make labels and weights more visible. The new positions persist after algorithm execution. However, when an algorithm moves nodes, the nodes will revert to their original positions when the algorithm is done. Snapshots of positions (and other aspects of graph state) during execution can be created using the Export option on the file menu of the graph window.

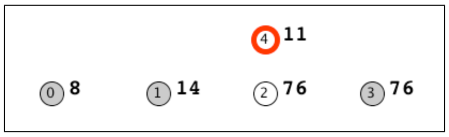
```
movesNodes();
for ( int i = 1; i < numNodes; i++ ) {
    beginStep();
    Double x = A[i]; setWeight(toInsert, A[i]);
    setX(toInsert, xCoord[i]);
    endStep();
    Integer j = i - 1;
    while ( j >= 0 ) {
        beginStep();
        setX(toInsert, xCoord[j]);
        highlight(nodes[j]);
        endStep();
        if ( A[j] <= x ) break;
        beginStep();
        A[j+1] = A[j]; setWeight(nodes[j+1], A[j]);
        unhighlight(nodes[j]);
        unmark(nodes[j]);
        mark(nodes[j+1]);
        endStep();
        j = j - 1;
    }
    beginStep();
    A[j+1] = x; setWeight(nodes[j+1], x);
    mark(nodes[j+1]);
    endStep();
}
```



(a) Starting to insert `x = A[3]`.



(b) Comparing `x` with `A[2]`.



(c) `A[2] > x` so `A[3] = A[2]`.

Figure 9: The insertion sort algorithm and three steps in the animation.

B User documentation

What follows are instructions for interacting with the Galant GUI interface.

B.1 Overview

Galant provides three major components across two windows:

1. a text window that can serve two distinct purposes –
   (a) as an editor of algorithms
   (b) as an editor of GraphML representations of graphs
2. a graph window that displays the current graph (independent of whether the text window shows an algorithm or the GraphML representation of the graph)

It is usually more convenient to edit algorithms offline using a program editor. The primary use of the text editor is to correct minor errors and to see the syntax highlighting related to macros and Galant API functions. The graph window is the primary mechanism for editing graphs. One exception is when precise adjustments node positions are desired. Weights and labels are sometimes also easier to edit in the text window.

These components operate in two modes: edit mode and animation mode. Edit mode allows the user to modify graphs – see Sec. B.3, or algorithms – see Sec. B.4. Animation mode disables all forms of modification, allowing the user to progress through an animation by stepping forward or backward, as described in Sec. B.5.

B.2 Workspace

Opened graph and algorithm files are displayed in the text window, which has tabs that allow the user to switch among different algorithms/graphs. New algorithms are created using the icon that looks like a page of text at the top left of the window; new graphs are created using the graph/tree icon to the left of that. More commonly, algorithm and graph files are loaded via the File-¿Open browser dialog. The File drop-down menu also allows saving of files and editing of preferences. Algorithm files have the extension .alg and graph files the extension .graphml.

Fig. 10 shows both the graph window (top) and the text window (bottom). Annotations on the graph window describe the components of the window that can be used to edit a graph visually.

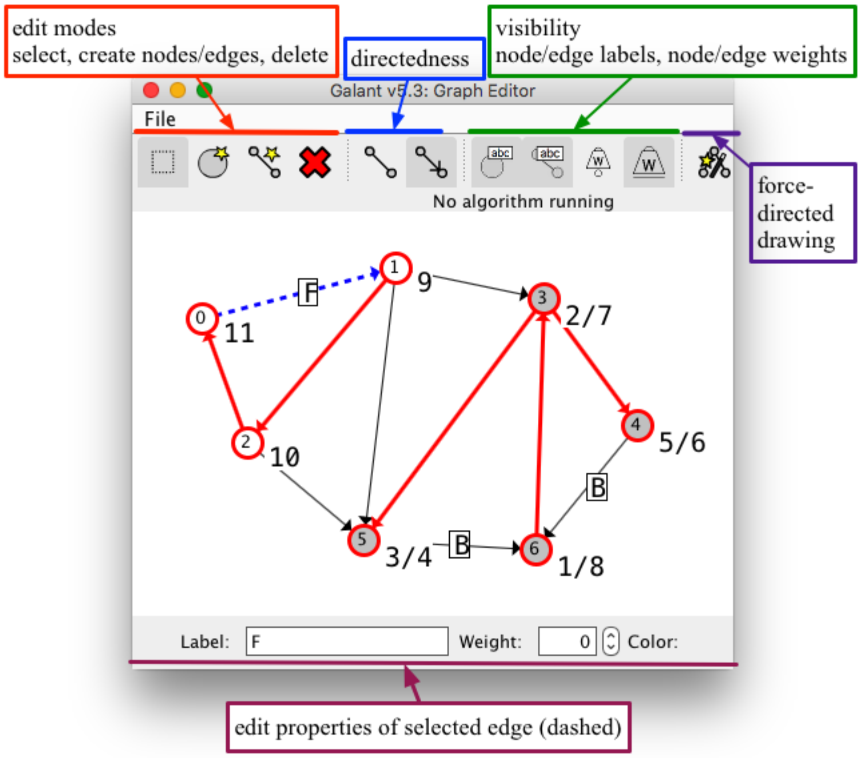
B.3 Graph editing

Graphs can be edited in their GraphML representation using the text window or visually using the graph window. These editors are linked: any change in the visual representation is immediately reflected in the text representation (and will overwrite what was originally there); a change in the GraphML representation will take effect in the visual representation when the file is saved.

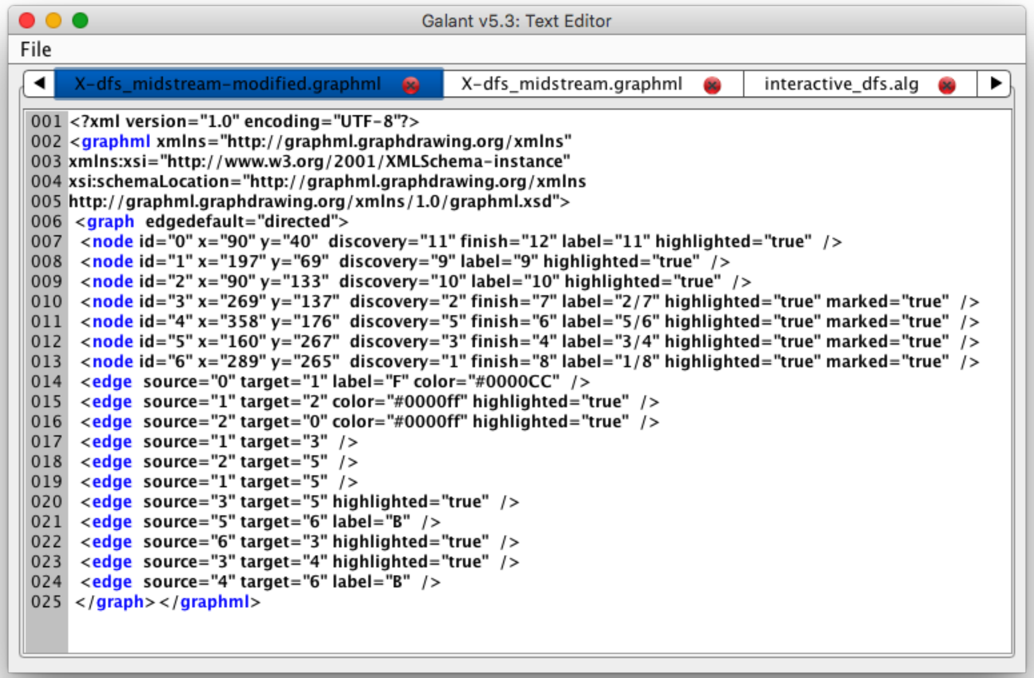
An improperly formatted GraphML file loaded from an external source will result in an error. Galant reports errors of all kinds (during reading of files, compilation of animation programs or execution of animations) by displaying a pop up window that allows the user to choose whether to continue (and usually return to a stable state) or quit the program. Error information, including a stack trace, is also displayed on the console.

