Galant: A Graph Algorithm Animation Tool[*]

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

August 10, 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 6.0.2 of Galant. Source code is available at
https://github.com/mfms-ncsu/galant

[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 [12, 13] and that of Bentley and Kernighan [8] 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 [25] 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 [34]) 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 [11].

Observer-oriented (passive/demonstration) systems. One general purpose animation program is ANIMAL [25, 26]; 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 [16] 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 [24] 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 [5, 6] 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 [23] is an impressive animation system. Like JAVENGA, 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 [9] is an applet that combines exploration, simple programming techniques (using the Snap [29] language) and a simple graph API. Since it stores graphs in `dot` format – see [20] – 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 [15, 27, 28] aims to create a textbook compilation of a variety of visualizations, mostly designed for observers.

Early work. Earlier animation tools/systems include GDR [32], John Stasko's Tango [33], Xtango, and SAMBA,[3] and, of course, the work of Brown and Sedgewick [12, 13] and that of Bentley and Kernighan [8]. 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 in Section 3.4. More detailed documentation is in the Appendices – Appendix A for animation examples, B for user documentation, C for animation programmer documentation, and D for known bugs.

---

[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 [31, 32] 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 [22]) 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 [20] and the huge body of research carried on by the graph drawing community [18], can be used externally as well. Graphs used in Galant animations can be analyzed externally using tools such as Gephi [17].
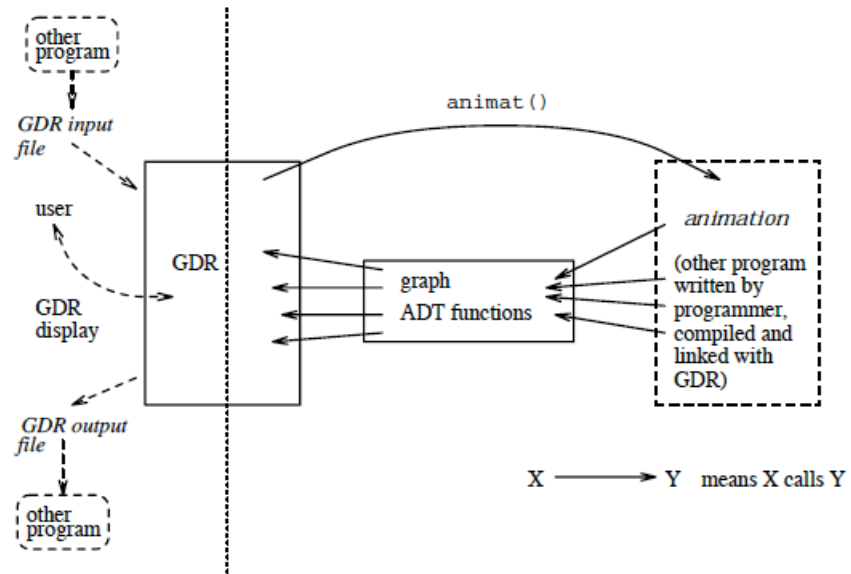
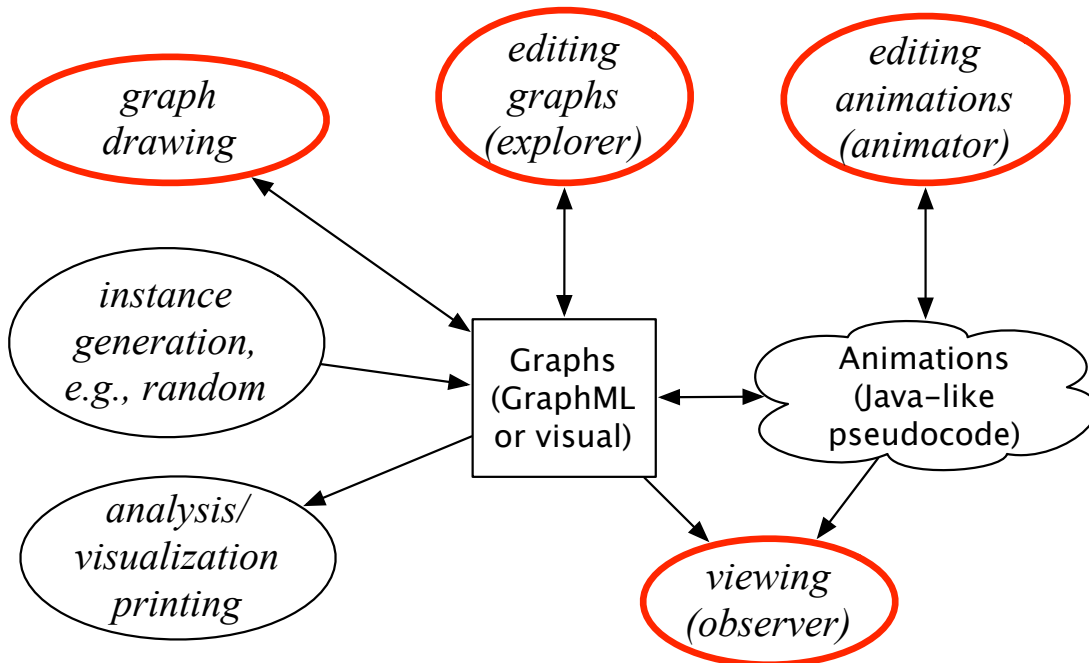Figure 1: GDR design.

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 [19] 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 [20] 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 [21], sgf (used for layered graphs in crossing minimization), and dot (GraphViz [20]).

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. [14], 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 [22] – 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., [30]). 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, 7]). 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. [10].

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] T. BALOUKAS, *Javenga.* users.uom.gr/∼thanasis/JAVENGA.html. 5

[6] ——, *Javenga: Java-based visualization environment for network and graph algorithms*, Computer Applications in Engineering Education, 20 (2012), pp. 255–268. 5

[7] 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

[8] 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

[9] S. BIRD, M. NAZEČIĆ-ANDRLON, AND J. GALLINA, *Edgy.* http://snapapps.github.io/. 5

[10] 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

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

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

[13] M. H. BROWN AND R. SEDGEWICK, *Techniques for algorithm animation*, IEEE Software, 2 (1985), pp. 28–39. 4, 5

[14] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, The MIT Press, third ed., 2009. 8, 13

[15] E. FOUH, M. SUN, AND C. SHAFFER, *OpenDSA: A creative commons active-eBook.* Poster presentation., SIGCSE, 2012. 5

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

[17] GEPHI, *The open GraphViz platform.* gephi.org. 6

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

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

[20] GRAPHVIZ, *Graph Visualization Software.* www.graphviz.org. 5, 6, 8

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

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

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

[24] K. MEHLHORN AND S. NÄHER, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999. 5