The graph window, as illustrated at the top of Fig. 10, has a toolbar with four sections:

1. **Graph edit mode** – this includes the *select*, *create node*, *create edge*, and *delete* buttons. Only one button is active at any time; it determines the effect of a user's interaction (mouse


edit modes
select, create nodes/edges, delete
directedness
visibility
node/edge labels, node/edge weights
Galant v5.3: Graph Editor
File
force-directed drawing
No algorithm running
0
11
F
1
9
3
2/7
2
10
4
5/6
B
5
3/4
B
6
1/8
Label: F
Weight: 0
Color:
edit properties of selected edge (dashed)

Graph window: add/delete nodes and edges, move nodes, etc.



```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
 <graph  edgedefault="directed">
  <node id="0" x="90" y="40"  discovery="11" finish="12" label="11" highlighted="true"  />
  <node id="1" x="197" y="69"  discovery="9" label="9" highlighted="true"  />
  <node id="2" x="90" y="133"  discovery="10" label="10" highlighted="true"  />
  <node id="3" x="269" y="137"  discovery="2" finish="7" label="2/7" highlighted="true" marked="true"  />
  <node id="4" x="358" y="176"  discovery="5" finish="6" label="5/6" highlighted="true" marked="true"  />
  <node id="5" x="160" y="267"  discovery="3" finish="4" label="3/4" highlighted="true" marked="true"  />
  <node id="6" x="289" y="265"  discovery="1" finish="8" label="1/8" highlighted="true" marked="true"  />
  <edge  source="0" target="1" label="F" color="#0000CC"  />
  <edge  source="1" target="2" color="#0000ff" highlighted="true"  />
  <edge  source="2" target="0" color="#0000ff" highlighted="true"  />
  <edge  source="1" target="3"  />
  <edge  source="2" target="5"  />
  <edge  source="1" target="5"  />
  <edge  source="3" target="5" highlighted="true"  />
  <edge  source="5" target="6" label="B"  />
  <edge  source="6" target="3" highlighted="true"  />
  <edge  source="3" target="4" highlighted="true"  />
  <edge  source="4" target="6" label="B"  />
 </graph></graphml>
```

Text window: GraphML representation.

Figure 10: The Galant user interface.

clicking, dragging, etc.) with the window. if there are conflicts in selection of objects, nodes with higher id numbers have precedence (are above those with lower id numbers) and nodes have precedence over edges (are above edges).

- *Select.* A mouse click selects the graph component with highest precedence. If the component is a node, it is shaded light blue; if it's an edge, it becomes dashed. The in-line editor at the bottom of the graph window allows editing of the component's label, weight, and color.
- *Create node.* A node is created at the location of a mouse click if there is not already a node there. If another node is present it is simply selected.
- *Create edge.* Two clicks are required to create an edge. The first falls on the desired source node and the second on the target node. The line representing the edge is shown after the first click. If the first click does not land on a node, no edge is created. If the second click does not land on a node, creation of the edge is canceled.
- *Delete.* A mouse click deletes the highest-precedence component at the mouse location. If a node is deleted, all of its incident edges are deleted as well.

2. **Directedness toggles** – These change both the interpretation and the display of the graph between directed and undirected. Pressing the undirected (line without arrows) button causes all edges to be interpreted as undirected: this means that, when the code calls for all incoming/outgoing edges, all incident edges are used. Undirected edges are displayed as simple lines.

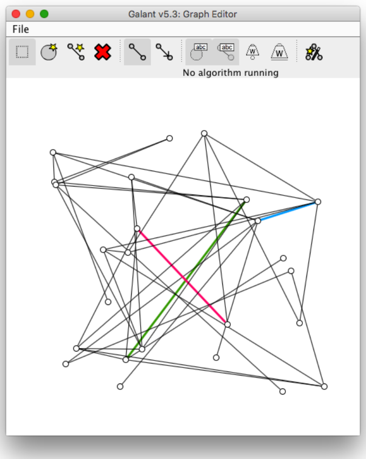
   Pressing the directed (line with arrow) button causes the macros for_incoming, for_outgoing, and for_adjacent to have three distinct meanings (they are all the same for undirected graphs): Incoming edges have the given node as target, outgoing as source, and adjacent applies to all incident edges.

3. **Display toggles** – The four display toggles turn on/off the display of node/edge labels and node/edge weights. A shaded toggle indicates that the corresponding display is *on*. When Galant is executed for the first time, all of these are *on*, but their setting persists from session to session. Labels and weights are also all displayed at the beginning of execution of an animation. The animation program can choose to hide labels and/or weights with simple directives. Hiding is usually unnecessary – the graphs that are subjects of the animations typically have only the desired attributes set.

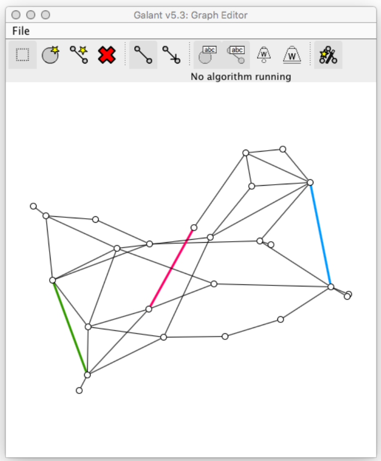
4. **Force directed drawing button** – Applies Hu's force directed algorithm [20] to the graph. Pushing the button a second time causes the drawing to revert to its previous state. Fig. 11 illustrates the use of force directed drawing to massage a randomly generated graph for convenient use in animations. The graph was generated randomly as a list of edges with weights and converted to graphml using a simple script. Galant, when reading the graph initially, assigned random positions to the nodes.

Keyboard shortcuts for graph editing operations are as follows:

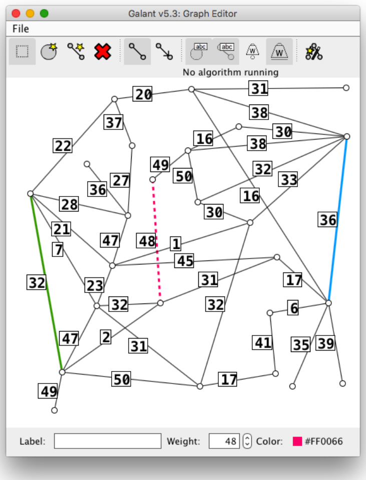
Ctrl-n	– create a new node in a random position
Ctrl-e	– create a new edge; user is prompted for id's of the nodes to be connected
Ctrl-i	– do a smart repositioning (force-directed) of nodes of the graph, useful when positions were chosen randomly
Del-n	– (hold delete key when typing n) delete a node; user is prompted for id
Del-e	– delete an edge; user is prompted for id's of the endpoints
Ctrl-ℓ	– toggle display of node labels
Ctrl-L	– toggle display of edge labels
Ctrl-w	– toggle display of node weights
Ctrl-W	– toggle display of edge weights



(a) A graph from an external random generator.