[25] 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

[26] G. RÖSSLING, M. SCHÜLER, AND B. FREISLEBEN, *The ANIMAL algorithm animation tool*, in ITCSE 2000, 2002. www.algoanim.info/AnimalAV/. 4

[27] 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

[28] 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

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

[30] 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

[31] 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

[32] ——, *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

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

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

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. [14]. 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
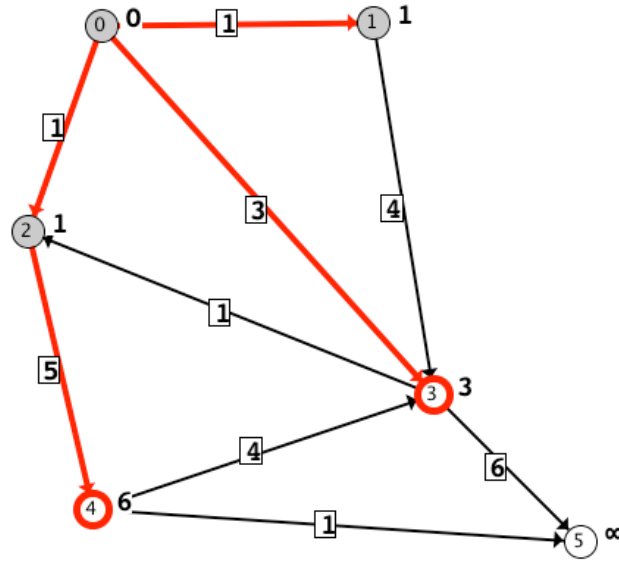
Figure 4: Dijkstra's algorithm on a directed graph.
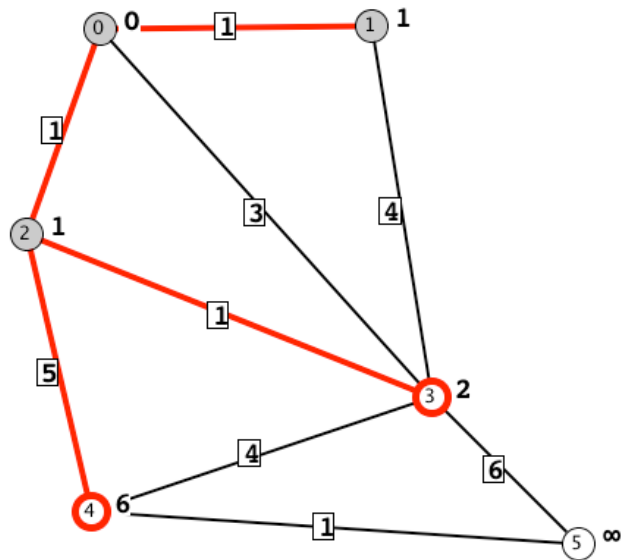
Figure 5: Dijkstra's algorithm on the same graph, undirected.

```
// 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 {
    hideNodeWeights();

    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 += weight(e);
            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.

(a) An edge is added to the tree.

(b) The current edge creates a cycle.
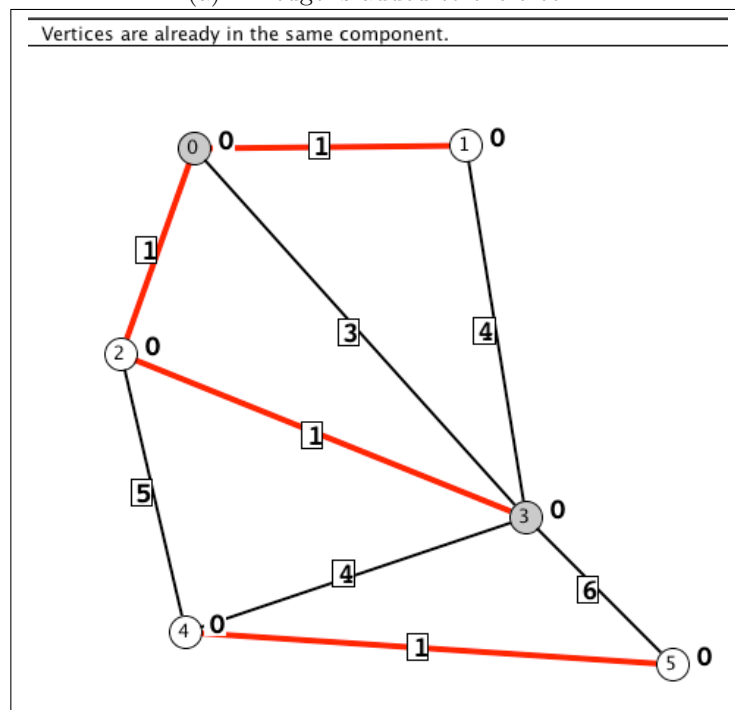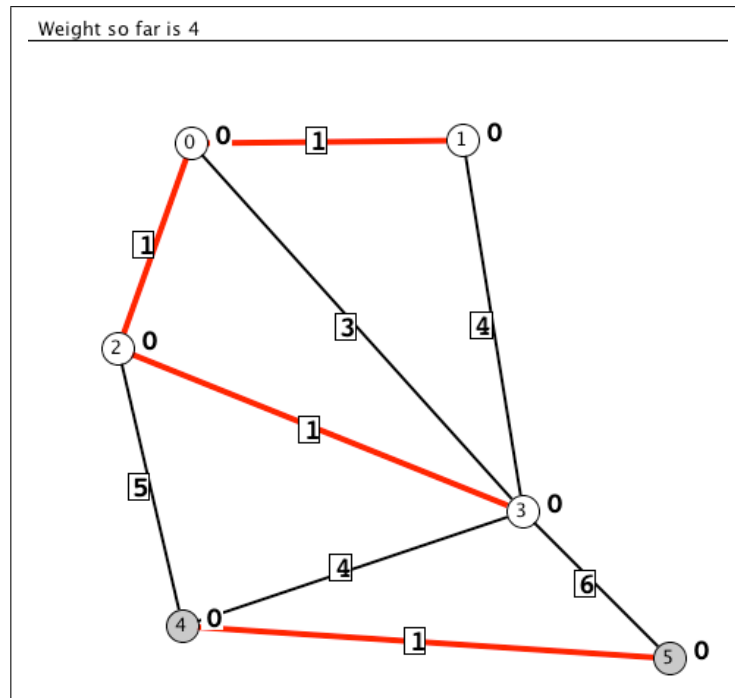
Figure 7: Two steps in Kruskal's algorithm. A message at the top left of the window describes the state of the algorithm.