(b) Force directed layout applied.



Label: Weight: 48 Color: #FF0066

(c) User moved nodes for an even better layout.

Figure 11: Using force directed drawing to lay out graphs for more convenient editing and more compelling animations. Node ids do not appear because a node radius of 3 was specified using the Preferences→Graph Display panel.

B.4 Algorithm editing

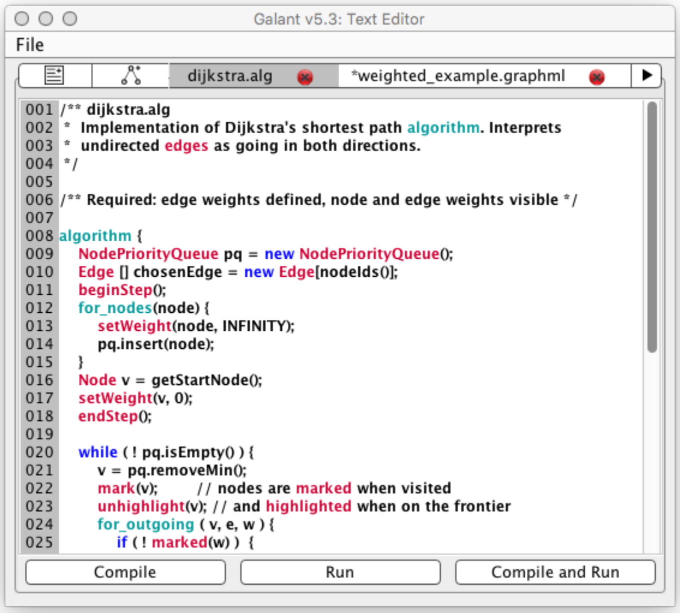
Algorithms can be edited in the text window. The editor uses Java keyword highlighting (default blue) and highlighting of Galant API fields and methods (default green). Since the current algorithm editor is fairly primitive (no search and replace, for example), it is more efficient to edit animation code offline using a program editor – for example emacs with Java mode turned on. The Galant editor is, however, useful for locating and correcting minor errors. For more details on how to compose animation code, see the programmer guide (Section C).

B.5 Animating algorithms

To animate an algorithm the code for it must be compiled and then run via the algorithm controls – the bottom tabs on the text window shown in Fig. 12. The algorithm runs on the *active graph*, the one currently displayed on the graph window, also shown in Fig. 12. While the algorithm is running, its text is grayed out.[8] If there are errors in compilation these will show up on the console (terminal from which Galant was run) and in a dialog box that allows the user to ask for more details and decide whether to exit Galant or not. The console also displays the what the code looks like after macro replacement in case obscure errors were the result of unexpected macro expansion. Line numbers in the macro-expanded code match those of the original so that all errors reported by the Java compiler will refer to the correct line number in the original Galant code. Runtime errors also open the above mentioned dialog box.

When the user initiates execution of an animation by pushing the Run button the animation program steps forward until displays the next animation event or, if a beginStep() call has marked the start of a sequence of events, until it reaches the next endStep() call. It then pauses execution and waits for the user to decide whether to step forward, step backward, or exit. A step forward resumes execution while a step backward returns the display to a previous state. The algorithm resumes execution

[8] This may not be desirable. The intent was to indicate to the user that operations on the text window are not desired.



```
/** dijkstra.alg
 *  Implementation of Dijkstra's shortest path algorithm. Interprets
 *  undirected edges as going in both directions.
 */

/** Required: edge weights defined, node and edge weights visible */

algorithm {
   NodePriorityQueue pq = new NodePriorityQueue();
   Edge [] chosenEdge = new Edge[nodeIds()];
   beginStep();
   for_nodes(node) {
      setWeight(node, INFINITY);
      pq.insert(node);
   }
   Node v = getStartNode();
   setWeight(v, 0);
   endStep();

   while ( ! pq.isEmpty() ) {
      v = pq.removeMin();
      mark(v);       // nodes are marked when visited
      unhighlight(v); // and highlighted when on the frontier
      for_outgoing ( v, e, w ) {
         if ( ! marked(w) ) {
```

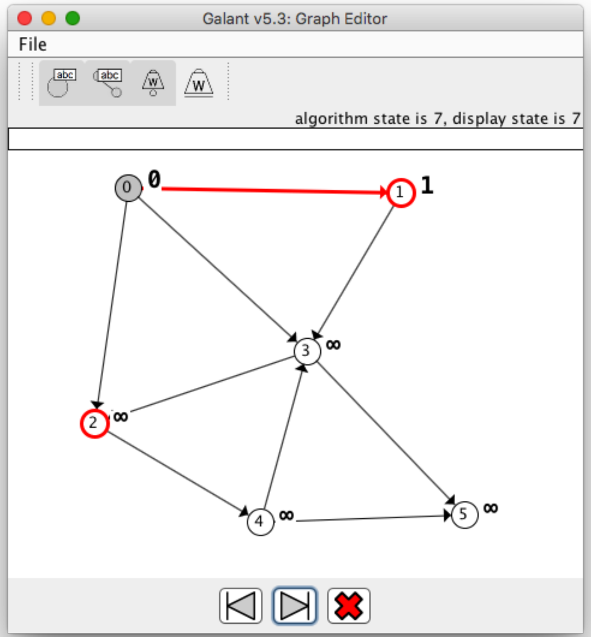

Galant v5.3: Graph Editor
File
algorithm state is 7, display state is 7
0
0
1
1
2
∞
3
∞
4
∞
5
∞

Figure 12: The text and graph windows when an algorithm is running. The text window is at the top – it has buttons for the user to select whether to compile the algorithm, run it (if already compiled) or do both. The graph window in the middle of algorithm execution is at the bottom.

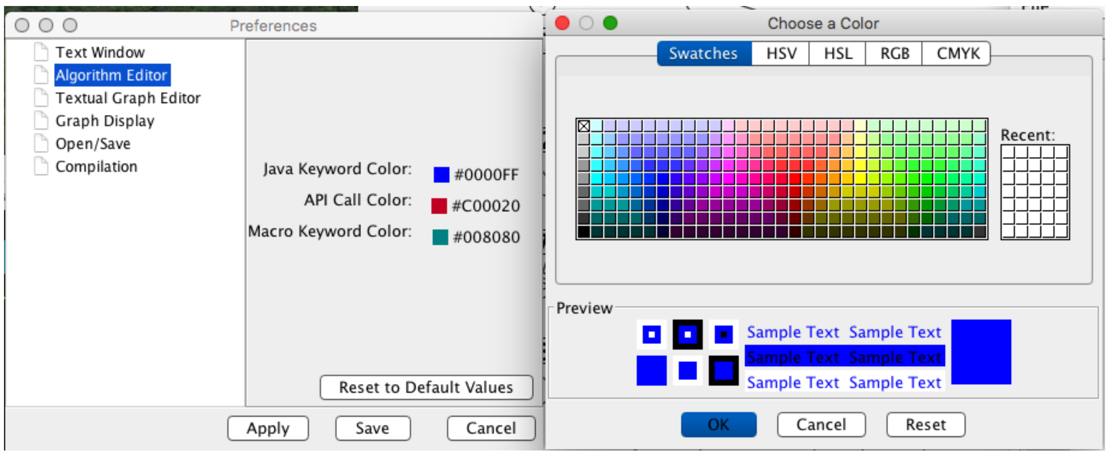
only when the *display state* indicated by the user's sequence of forward and backward steps ($f - b$, where f is the number of forward and b the number of backward steps) exceeds the *algorithm state*, the number of animation steps the algorithm has executed. The user controls forward and backward steps using either the buttons at the bottom of the graph window (shown in Fig. 12) or the right/left arrow keys. During the execution of the animation, all graph editing functions are disabled. These are re-enabled when the user exits the animation by pressing the red **X** button or the Esc (escape) key on the terminal.