```
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();
}
```
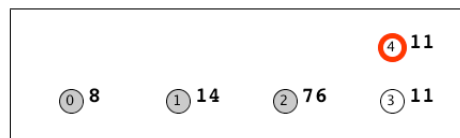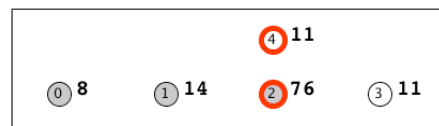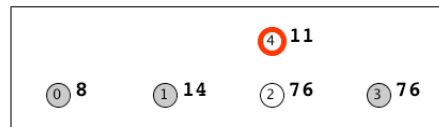

(a) Starting to insert x = A[3].

(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 graphical user interface, primarily for creating, editing, saving and loading graphs and animations and for executing animations.

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 such as Emacs or Notepad++, setting preferences so that files with the .alg extension are treated as if they were Java source files. The primary use of the text editor is to correct minor errors and to see the syntax highlighting related to Galant API functions and macros. 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 only 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:
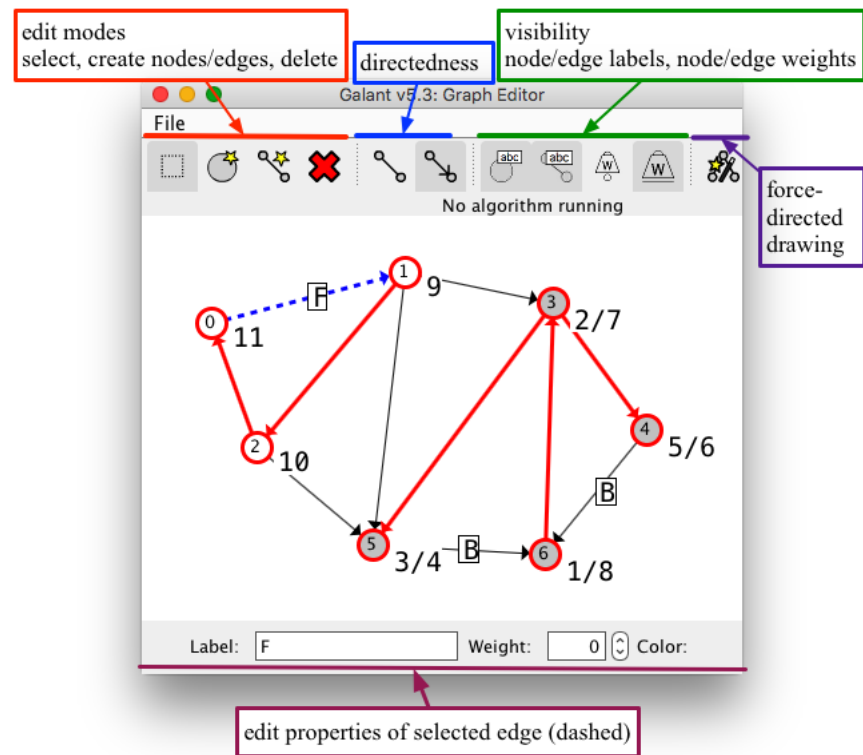
Graph window: add/delete nodes and edges, move nodes, etc.
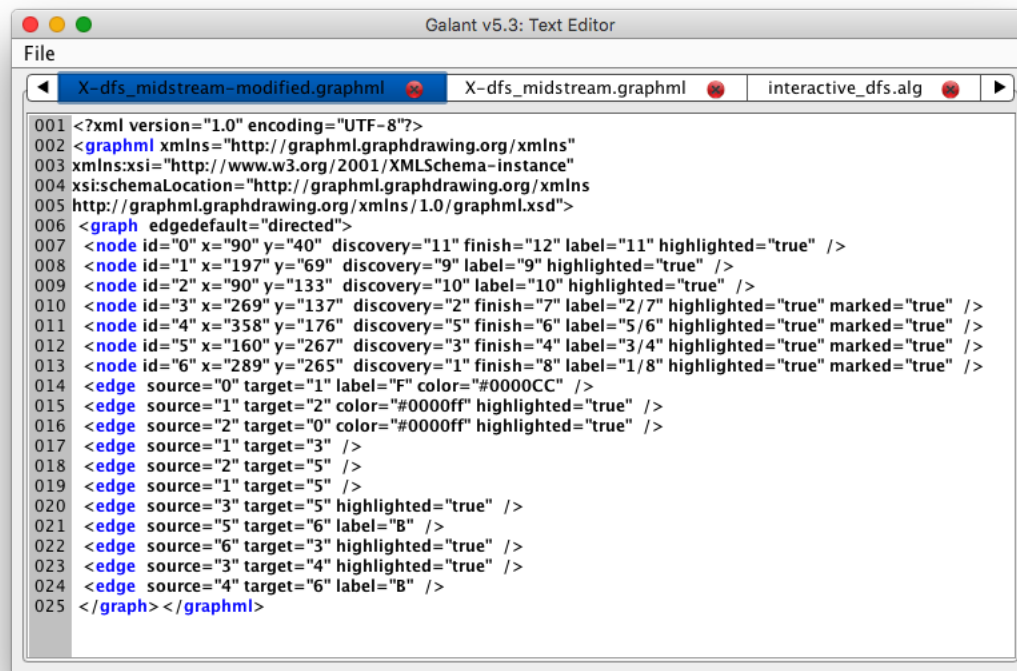
Text window: GraphML representation.

Figure 10: The Galant user interface.

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 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 – so that the lines corresponding to edges are hidden below the filled circles corresponding to nodes).

   - *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. This is the default mode. The square should be outlined in gray unless the user wants to alter the graph in some way. If any other mode is active, e.g., delete, the consequences can be unpleasant (there is currently no "undo" mechanism). A future release will have "modeless" editing.
   - *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 often 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 [22] 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.

Table 1 lists keyboard shortcuts for various editing operations.

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

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
Ctrl-d	toggle graph directedness

Table 1: Keyboard shortcuts for editing operations.

(a) A graph from an external random generator.

(b) Force directed layout applied.