B.6 Preferences

Galant preferences can be accessed via the File-¿Preferences menu item or by using the keyboard shortcut Ctrl-P (Cmd-P for Mac). Preferences that can be edited are:

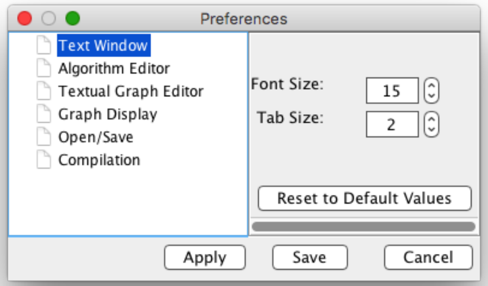
- Default directories for opening and saving files (Open/Save).
- Directory where compiled animation code is stored (Compilation).
- Font size and tab size for text window editing (Editors).
- Colors for keyword and Galant API highlighting (Algorithm Editor).
- Color for GraphML highlighting (Textual Graph Editor).
- Node radius (Graph Display); when the radius is below a threshold (9 pixels), node id's are not displayed; this is useful when running animations on large graphs.
- Edge width (Graph Display).

Fig. 13 shows the four preference panels of most interest to the user.



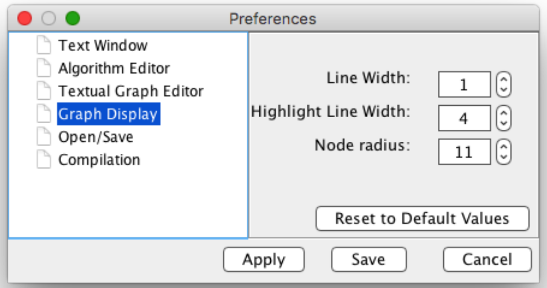
Java Keyword Color: 🟦 #0000FF
API Call Color: 🟥 #C00020
Macro Keyword Color: 🟩 #008080

(a) syntax highlight colors – same color chooser as for editing node/edge colors



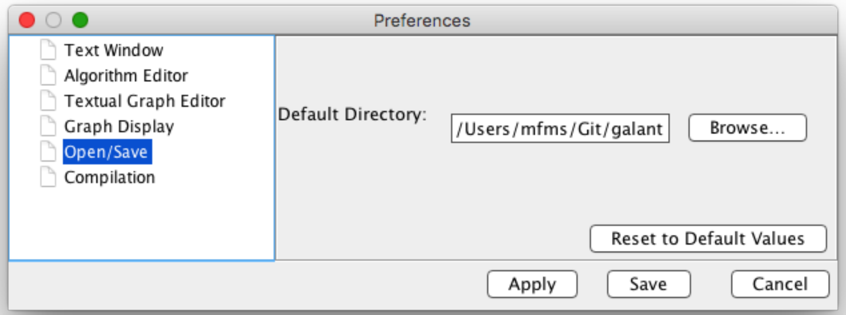
Font Size: 15
Tab Size: 2

(b) text font size and tab spaces



Line Width: 1
Highlight Line Width: 4
Node radius: 11

(c) line widths and node radius



Default Directory: /Users/mfms/Git/galant

(d) default directory for opening and saving files

Figure 13: The four most important preference panels.

C Programmer guide

Animation programmers can write algorithms in notation that resembles textbook pseudocode in files that have a .alg extension. The animation examples have used procedural syntax for function calls, as in, for example, setWeight(v,0). Java (object oriented) syntax can also be used: v.setWeight(0). A key advantage of Galant is that a seasoned Java programmer can not only use the Java syntax but can also augment Galant algorithms with arbitrary Java classes defined externally, using import statements. All Galant code is effectively Java, either natively, or via macro preprocessing.

The text panel provides a crude editor for algorithms (as well as GraphML descriptions of graphs); its limited capabilities make it useful primarily for fine tuning and error correction. The first author uses emacs in Java mode to edit algorithms offline, for example, not a major inconvenience – it is easy to reload algorithms when they are modified without exiting Galant. The Galant editor is also useful in that it provides syntax highlighting of Galant functions and macros.

The source code for an algorithm begins with any number (including none) of global variable declarations and function definitions. The animator can import code from other sources using appropriate import statements; these must occur at the very beginning. The code for the algorithm itself follows, starting with the keyword algorithm. A *code block* is a sequence of statements, each terminated by a semicolon, just as in Java. The main algorithm has the form

```
algorithm {
    code block
}
```

Declarations of global variables are also like those of Java:
type variable_name;
or
type [] *variable_name*;
to declare an array of the given type. All variables must be initialized either within a function definition or in the algorithm. The Java incantation
type variable_name = new *type*[*size*]
is used to initialize an array with *size* elements initialized to null or 0. Arrays use 0-based indexing: the largest index is $size - 1$. Detailed information about function declarations is in Section C.2.1 below.

Central to the Galant API is the Graph object: currently all other parts of the API refer to it. The components of a graph are declared to be of type Node or Edge and can be accessed/modified via a variety of functions/methods. When an observer or explorer interacts with the animation they move either forward or backward one step at a time. All aspects of the graph API therefore refer to the current *state of the graph*, the set of states behaving as a stack. API calls that change the state of a node or an edge automatically generate a next step, but the programmer can override this using a beginStep() and endStep() pair. For example, the beginning of our implementation of Dijkstra's algorithm looks like

```
beginStep();
for_nodes(node) {
    setWeight(node, INFINITY);
    nodePQ.add(node);
}
endStep();
```

Without the beginStep/endStep override, this initialization would require the observer to click through multiple steps (one for each node) before getting to the interesting part of the animation. For convenience the function step() is a synomym for endStep(); beginStep(). If a step takes longer than 5 seconds, the program is terminated under the presumption that there may be an infinite loop.

List⟨Node⟩ getNodes() NodeList getNodes()	returns a list of the nodes of the graph; type NodeList is built into Galant and essentially equivalent to the Java List⟨Node⟩; see Table 6
List⟨Edge⟩ edges() EdgeList edges()	returns a list of edges of the graph; return type EdgeList is analogous to NodeList
for_nodes(v) { *code block* }	equivalent to for (Node v : nodes()) { *code block* }; the statements are executed for each node v
for_edges(e) { *code block* }	analogous to for_nodes
Integer numberOfNodes()	returns the number of nodes
Integer numberOfEdges()	returns the number of edges
int id(Node v), int id(Edge e)	returns the unique identifier of v or e
int nodeIds(), int edgeIds()	returns the largest node/edge identifier plus one; useful when an array is to be indexed using node/edge identifiers, since these are not necessarily contiguous
source(Edge e), target(Edge e)	returns the source/target of edge e, sometimes called the (arrow) tail/head or source/destination
Integer degree(Node v) Integer indegree(Node v) Integer outdegree(Node v)	the number of edges incident on v, total, incoming and outgoing; if the graph is undirected, the outdegree is the same as the degree
EdgeList edges(Node v) EdgeList inEdges(Node v) EdgeList outEdges(Node v)	returns a list of v's incident, incoming or outgoing edges, respectively; outgoing edges are the same as incident edges if the graph is undirected
Node otherEnd(Edge e, Node v) Node otherEnd(Node v, Edge e)	returns the node opposite v on edge e; if v is the source otherEnd returns the target and vice-versa
NodeList neighbors(Node v)	returns a list of nodes adjacent to v

for_adjacent(v, e, w) { *code block* }
for_incoming(v, e, w) { *code block* }
for_outgoing(v, e, w) { *code block* }
	for_adjacent executes the code block for each edge e incident on v, where w is otherEnd(e,v); v must already be declared but e and w are declared by the macro; the other two are analogous for incoming and outgoing edges

getStartNode()	returns the first node in the list of nodes, typically the one with smallest id; used by algorithms that require a start node
isDirected()	returns true if the graph is directed
setDirected(boolean directed)	makes the graph directed or undirected depending on whether `directed` is true or false, respectively
Node addNode() Node addNode(Integer x, Integer y)	returns a new node and adds it to the list of nodes; the id is the smallest integer not currently in use as an id; attributes such as weight, label and position are absent and must be set explicitly by appropriate method calls; the second version puts the node at position (x,y), where x and y are pixel coordinates.

addEdge(Node source, Node target)
addEdge(int sourceId, int targetId)
	adds an edge from the source to the target (source and target are interchangeable when graph is undirected); the second variation specifies id's of the nodes to be connected; as in the case of adding a node, the edge is added to the list of edges and its weight and label are absent

deleteNode(Node v)	removes node v and its incident edges from the graph
deleteEdge(Edge e)	removes edge e from the graph

Table 1: Functions and macros that apply to the structure of a graph.

print(String s)	prints s on the console; useful for debugging
display(String s)	writes the string s at the top of the window
String getMessage()	returns the message currently displayed on the message banner
error(String s)	prints s on the console with a stack trace; also displays s in popup window with an option to view the stack trace; the algorithm terminates and the user can choose whether to terminate Galant entirely or continue interacting
beginStep(), endStep(), step()	any actions between a beginStep() and an endStep() take place atomically, i.e., all in a single "step forward" action by the user; step() is a synonym for endStep(); beginStep()
Node getNode(String message)	pops up a window with the given message and prompts the user to enter the identifier of a node, which is returned; if no node with that id exists, an error popup is displayed and the user is prompted again
Edge getEdge(String message)	pops up a window with the given message and prompts the user to enter the identifiers of two nodes, the endpoints of an edge, which is returned; if either id has no corresponding node or the the two nodes are not connect by an edge (in the right direction if the graph is directed), an error popup is displayed and the user is prompted again
Node getNode(String p, NodeSet s, String e) Edge getEdge(String p, EdgeSet s, String e)	variations of getNode and getEdge; here p is the prompt, s is the set from which the node or edge must be chosen and e an error message if the node/edge does not belong to the specified set; useful when wanting to specify an adjacent node or an outgoing edge
String getString(String message) Integer getInteger(String message) Double getReal(String message)	analogous to getNode and getEdge; allow algorithm to engage in dialog with the user
Integer integer(String s) Double real(String s)	performs conversion from a string to an integer/double; useful when parsing labels that represent numbers
windowWidth(), windowHeight()	current width and height of the window, in case the algorithm wants to rescale the graph

Table 2: Utility functions.

Functions and macros for the graph as a whole are shown in Table 1, while Table 2 lists some algorithm functions not related to any aspect of a graph.

The nodes and edges, of type Node and Edge, respectively, are subtypes/extensions of GraphElement. Arbitrary attributes can be assigned to each graph element. In the GraphML file these show up as, for example,

<node *attribute_1*="*value_1*" ... *attribute_k*="*value_k*" />

Each node and edge has a unique integer id. The id's are assigned consecutively as nodes/edges are created; they may not be altered. The id of a node or edge can be accessed via the id() function. Often, as is the case with the depth-first search algorithm, it makes sense to use arrays indexed by node or edge id's. Since graphs may be generated externally and/or have undergone deletions of nodes or edges, the id's are not always contiguous. The functions nodeIds() and edgeIds() return the size of an array large enough to accommodate the appropriate id's as indexes. So code such as

```
        Integer myArray[] = new Integer[nodeIds()];
        for_nodes(v) { myArray[id(v)] = 1; }
}
```

is immune to array out of bounds errors.

C.1 Node and edge methods

Nodes and edges have 'getters' and 'setters' for a variety of attributes, i.e.,
set$a$($\langle a's$ type$\rangle$ x)
and
$\langle a's$ type$\rangle$ get$a$(), where a is the name of an attribute such as Color, Label or Weight. A more convenient way to access these standard attributes omits the prefix get and uses procedural syntax: color(x) is a synonym for x.getColor(), for example. Procedural syntax for the setters is also available: setColor(x,c) is a synonym for x.setColor(c). In the special cases of color and label it is possible to omit the set (since it reads naturally): color(x,c) instead of setColor(x,c).

C.1.1 Logical attributes: functions and macros

Nodes. From a node's point of view we would like information about the adjacent nodes and incident edges. The relevant *methods* require the use of Java generics, but macros are provided to simplify graph API access. The macros, which have equivalents in GDR, are:

- for_adjacent(x, e, y){ *code block* } executes the statements in the code block for each edge incident on x. The statements can refer to x, or e, the current incident edge, or y, the other endpoint of e. The macro assumes that x has already been declared as Node but e and y are declared automatically.
- for_outgoing(Node x, Edge e, Node y){ *code block* }
  behaves like for_adjacent except that, when the graph is directed, it iterates only over the edges whose source is x (it still iterates over all the edges when the graph is undirected).
- for_incoming(Node x, Edge e, Node y){ *code block* }
  behaves like for_adjacent except that, when the graph is directed, it iterates only over the edges whose sink is x (it still iterates over all the edges when the graph is undirected).

The actual API methods hiding behind these macros are (these are Node methods):

- List⟨Edge⟩ getIncidentEdges() returns a list of all edges incident to this node, both incoming and outgoing.
- List⟨Edge⟩ getOutgoingEdges() returns a list of edges directed away from this node (all incident edges if the graph is undirected).
- List⟨Edge⟩ getIncomingEdges() returns a list of edges directed toward this node (all incident edges if the graph is undirected).
- Node travel(Edge e) returns the other endpoint of e.

The above all use Java syntax, as in v.travel(e). The following are node-related functions with procedural syntax.

- degree(v), indegree(v) and outdegree(v) return the appropriate integers.
- otherEnd(v, e), where v is a node and e is an edge returns node w such that e connects v and w; the programmer can also say otherEnd(e, v) in case she forgets the order of the arguments.
- neighbors(v) returns a list of the nodes adjacent to node v.

Edges. The logical attributes of an edge are its source and target (destination).

- setSourceNode(Node) and Node getSourceNode()
- setTargetNode(Node) and Node getTargetNode()
- getOtherEndPoint(Node u) returns v where this edge is either uv or vu.