(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 id's do not appear because a node radius of 3 was specified using the Preferences→Graph Display panel.

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 to indicated that it cannot be edited. 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
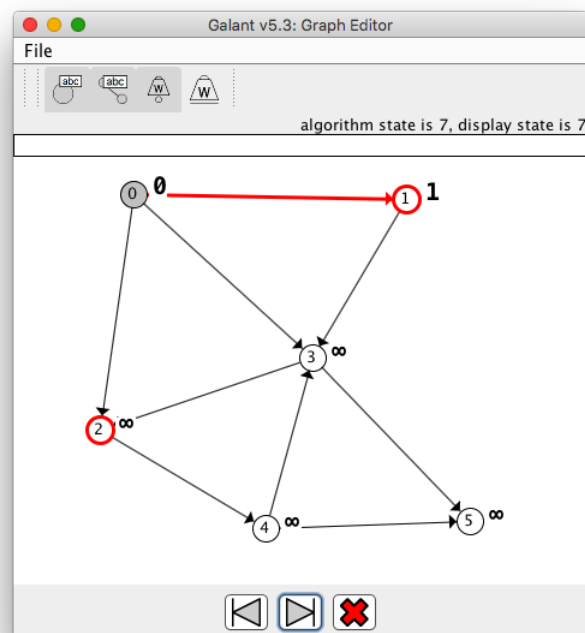
Figure 12: The text and graph windows when an algorithm is running. The text window is at the top – it has buttons for the user to select whether to compile the algorithm, run it (if already compiled) or do both. The graph window in the middle of algorithm execution is at the bottom.

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 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 and node boundary width, both normal and highlighted/colored (Graph Display).

Fig. 13 shows the four preference panels of most interest to the user.
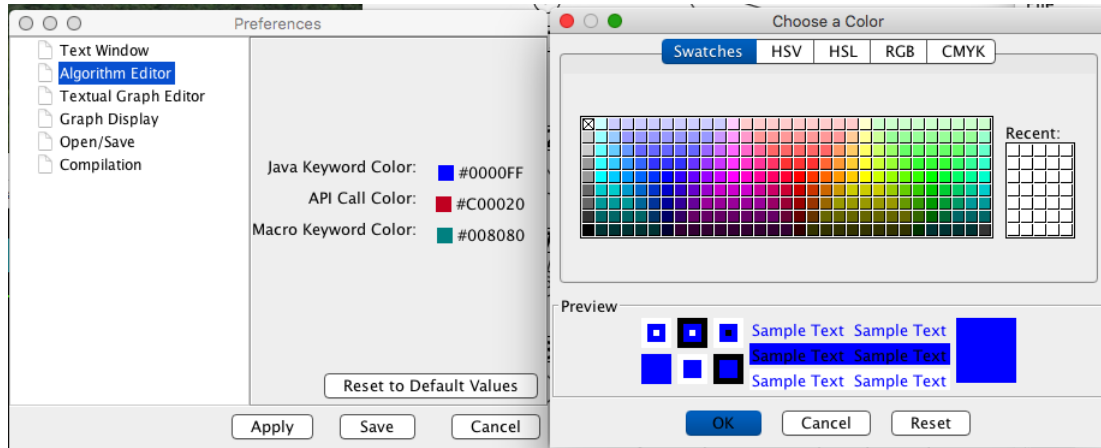
(a) syntax highlight colors – same color chooser as for editing node/edge colors
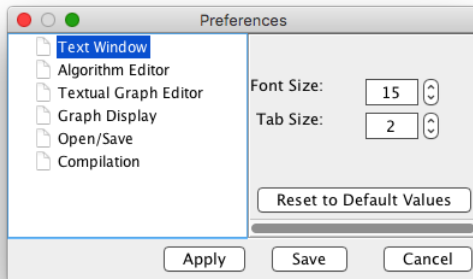
(b) text font size and tab spaces
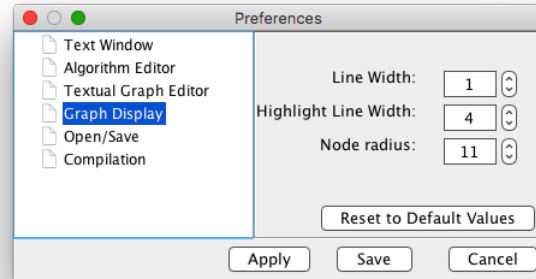
(c) line widths and node radius
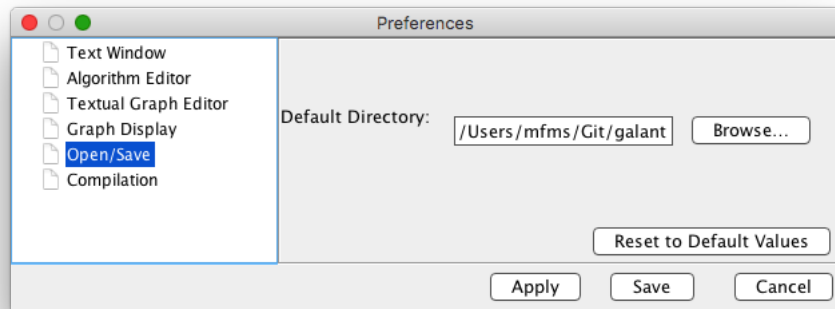
(d) default directory for opening and saving files

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.

Some Java constructs have no equivalent procedural syntax. Most notably *sOne*.equals(*sTwo*), where *sOne* and *sTwo* are strings (of type String) is the only means of equality comparison for strings. Equality of numbers is tested using the == operator. To compare strings lexicographically you have to use *sOne*.compareTo(*sTwo*), which returns a positive number, negative number, or 0 depending on whether *sOne* lexicographically follows, precedes or is equal to *sTwo*, respectively .

Descriptions in this document give the most natural and most common constructs for procedures and functions, usually procedural rather than object-oriented. A perusal of the file Algorithm.java in directory

   src/edu/ncsu/csc/Galant/algorithm

shows the animator the variety of synonyms for these constructs as well as functions not described here. For the interested Java programmer, object-oriented syntax appears as a call in the body of the corresponding function/procedure. For example,