Graph Elements. Nodes and edges both have a mechanism for setting (and getting) arbitrary attributes of type Integer, String, and Double. the relevant methods are
setIntegerAttribute(String key,Integer value)

to associate an integer value with a node and
Integer getIntegerAttribute(String key)
to retrieve it. String and Double attributes work the same way as integer attributes. These are useful when an algorithm requires arbitrary information to be associated with nodes and/or edges. The user-defined attributes may differ from one node or edge to the next. For example, some nodes may have a depth attribute while others do not.

C.1.2 Geometric attributes

Currently, the only geometric attributes are the positions of the nodes. Unlike GDR, the edges in Galant are all straight lines and the positions of their labels are fixed. The relevant methods for nodes – using procedural syntax – are int getX(Node), int getY(Node) and Point getPosition(Node) for the 'getters'. To set a position, one should use
setPosition(Node,Point)
or
setPosition(Node,int,int).
Once a node has an established position, it is possible to change only one coordinate using setX(Node,int) or setY(Node,int). Object-oriented variants of all of these, e.g., v.setX(100), are also available.

Ordinarily the user can move nodes during algorithm execution and the resulting positions persist after execution terminates. For some algorithms, such as sorting, the algorithm itself needs to move nodes. It is desirable then to keep the user from moving nodes. The declaration movesNodes() at the beginning of an algorithm accomplishes this.

C.1.3 Display attributes

Each node and edge has both a (double) weight and a label. The weight is also a logical attribute in that it is used implicitly as a key for sorting and priority queues. The label is simply text and may be interpreted however the programmer chooses. The conversion functions integer(String) and real(String) – see Table 2 – provide a convenient mechanism for treating labels as objects of class Integer or Double, respectively. The argument of setLabel and its relatives – see below – is not the expected String but Object; any type of object that has a Java toString method will work – numbers have to be of type Integer or Double rather than int or double since the latter are not objects in Java.[9] So conversion between string labels and numbers works both ways.

Aside from the setters and getters: setWeight(double), Double getWeight(), setLabel(Object) and String getLabel(), the programmer can also manipulate and test for the absence of weights/labels using clearWeight() and boolean hasWeight(), and the corresponding methods for labels. The procedural variants in this case are setWeight(Node,double), Double weight(Node),[10] label(Node,Object),[11] and String label(Node)

Nodes can either be plain, highlighted (selected), marked (visited) or both highlighted and marked. Being highlighted alters the the boundary (color and thickness) of a node (as controlled by the implementation), while being marked affects the fill color. Edges can be plain or selected, with thickness and color modified in the latter case.

The relevant methods are (here Element refers to either a Node or an Edge):

- highlight(Element), unhighlight(Element) and Boolean isHighlighted(Element)
- correspondingly, setSelected(true), setSelected(false), and boolean isSelected()
- mark(Node), unmark(Node) and Boolean isMarked(Node), equivalently Boolean marked(Node).

Although the specific colors for displaying the outlines of nodes or the lines representing edges are predetermined for plain and highlighted nodes/edges, the animation implementation can modify

[9] Galant functions return objects, Integer or Double, when return values are numbers for this reason.
[10] The *get* is omitted here for more natural syntax.
[11] A natural syntax that resembles English. However, setLabel(Node,Object) is also allowed.

RED	= "#ff0000"
BLUE	= "#00ff00"
GREEN	= "#0000ff"
YELLOW	= "#ffff00"
MAGENTA	= "#ff00ff"
CYAN	= "#00ffff"
TEAL	= "#009999"
VIOLET	= "#9900cc"
ORANGE	= "#ff8000"
GRAY	= "#808080"
BLACK	= "#000000"
WHITE	= "#ffffff"

Table 3: Predefined color constants.

these colors, thus allowing for many different kinds of highlighting. The getColor and setColor methods and their procedural variants have String arguments in the RGB format #RRGGBB; for example, the string #0000ff is blue. The predefined color constants are listed in Table 3. Note: In the graph display *highlighting takes precedence over color*; if a node is highlighted, its color is ignored and the default highlight color is used.

Special handling is required when any attribute is nonexistent or has a null value – these two are equivalent. When displayed in the graph window, nonexistent labels and weights simply do not show up while nonexistent colors are rendered as thin black lines (thickness determined by user preference). In an animation program, however, nonexistent attributes are handled differently.

- color() returns null as expected
- label() returns an empty string; this ensures that it is always safe to use a label in an expression calling for a string
- weight() throws an exception; there is no obvious default weight; a program can test for the presence/absence of a weight using the hasWeight() or hasNoWeight() methods

Of the attributes listed above, weight, label, color and position can be accessed and modified by the user as well as the program. In all cases, modifications by execution of the animation are ephemeral – the graph returns to its original state after execution. The user can save the mid-execution state of the graph: select the Export option on the file menu of the *graph* window.

A summary of functions relevant to node and edge attributes (their procedural versions) is given in Table 4.

C.1.4 Global access for individual node/edge attributes and graph attributes

It is sometimes useful to access or manipulate attributes of nodes and edges globally. For example, an algorithm might want to hide node weights entirely because they are not relevant or hide them initially and reveal them for individual nodes as the algorithm progresses. These functionalities can be accomplished by hideNodeWeights or hideAllNodeWeights, respectively. A summary of these capabilities is given in Table 5.

C.2 Additional programmer information

A Galant algorithm/program is executed as a method within a Java class. In order to shield the Galant programmer from Java idiosyncrasies, some features have been added.

id(*element*)	returns the unique identifier of the node or edge
source(Edge e), target(Edge e)	returns the source/target of edge e, sometimes called the (arrow) tail/head or source/destination
mark(Node v), unmark(Node v)	shades the interior of a node or undoes that
Boolean marked(Node v)	returns true if the node is marked
highlight(*element*), unhighlight(*element*)	makes the node or edge highlighted, i.e., thickens the border or line and makes it red / undoes the highlighting
Boolean highlighted(*element*)	returns true if the node or edge is highlighted
select(*element*), deselect(*element*) selected(*element*)	synonyms for highlight, unhighlight and highlighted
Double weight(*element*) setWeight(*element*, double weight)	get/set the weight of the element
showWeight(*element*), hideWeight(*element*) Boolean weightIsVisible(*element*) Boolean weightIsHidden(*element*)	make the weight of the element visible/invisible, query their visibility; weights of the element type have to be globally visible – see Table 5 – for showWeight to have an effect
String label(*element*) label(*element*, Object obj)	get/set the label of the element, the Object argument allows an object of any other type to be converted to a (String) label, as long as there is a toString method, which is true of all major classes (you have to be careful, for example, to use Integer instead of int)
showLabel(*element*), hideLabel(*element*) `Boolean labelIsVisible(`*element*`)` `Boolean labelIsHidden(`*element*`)`	analogous to the corresponding weight functions
hide(*element*), show(*element*) `Boolean hidden(`*element*`)` `Boolean visible(`*element*`)`	makes nodes/edges disappear/reappear and tests whether they are visible or hidden; useful when an algorithm (logically) deletes objects, but they need to be revealed again upon completion
String color(*element*) color(*element*, String c) uncolor(*element*)	get/set/remove the color of the border of a node or line representing an edge; colors are encoded as strings of the form "#RRBBGG", the RR, BB and GG being hexadecimal numbers representing the red, blue and green components of the color, respectively; see Table 3 for a list of predefined colors; when an element has no color, the line is thinner and black
boolean set(*element*, String key, ⟨*type*⟩ value)	sets an arbitrary attribute, key, of the element to have a value of a given type, where the type is one of Integer, Double, Boolean or String; in the special case of Boolean the third argument may be omitted and defaults to true; so set(v,"attr") is equivalent to set(v,"attr",true); returns true if the element already has a value for the given attribute, false otherwise
boolean clear(*element*, String key)	removes the attribute key from the element; if the `key` refers to a Boolean attribute, this is logically equivalent to making it false
⟨*type*⟩ get⟨*type*⟩(*element*, String key) Boolean is(*element*, String key)	returns the value associated with key or null if the graph has no value of the given type for key, i.e., if no set(String key, ⟨*type*⟩ value) has occurred; in the special case of a Boolean attribute, the second formulation may be used; the object-oriented syntax, such as e.is("inTree"), sometimes reads more naturally

Table 4: Functions that query and manipulate attributes of individual nodes and edges. Here, *element* refers to either a Node or an Edge, both the type and the formal parameter.

Boolean nodeLabelsAreVisible() Boolean edgeLabelsAreVisible() Boolean nodeWeightsAreVisible() Boolean edgeWeightsAreVisible()	returns true if node/edge labels/weights are *globally* visible, the default state, which can be altered by hideNodeLabels(), etc., defined below

hideNodeLabels(), hideEdgeLabels()
hideNodeWeights(), hideEdgeWeights()
	hides all node/edge labels/weights; typically used at the beginning of an algorithm to hide unnecessary information; labels and weights are shown by default

showNodeLabels(), showEdgeLabels()
showNodeWeights(), showEdgeWeights()
	undoes the hiding of labels/weights

hideAllNodeLabels()
hideAllEdgeLabels()
hideAllNodeWeights()
hideAllEdgeWeights()
	hides all node/edge labels/weights even if they are visible globally by default or by showNodeLabels(), etc., or for individual nodes and edges; in order for the label or weight of a node/edge to be displayed, labels/weights must be visible globally and its label/weight must be visible; initially, all labels/weights are visible, both globally and for individual nodes/edges; these functions are used to hide information so that it can be revealed subsequently, one node or edge at a time

showAllNodeLabels()
showAllEdgeLabels()
showAllNodeWeights()
showAllEdgeWeights()
	makes all individual node/edge weights/labels visible if they are globally visible by default or via showNodeLabels(), etc.; this undoes the effect of hideAllNodeLabels(), etc., and of any individual hiding of labels/weights

clearNodeLabels(), clearEdgeLabels()
clearNodeWeights(), clearEdgeWeights()
	gets rid of all node/edge labels/weights; this not only makes them invisible, but also erases whatever values they have

showNodes(), showEdges()	undo any hiding of nodes/edges that has taken place during the algorithm
NodeSet visibleNodes() EdgeSet visibleEdges()	return the set of nodes/edges that are not hidden

clearNodeMarks()
clearNodeHighlighting()
clearEdgeHighlighting()
	unmarks all nodes, unhighlights all nodes/edges, respectively

clearNodeLabels()
clearNodeWeights()
clearEdgeLabels()
clearEdgeWeights()
	erases labels/weights of all nodes/edges; useful if an algorithm needs to start with a clean slate with respect to any of these attributes

clearAllNode(String attribute)
clearAllEdge(String attribute)
	erases values of the given attribute from all nodes/edges, a generalization of clearNodeLabels, etc.

boolean set(String attribute, ⟨*type*⟩ value)	sets an arbitrary attribute of the graph to have a value of a given type, where the type is one of Integer, Double, Boolean or String; in the special case of Boolean the second argument may be omitted and defaults to true; so set("attr") is equivalent to set("attr",true); returns true if the graph already has a value for the given attribute, false otherwise
⟨*type*⟩ get⟨*type*⟩(String attribute) Boolean is(String attribute)	returns the value associated with attribute or null if the graph has no value of the given type for attribute, i.e., if no set(String attribute, ⟨*type*⟩ value) has occurred; in the special case of a Boolean attribute, the second formulation may be used

clearAllNode(String attribute)
clearAllEdge(String attribute)
	erases the value of the given attribute for all nodes/edges

Table 5: Functions that query and manipulate graph node and edge attributes globally, i.e., for all nodes or edges at once. Also included are functions that deal with graph attributes.

C.2.1 Definition of Functions/Methods

A programmer can define arbitrary functions (methods) using the construct

```
function [return_type] name ( parameters ) {
        code block
}
```

The behavior is like that of a Java method. So, for example,

```
function int plus( int a, int b ) {
    return a + b;
}
```

is equivalent to

```
static int plus( int a, int b ) {
    return a + b;
}
```

The *return_type* is optional. If it is missing, the function behaves like a void method in Java. An example is the recursive
function visit(Node v) { *code* }
The conversion of functions into Java methods when Galant code is compiled is complex and may result in indecipherable error messages.

C.2.2 Data Structures

Galant provides some standard data structures for nodes and edges automatically. These are described in detail in Table 6. Data structures use object-oriented syntax. For example, to add a node v to a NodeList L, the appropriate syntax is L.add(v). Most data structure operations are as efficient as one might expect. A notable exception is decreaseKey for priority queues. The operation pq.decreaseKey(v,k) does pq.remove(v) followed by pq.add(v). The new key is implicit – it is the weight of the element. In practice, assuming pq is a NodePriorityQueue, v is a Node and k is a Double, the sequence would be

```
setWeight(v, k);
pq.decreaseKey(v,k);
```

which would translate to

```
setWeight(v, k);
pq.remove(v);
pq.add(v)
```

As pointed out above, the weight of a node or edge is used for priority in a priority queue or for sorting. The programmer can change this default by redefining the Java compareTo() method, but this requires finesse and Java expertise.

NodeList and EdgeList: lists of nodes or edges, respectively. We use *list* as shorthand for either NodeList L or EdgeList L, *type* for either Node or Edge and *element* for Node v or Edge e.

type first(*list*)	returns the first element of this list
add(*element*, *list*)	adds the element to the end of the list; along with first we get the effect of a queue
remove(*element*, *list*)	removes the first occurrence of the element from the list
sort(*list*)	use the weights of the nodes/edges to sort the list L; sort is actually a macro that invokes Java Collections.sort(L); this means that L can be any collection (list, stack, queue, set) of comparable elements

NodeQueue and EdgeQueue: queues of nodes or edges, respectively. Same conventions as for lists.

void enqueue(*element*)	adds the element to the rear of the queue
type dequeue()	returns and removes the element at the front of the queue; returns null if the queue is empty
type remove()	returns and removes the element at the front of the queue; throws an exception if the queue is empty
type element()	returns the element at the front of the queue without removing it; throws an exception if the queue is empty
type peek()	returns the element at the front of the queue without removing it; returns null if the queue is empty
size()	returns the number of elements in the queue
isEmpty()	returns true if the queue is empty

NodeStack and EdgeStack: stacks of nodes or edges, respectively.

void push(*element*)	adds the element to the top of the stack
type pop()	returns and removes the element at the top of the stack; throws an exception if the stack is empty
type peek()	returns the element at the top of the stack without removing it; returns null if the stack is empty
size(), isEmpty()	analogous to the corresponding queue methods