```
public void mark(Node n) throws Terminate, GalantException {
  checkGraphElement(n);
  n.mark();
}
```

shows how the procedural version of mark translates to the corresponding Node method. Here, checkGraphElement(n) throws an exception with a meaningful error message if n is null. The Terminate exception results when the user chooses to exit the animation (not really an exception so much as an alert to the animation program).

We encourage an animator to browse the example animations in subdirectories Algorithm, Test and in the subdirectories of Research.

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 animator should use a program editor such as Emacs or Notepad++ (in Java mode) to edit algorithms offline, not a major inconvenience – it is easy to reload algorithms when they are modified without exiting Galant. The Galant editor is, however, 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. An animation program has the form

> *global variable declarations*
>
> *function definitions*
>
> algorithm {
>   *code block*
> }

Declarations of global variables are also like those of Java:
   type variable_name;

to declare a variable 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. Unlike Java variables, they cannot be initialized in the statement that declares them.[8] 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 $size - 1$. The *type* prefix is omitted if the array has been declared globally. For example, you could have a global declaration

   String [] alpha;

and then, within a function or the algorithm body,

   alpha = new String[10]. The array alpha would then contain 10 null strings (not to be confused with "")[9] indexed from 0 through 9.

Detailed information about function declarations is in Section C.2 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);
    insert(node, pq);
}
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 synonym 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.

Functions and macros for the graph as a whole are shown in Table 2, while Table 3 lists some algorithm functions not related to any aspect of a graph.

Note: *The functions/methods provided by Galant may have multiple synonyms for convenience and backward compatibility. A full list of methods and functions is given in* Algorithm.java *in the subdirectory* src/edu/ncsu/csc/Galant/algorithm.

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,

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 x can be accessed via the function call id(x). 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

---

[8]This restriction applies to global variables only. Variables local to function definitions or to the algorithm can be initialized in-line, just as in Java.

[9] For example, alpha[2].equals("") would cause a null pointer exception but not if you did alpha[2] = "abc" first – then it would simply return false.

NodeList getNodes() NodeSet getNodeSet()	returns a list or set of the nodes of the graph; see Section C.3 for more information about the return types
EdgeList getEdges() EdgeSet getEdgeSet()	returns a list of edges of the graph; return types are analogous to those for nodes
for_nodes(v) { *code block* }	equivalent to for (Node v : getNodes()) { *code block* }; the statements in *code block* are executed for each node v
for_edges(e) { *code block* }	analogous to for_nodes with getEdges() in place of getNodes()
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
edge(Node source, Node target)	returns the edge with the given source and target
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
NodeList neighbors(Node v)[a]	returns a list of nodes adjacent to v
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
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
NodeList visibleNeighbors(Node v) EdgeList visbleEdges(Node v) EdgeList visbleInEdges(Node v) EdgeList visbleOutEdges(Node v)	return lists of neighbors and edges like the corresponding functions (without the visible modifier) defined above; here only the visible nodes/edges are returned – see Table 5
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

[a]The functions edges, inEdges, outEdges, and neighbors have equivalents that return sets: edgeSet, incomingSet, outgoingSet and neighborSet, respectively.

Table 2: Functions and macros that apply to the structure of a graph.

print(Object o1, Object o2, ...)	prints the list of objects as strings on the console; useful for debugging
display(Object o1, Object o2, ...)	writes a message composed of the list of objects as a banner 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 s; useful when wanting user 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; allows algorithm to engage in dialog with the user to obtain values of various types; getDouble is synonymous with getReal
Boolean getBoolean(String message) Boolean getBoolean(String message, String yes, String no)	similar to getString, etc., but differs in that the only user input is a mouse click or the Enter key and the algorithm steps forward immediately after the query is answered; the second variation specifies the text for each of the two buttons – default is "yes" and "no"
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 re-scale the graph

Table 3: Utility functions.

nodes or edges, the id's are not always contiguous.[10] 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 attributes and 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); and label(x,c) instead of setLabel(x,c).

In the subsections below we discuss attributes in more detail, using the categories articulated in the GDR paper – (earlier) citation in the technical report.

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. Macros that hide some Java syntax and extra function calls are provided for that purpose. The macros (which are borrowed from their equivalents in GDR) are:

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

- EdgeList edges(v) returns a list of all edges incident to v, both incoming and outgoing.
- EdgeList outgoingEdges(v) returns a list of edges directed away from v (all incident edges if the graph is undirected).
- EdgeList incomingEdges(v) returns a list of edges directed toward v (all incident edges if the graph is undirected).
- Node otherEnd(e, v) returns the endpoint, other than v, of e.

The following are node 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.

---

[10] The edges in GraphML files are not required to have id's. If they have none, id's are assigned as the file is parsed. These assigned id's are not preserved when the graph (GraphML file) is saved.

Edges. The logical attributes of an edge e are its source and target (destination) accessed using source(e) and target(e), respectively.

Graph Elements. Nodes and edges both have a mechanism for setting (and getting) arbitrary attributes of type Integer, String, and Double. the relevant functions are listed below. Note that the type can be implicit for the setters – the compiler can figure that out, but needs to be explicit for the getters – in Java, two methods that differ only in their return type are indistinguishable. In each case g stands for a graph element (node or edge).

- set(g, String *attribute*, ⟨*type*⟩ *value*), where *type* can be String, Boolean, Integer, or Double.
- set(g, String *attribute*); the attribute is assumed to be Boolean, the value is set to true.
- String getString(g, String *attribute*)
- Boolean getBoolean(g, String *attribute*)
- Boolean is(g, String *attribute*), a synonym for getBoolean
- Integer getInteger(g, String *attribute*)
- Double getDouble(g, String *attribute*)

An object oriented syntax can also be used – this is especially natural in case of is, as in v.is("inTree") – see boruvka.alg in the Algorithms directory.

Arbitrary attributes are useful when an algorithm requires additional 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.

Note: *Attributes retain their types throughout algorithm execution but Galant does not attempt to guess the type of an attribute when reading a graph from a file.* For example, suppose an algorithm does set(v, "pre", 5). To read the value of attribute pre for node v the algorithm will have to use getInteger(v, "pre") (or one of its synonyms) – getDouble(v, "pre") or getString(v, "pre") will return null. However, suppose the user exports the graph to a file mine.graphml after setting the attribute. The GraphML representation for node v will have pre="5", but when Galant reads mine.graphml, the attribute pre, and any other user-defined attribute, will be treated as a string attribute. So getInteger(v, "pre") will return null whereas getString(v, "pre") will return the string "5". This is not a problem unless the user wants to save the state of an algorithm execution midstream *and then use it as a starting point.* The only workaround is for an animation program to do its own parsing, using the utility functions integer and real – see Table 3.

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 functions are int getX(Node v), int getY(Node v) and Point getPosition(Node v) for the getters. The Java type/class Point has fields x and y that can be retrieved using p.x and p.y, where p is a Point. To set a position, use

   setPosition(Node v, Point p)

or

   setPosition(Node v, int x, int y).

Once a node has an established position, it is possible to change one coordinate at a time using setX(Node v, int x) or setY(Node v, int y).

The user is allowed to move nodes during algorithm execution and the resulting positions persist after execution terminates (other attributes do not). Node position is the only attribute that can be "edited" by the user at runtime. For some animations, however, such as sorting, the animation itself needs to move nodes. To avoid potential conflicts between position changes inflicted by the user and those desired by the animation, the function movesNodes(), called at the beginning of an algorithm, will keep the user from moving nodes (mouse actions on the graph panel have no effect).

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 3 – provide a convenient mechanism for treating labels as objects of class Integer or Double, respectively. The second argument of label 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.[11] Thus, conversion between string labels and numbers works both ways.

The display and print functions also take arbitrarily many objects as arguments. So display(g) and print(g), where g is a node or edge will print information about the attributes of g (as a list in square brackets). Often, you only want to display/print the id of a node or the source and target of an edge. The appropriate incantations for display are display(id(g)) and display(string(g)), respectively. The string function is designed specifically for this purpose and applies only to edges. Because these functions allow more than one argument, you can say, for example,