NodePriorityQueue and EdgePriorityQueue: priority queues of nodes or edges, respectively.

void add(*element*)	adds the element to the priority queue; the priority is defined to be its weight – see Section C.1.3
type removeMin()	returns and removes the element with minimum weight; returns null if the queue is empty
boolean remove(*element*)	removes the element; returns true if it is present, false otherwise
void decreaseKey(*element*, double key)	changes the weight of the element to the new key and reorganizes the priority queue appropriately; since this is accomplished by removing and reinserting the object, i.e., inefficiently, this method can also be used to increase the key
size(), isEmpty()	analogous to the corresponding queue methods

NodeSet and EdgeSet: sets of nodes or edges, respectively.

boolean add(*element*)	adds the element to the set; returns true if the element was a new addition, false otherwise
boolean remove(*element*)	removes the element returns true if the element was present, false otherwise
boolean contains(*element*)	returns true if the element is in the set
size(), isEmpty()	analogous to the corresponding methods for other data structures

Table 6: Built-in data structures and their methods. These methods use object-oriented syntax: $\langle structure\rangle.\langle method\rangle(\langle arguments\rangle)$ and are created using, e.g., NodeQueue Q = new NodeQueue(); the new operator in Java.

D Known Bugs and Annoyances

Input/output

- When you save the current state of an animation using `File->Export` on the graph panel some of the information might not be saved. In particular, weights and labels are saved, but highlighting is not (since it is not equivalent to an actual color change of a node or edge).

Text editing (of programs or graphs)[12]

- Tabs for graphs and algorithms are often hard to deal with: (a) if you reread a graph or algorithm, it appears twice; (b) you can only run an algorithm on a graph if the tabs for the two appear at the top of the window at the same time – this may be impossible if there are other intervening graphs and algorithms and the window is not wide enough.
- A text panel for a graph is often marked as having been changed when this is not really the case. This will happen if an algorithm runs on the graph and sometimes immediately when the graph is read from a file.
- If you attempt to do anything in the text window (File menu or tabs) while an algorithm is running, Galant hangs; it appears that you can exit (quit) from the file menu of the graph window, however. Oddly, when Galant hangs, you can bring it back to life by executing a command in a separate terminal window (at least this works on a Mac).
- Some preferences, such as syntax highlight colors, require the user to exit Galant before they take effect.
- When saving a file, Galant complains if the extension is not correct (.graphml or .alg) but does not fill it in automatically.

Graph editing (in graph panel or via keyboard shortcuts)

- Nodes have to be moved individually. In a large graph there is no way to select a collection of nodes and move them all at once.
- The force directed layout algorithm clusters nodes too close to each other when there are cliques or near cliques.
- Semantics of force directed layout when combined with adding edges is not intuitive (force directed layout takes over if the button is pressed, so adding nodes/edges is dependent on the state of the button). It's generally a bad idea to change the graph when the smart reposition button is pressed.
- It is not possible to change the thickness of an edge or node boundary directly from the editor or an algorithm nor is it possible to change the fill color of a node. The only way to change these properties is via highlighting, selecting, and marking nodes/edges during the animation.
- Once you choose a color for a node in the editor, you can't uncolor it in the editor. The best you can do is set it to black, but then it appears thicker. However, the algorithm strip_attributes.alg can be run to reset colors and other nonessential attributes. You can save the cleaned up graph using File→Export in the graph window.
- If a node is selected for editing (other than immediately on creation) its weight is set to 0 when the spinner shows up. Initially a node has no weight at all; this should continue to be the case unless the user specifies a weight.
- When user creates a new node/edge via keyboard shortcut, there is no obvious way to enter the weight and label (except to click in the appropriate text field).

[12] Galant's editor is primitive, but programs can easily be edited externally. Text representations of graphs can either be edited or generated externally.

Compilation and execution

- When there are compilation errors the user cannot scroll the text window (or make modifications in it) while the popup window showing the errors is displayed. There are two possible workarounds: (i) open the algorithm in an external editor, or (ii) view the error messages in the console.
- Every once in a while an exception occurs when an error-free algorithm is executed or when Galant initially fires up, but it is possible to step through the animation normally after hitting the Continue button.
- When an algorithm controls visibility of node/edge labels or weights and the user overrides in the middle of execution, Galant sometimes freezes when user terminates the algorithm. This appears to happen more frequently if user does a lot of fast forward/reverse between visibility changes. The problem does not seem to occur if user toggles visibility via keyboard shortcuts.
- Compiler error messages can be cryptic (but at least they refer to the correct line numbers). Because of the macro preprocessing, it may be necessary to look at the console to get an idea of what is causing a particular error. If the parentheses/brackets/braces inside a function definition or body of one of the for_... macros are unbalanced, the macro preprocessor will simply report the fact with no indication of the location of the error except for an excerpt from the beginning of the body.
- Line numbers do, however, get out of sync if the header of a function declaration takes up more than one line. For example, in
  ```
  function foo(Node v,
               Node w,
               Edge e) {
  }
  ```
  The first three lines are treated as one.
- If a macro is used incorrectly, the preprocessor does not report a line number.
- After hitting Enter or Return at the end of a query, user still needs to step forward to do the next step of the algorithm.
- There is no way to execute the animation in a continuous fashion with a controllable speed. The current workaround is the use of arrow keys as keyboard shortcuts for stepping forward or backward – these can be held down to generate multiple steps in rapid succession, but finer grained control is difficult.

E Feature Requests

Some features under development for the next major release are listed here.

1. Eventually lists, queues, stacks and priority queues of nodes and edges will all be concrete and all graph, node and edge methods that return containers will return one of these. For example, `getNodes()` will return a `NodeList` instead of a `List<Node>`.

2. *Mode-less graph editing.* In place of the GDR-like mechanism, where panel buttons are used to determine how the graph editor responds to mouse actions – a click might create a node, initiate an edge or select an object – the Ctrl and Shift keys can determine the "mode". So, for example, Ctrl-Click would create a node and Shift-Click would initiate an edge.

3. *Scrolling in the graph window.* Currently, if the graph is too large to fit in the current window, the only recourse is to do force-directed layout.

4. *Scaling of graph within window.* A possible alternative or option to the previous item is making the node positions adjust so that they always fit into the current window. How best to handle the semantics is not clear: you don't necessarily want the editor to change x and y attributes in the GraphML whenever user resizes the window. It may make sense to establish a grid based on maximum horizontal and vertical dimensions when the graph is first loaded.

5. *Mapping attributes to actions.* In order to make animations more accessible to visually impaired users, there should be a mechanism that, under user control, specifies how Boolean attributes such as marking or highlighting are "displayed". Currently, the thickness of highlighted node borders and edges can be controlled in the Preferences panel. A more sophisticated mapping mechanism that incorporates sound as well as visuals is needed. The ultimate approach would allow mappings for arbitrary attributes defined by user or programmer.

6. *Inflection points on edges.* For animation of automata it's important to have curved edges if there is a transition going from state q to state r and another from r to q; other applications may need this as well. A single inflection point, carefully placed, and present only if there are parallel edges, could accomplish this.

7. *More preferences.* Font sizes for labels and weights, thickness for colors distinct from highlights, distinct highlight colors for Galant types versus functions.

8. *Selection of multiple nodes and/or edges.* It might prove useful to move a collection of nodes instead of just a single node or give a collection of nodes or edges the same color, label or weight. This is mostly for edit mode.