      display("node = ", id(node), " edge = ", edge)

instead of the more awkward

      display("node = " + id(node) + " edge = " + edge)

demanded by Java string syntax.

In order to make the display of weights more attractive, weights that happen to be integers are shown without the decimal point and at most two positions to the right of the decimal point are shown.

Aside from the setters and getters: setWeight(GraphElement g, double wt), Double getWeight(g), label(g, Object o) and String label(g), the programmer can also manipulate and test for the absence of weights/labels using clearWeight(g) and Boolean hasWeight(g), and the corresponding methods for labels. It is also possible to remove an arbitrary attribute a using clear(g, String a), but there is not yet a has function for arbitrary attributes.[12] You have to test whether the getter returns null.

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 (thickness is controlled by user preference), 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 g refers to an object of type GraphElement, a Node or an Edge):

- highlight(g), unhighlight(g) and Boolean highlighted(g)
- correspondingly, select(g), deselect(g), and Boolean selected(g)
- mark(Node v), unmark(Node v) and Boolean marked(Node v)

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 colors explicitly, thus allowing for many different kinds of highlighting. Use the color(g) (getter), color(g, c) (setter) and uncolor(g) functions, where g is a node/edge and c a color string in the RGB format #RRGGBB; for example, the string #0000ff is blue. Galant defines several color constants for convenience – these are listed in Table 4 – so one can say, e.g., color(g,TEAL) instead of color(g,"#009999").

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 one of the native Galant attributes is nonexistent or has a null value – these two are equivalent. When displayed in the graph window, nonexistent labels and

---

[11] Galant functions return objects, Integer or Double, when return values are numbers for this reason.

[12] Semantic details, such as whether to implement hasString(g, a), hasInteger(g, a), etc., and/or to use has(g, a) to check whether the attribute exists in any form need to be worked out.

`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 4: Predefined color constants.

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(g) returns null as expected
- all functions returning Boolean values, such as highlighted(g), marked(Node v) and those for attributes defined by the animator, return false
- label(g) returns an empty string; this ensures that it is always safe to use a label in an expression calling for a string
- weight(g) throws an exception; there is no obvious default weight; a program can test for the presence/absence of a weight using the hasWeight(g) or hasNoWeight(g) methods

Of the attributes listed above, weight, label, color and position can be accessed and modified by the user as well as the program. Except in case of node positions, as noted above, the user can do this only in edit mode. In all cases of display attributes modifications during runtime are ephemeral – the graph returns to its original, pre-execution, state after running the animation. The user can save the mid-execution state of the graph: select the Export option on the file menu of the *graph* window.

A final display attribute is visibility. An algorithm sometimes involves deletion of nodes or edges, or, as is the case with our implementation of Boruvka's algorithm, some edges are no longer important. Graph elements can be hidden and made to reappear using the hide and show functions.

To understand how showing and hiding of nodes and edges works, we need to distinguish between the logical state of a node/edge and what is displayed. Logically, for any graph element g, show(g) makes g visible and hide(g) makes it invisible. For nodes, the logical and display state are the same – a logically visible node will be drawn on the graph window. But an edge will be drawn only if it is logically visible *and both of its endpoints are also logically visible.*

A summary of functions relevant to node and edge attributes (their procedural versions) is given in Table 5. Some of the most important functions, those relevant to the structure of the graph, are listed in Table 2.

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 animation might want to hide node weights initially and reveal them for individual nodes as they become relevant. Or it may want to hide edges individually as it progresses to get the effect of deleting them and then reveal them again at the end. These functionalities can be accomplished

Here, *element* refers to either a Node or an Edge, both as a type and as a formal parameter.

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
string(Edge e)	returns a string of the form "(s,t)", where $s = $ source(e) and $t = $ target(e)
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
NodeList unmarkedNeighbors(Node v)	returns a list of the adjacent nodes that are not 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 6 – 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; if node is hidden, all its incident edges are hidden as well; show(v), where v is a node, will show only the incident edges that are not hidden
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 4 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 5: Functions that query and manipulate attributes of individual nodes and edges.

These functions are designed to access or manipulate attributes for all nodes or edges at once instead of individually. Also included are functions that deal with graph attributes.

hideAllNodeLabels() hideAllEdgeLabels() hideAllNodeWeights() hideAllEdgeWeights()	hide all node/edge labels/weights these functions are typically used to hide information so that it can be revealed subsequently, one node or edge at a time
showAllNodeLabels() showAllEdgeLabels() showAllNodeWeights() showAllEdgeWeights()	make all individual node/edge weights/labels visible 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(), showGraph()	undo any hiding of nodes/edges that has taken place during the algorithm; showNodes() only shows edges currently (logically) visible; showGraph() restores visibility of both nodes and edges
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 6: Functions that query and manipulate graph node and edge attributes globally.

movesNodes()	keeps the *user* from moving nodes during animation execution
setDirected(boolean d)	determine whether the animation will treat the graph as directed (d = true) or undirected (d = false)
showNodeLabels(), showEdgeLabels() showNodeWeights(), showEdgeWeights()	ensure that node/edge weights/labels will be visible throughout animation execution, unless explicitly hidden by the animation, or visibility is changed by the user using the toggle buttons
hideNodeLabels(), hideEdgeLabels() hideNodeWeights(), hideEdgeWeights()	ensure that the specified attributes remain hidden throughout the animation unless visibility is changed using toggle buttons; the attributes are hidden *regardless of whether the animation shows or hides them*

Table 7: Functions that act as declarations.

by hideAllNodeWeights[13] or showEdges, respectively. A summary of these capabilities is given in Table 6.

- An algorithm hides some edges individually and uses showEdges to make all of them visible in one step later.
- An algorithm hides some nodes individually (along with their incident edges); it later makes all nodes visible and selectively makes some of their incident edges visible; nodes that were not hidden are unaffected; the visibility of any edges is not affected either.
- An algorithm hides individual nodes and edges and later restores visibility to the whole graph; the function to use in this case is showGraph.

C.1.5   Functions that behave as declarations

Some Galant functions are independent of animation steps and are intended as declarations at the beginning of the algorithm code. These are listed in Table 7. We already mentioned movesNodes() – it signals that the animation will control node movement and prevents the user from moving nodes.

A second example is setDirected. It has the same function as the toggle buttons at the top of the graph window. In some cases, such as shortest paths algorithms and breadth-first search, it makes sense for the user to decide whether or not the animation should treat the graph as a directed graph. Other animations, such as dfs_d, are specifically designed for directed graphs. In those cases it makes sense for the animation to make the graph directed. When setDirected is called, the toggle buttons (not visible during execution) are set as well, so that the state of the buttons is consistent with that of the graph.

The remaining declarations involve visibility of weights and labels. As with directedness, these functions mimic the actions of the appropriate toggle buttons. The user may also change visibility of labels and weights during execution. The declarations allow the animation to make visible exactly those attributes that are relevant during its execution. For example, the weighted graph algorithms (shortest paths and minimum spanning trees) need the edge weights to be visible; a showEdgeWeights() call at the beginning ensures that this happens regardless of whether the user made edge weights visible during editing (or execution of a previous algorithm).

C.2   Definition of Functions/Methods

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

---

[13] The modifier All distinguishes show/hideAllNode/EdgeLabels/Weights from show/hideNode/EdgeLabels/Weights. The latter are designed to mimic user toggle buttons – see Section C.1.5.

In the table below *EdgeCollection* refers to any preexisting data structure whose elements are edges while *NodeCollection* refers to one whose elements are nodes. Each type of structure can be initialized as empty or as containing elements of a given collection. As with arrays – see page 28 – the new operator creates a new instance of a structure.

data structure	initializers
EdgeList	new EdgeList()
	new EdgeList(*EdgeCollection*)
NodeList	new NodeList()
	new NodeList(*NodeCollection*)
EdgeSet	new EdgeSet()
	new EdgeSet(*EdgeCollection*)
NodeSet	new NodeSet()
	new NodeSet(*NodeCollection*)
EdgeQueue	new EdgeQueue()
	new EdgeQueue(*EdgeCollection*)
NodeQueue	new NodeQueue()
	new NodeQueue(*NodeCollection*)
EdgePriorityQueue	new EdgePriorityQueue()
	new EdgePriorityQueue(*EdgeCollection*)
NodePriorityQueue	new NodePriorityQueue()
	new NodePriorityQueue(*NodeCollection*)

Table 8: Basic Galant data structures and initialization.

```
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 in depth-first search.

```
function visit( Node v ) { code }
```

C.3   Data Structures

Galant provides some standard data structures for nodes and edges. These are described in detail in Tables 8 and 9. All Galant data structures are instances of the Java Collection interface and therefore automatically have the methods required of a collection. For the interested Java programmer, they are also extensions of relevant concrete Java classes and inherit those methods as well. For details, see the Java source code in directory

src/edu/ncsu/csc/Galant/graph/datastructure.

Galant provides procedural versions of the required methods size, isEmpty, add and remove for all structures, i.e., if C is a collection (data structure) and g is a graph element of the appropriate type,

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

type first(*list* L)	returns the first element of L
type top(*list* L)	returns the first element of L (top of stack)
add(*element* g, *list* L)	adds the element g to the end of list L
push(*element*, *list* L)	adds the element g to the front of list L (pushes on stack)
type pop(*list* L)	removes and returns the first element of L
remove(*element*, *list* L)	removes the first occurrence of g from L
size(*list* L)	returns the number of elements in L
empty(*list* L)	returns true if L is empty

NodeQueue and EdgeQueue: queues of nodes or edges, respectively. Same conventions as for lists, except we use *queue* in place of *list*.

void put(*element*, *queue*)	adds the element to the rear of the queue; throws an exception if the element is null
type get(*queue*)	returns and removes the element at the front of the queue; throws an exception if the queue is empty
type front(*queue*)	returns the element at the front of the queue without removing it; throws an exception if the queue is empty

NodeSet and EdgeSet: sets of nodes or edges, respectively; same conventions as for lists, except for the use of *set*. The first two table entries give natural syntax for set membership.

boolean *set*.contains(g)	returns true if g is an element of the set, where g is a node or edge, as appropriate; if g has the wrong type, this method will simply return false
boolean g.in(*set*)	returns true if g is an element of the set, where g is a node or edge, as appropriate; here, if g has the wrong type, the error will be caught by the compiler
union(s_1, s_2)	returns the union of s_1 and s_2; here s_1, s_2 and the return value are all of type NodeSet or EdgeSet
intersection(s_1, s_2)	returns the intersection of s_1 and s_2
difference(s_1, s_2)	returns the set difference $s_1 - s_2$, elements that are in s_1 but not in s_2
symmetricDifference(s_1, s_2)	returns the set symmetric difference of s_1 and s_2, i.e., the union of $s_1 - s_2$ and $s_2 - s_1$
subset(s_1, s_2) or s_1.subset(s_2)	returns true if s_1 is a subset of s_2, false otherwise

Table 9: Operations on Galant data structures (procedural versions).

the functions are size(C), empty(C), add(e, C) and remove(e, C), respectively.

Table 8 shows how each data structure can be initialized in two different ways. One is to create a collection with no elements, the other creates one from another collection that has the same element type. For example, if S is a previously declared and initialized EdgeSet (with elements added initially or later) you can say

   EdgeList L = new EdgeList(S)

This becomes especially useful for priority queues – see Section C.4 and Table 10 below for more details on initializing and using these. To create a priority queue containing all edges you would simply say

   EdgePriorityQueue PQ = new EdgePriorityQueue(getEdges())

Java collections also have iterators. The syntax for accessing all elements of a data structure is

   for (*type g* : *collection*) { *code body* }

where *type* is either Node or Edge, g is a variable, and *collection* is a data structure containing elements of the given type. The code body is executed for each element in the collection, referenced as g. For example, if S is an EdgeSet you could do

```
for ( Edge e : S ) {
    highlight(e);
}
```

to highlight all the edge in S. The order of appearance of the elements depends on the structure. For lists and queues it is the order in which they were added, for sets it is random (the elements are hashed), and for priority queues it is based on the values of the keys.

It is possible to create structures of type EdgeStack and NodeStack but if you want simple procedural syntax and friendlier error reporting we recommend using lists instead.

Table 9 summarizes the operations available on each of the data structures. The semantics of functions add, remove, size, and empty are shown in the table only for lists, but apply to the others as well.

C.4   Sorting, Priority Queues, and Comparators

Priority queues are a special case in that the order of appearance, which affects the semantics of the key functions, is dependent on the *comparator* that is used to determine their total order. Sorting also depends on a comparator, so we discuss sorting and priority queues in the same section. The default comparator for graph elements, nodes and edges, sorts them by increasing weight. If L is a list of nodes or edges

   sort(L)

rearranges the elements of L so that they are in order of increasing weight (if any element has no weight there is a null pointer exception). Galant sort is actually a Galant macro that expands to the Java Collections.sort. A comparator as second argument to sort can alter the default behavior. Galant makes available comparators that specify the attribute (instead of weight) and the ordering (increasing or decreasing) as follows.

- getDoubleComparator(String attribute) returns a comparator based on the values of the given attribute, in increasing order;
  the default comparator is equivalent to getDoubleComparator("weight")
- getDoubleComparator(String attribute, boolean reverse) returns a comparator based on the values of the given attribute, in decreasing order if reverse is true, increasing if it is false;
  the default comparator is equivalent to getDoubleComparator("weight", false).
- getIntegerComparator(String attribute) and getIntegerComparator(String attribute, boolean reverse) behave like getDoubleComparator except that the attribute has integer values;
  for example, getIntegerComparator("id", true) sorts nodes (or edges) by decreasing id.
- getStringComparator(String attribute) and getStringComparator(String attribute, boolean reverse) behave like their numerical counterparts; here the ordering is lexicographic;

Initialization

new EdgePriorityQueue() new NodePriorityQueue()	creates an empty min-heap that uses weight as key
new EdgePriorityQueue(C) new NodePriorityQueue(C)	creates a min-heap that uses weight as key and contains all the elements of C, which can be a list, set, or queue
new EdgePriorityQueue(boolean isMax) new NodePriorityQueue(boolean isMax)	creates an empty priority queue that uses weight as key, if isMax is true, the queue is a max-heap, otherwise it's a min-heap
new EdgePriorityQueue(Comparator C) new NodePriorityQueue(Comparator C)	creates an empty priority queue that uses C to compare elements; if C is a GraphElementComparator, the attribute and whether it's a min or max heap are extracted from the comparator

Functions. Here g is a GraphElement, i.e., an Edge or Node and Q is a priority queue of the appropriate type. The return type *element* is either Edge or Node.

void add(g, Q)	adds g to the priority queue; the priority is defined by the initialization method
void insert(g, Q)	same as add but does error checking
void remove(g, Q)	removes g from Q; used for any element; not efficient because it searches the whole queue
element best(Q)	returns the best element, min or max value as determined by initialization, in Q; runtime is constant
element removeBest(Q)	returns and removes the best element, min or max value as determined by initialization, in Q; runtime is $O(\lg |Q|)$
void changeKey(g, Q	reorganizes Q taking into account a change in value for g of the attribute specified at initialization; *the animation program must execute the actual change beforehand*; runtime is $O(|Q|)$ because this translates to a remove followed by an insert

Methods. These have not (yet) been made available in procedural syntax. Conventions same as for functions

element Q.min() *element* Q.max()	equivalent to *element* best(Q), specific to min/max heap; an exception if heap is the wrong kind, based on initialization
element Q.removeMin() *element* Q.removeMax()	equivalent to *element* removeBest(Q), specific to min/max heap; an exception if heap is the wrong kind, based on initialization
Q.changeKey(g, *val*)	equivalent to set(g, "*a*", *val*), where a is an attribute, followed by changeKey(g, Q); here a must be an attribute with a Double value – otherwise the method has no effect
Q.changeIntegerKey(g, *val*)	same as Q.changeKey(g, *val*) except that a must be an attribute with an Integer value
Q.changeStringKey(g, *val*)	same as Q.changeKey(g, *val*) except that a must be an attribute with a String value
Q.decreaseKey(g, *val*)	same as Q.changeKey(g, *val*); provided because it's standard nomenclature in many algorithms

Table 10: Priority queue initialization and functions in Galant.

for example, getStringComparator("label", true) sorts nodes or edges based on their labels, in decreasing lexicographic order ("zebra" comes before "antelope").

To do the sorting simply call sort with the appropriate comparator as second argument. For example, after

```
EdgeList edges = getEdges();
sort(edges, getStringComparator("label", true));
```

the list edges consists of all edges of the graph, sorted in decreasing lexicographic order by their labels.

Caution: *The type of the attribute must match the type of the comparator.* The incantation getIntegerComparator("label") does not work as expected. Because label is not an integer attribute, the values being compared will all be null and any sort or priority queue operation based on the comparator will result in a null pointer exception.

Table 10 shows the procedural versions of priority queue methods made available by Galant, along with the four ways to initialize priority queues. Also listed are object-oriented methods that yield additional functionality not yet implemented in procedural form.

If the priority queue is initialized from an existing data structure C the only option is for it to be a min-heap (the element that can be accessed and removed efficiently is the one with smallest value) based on weight. An empty queue can be initialized as a max-heap (using weight) or as a heap that uses any valid comparator. A GraphElementComparator is a special case that stores both the attribute used for comparison and whether the heap is to be min or max, obtained using the get*type*Comparator functions described earlier. The additional information allows Galant to report errors if an element to be added has a null value for the given attribute or if the animation program attempts to removeMin from a max-heap or vice-versa.

Priority queue functions are usually as efficient as expected. A notable exception is changeKey. The operation changeKey(g, Q) does remove(g, Q) followed by insert(g, Q) and takes $O(|Q|)$ because of the remove, which needs to search the whole queue.

Unless you use object-oriented syntax you have to change the value of an attribute before calling changeKey. For example, you can do

```
NodePriorityQueue Q = new NodePriorityQueue();
...
Node v = ...;
```

and later

```
setWeight(v, 25);          or    Q.changeKey(v, 25);
changeKey(v, Q);
```

C.5   Queries

An animation program can query the user for various kinds of input. For example, the interactive_dfs algorithm asks the user to give a starting node for a (directed) depth-first search and to give another start node if the search terminates before all nodes are visited. The different query options are listed in Table 3.

A query statement in an animation program initiates an algorithm step unconditionally, i.e., even if it occurs within a beginStep-endStep pair. After the user responds to the query she has to do another step forward before the animation proceeds (except in case of a Boolean query). If the user steps backward after responding to a query, the query is not invoked again. Subsequent forward steps use the same answer. Thus it is not possible to allow a user to explore multiple alternative executions in the same run (such a feature would require major enhancements to the existing implementation).

Queries for nodes and edges ask for node id's (two of them in case of an edge). Galant checks whether an id is that of a valid node and, in case of an edge, whether an edge between the two nodes exists. If the graph is currently directed, the direction of the edge also has to correspond. Any violation

causes an exception to be thrown – a popup window reports the nature of the error and allows the user to choose (a) different id(s). The animator can impose additional restrictions by specifying a set of permissible nodes/edges (unvisited nodes in the case of interactive_dfs). If so, the animator also specifies an error message in case the additional restriction is violated.

Other queries allow an animation to get strings, Boolean values, or numbers from the user. Examples are in the binary_tree and grid algorithms which create complete binary trees or grid graphs based on tree height or grid dimensions, respectively, specified as integers by the user. These queries work the same way as those for vertices and edges. In case of numbers Galant checks whether the input string is a valid integer or floating point number and reports an error otherwise.

Boolean queries are a special case. The user does not type a response. The only options are to press one of two buttons with the mouse or to press the Enter key to specify the default answer (true). The animator can specify the text displayed on the buttons; defaults are "yes" and "no". Another difference with Boolean queries is that the algorithm steps forward immediately when the user responds to the query.[14]

C.6   Exceptions: Compile and Runtime Errors

Errors can occur at compile time, either because a macro is malformed or because the Java code, after macro translation, has errors. The reporting of Java compiler errors is straightforward. They are reported, with line numbers, on the console. The line numbers correspond to those in the Java code listing that also appears on the console (even if there are no errors). In almost all cases the line numbers also correspond to those of the original algorithm (before macro translation) in the text window.[15]

Errors due to malformed macros are not, unfortunately, reported with line numbers. To make matters worse, unbalanced parentheses or braces inside a function definition or one of the for_... macros result in a malformed macro exception. The best strategy is to use a program editor that does automatic indentation.

Runtime errors are also reported with (almost always correct) line numbers. Galant makes every effort to catch errors before they result in, for example, null pointer exceptions in the Galant implementation. Every function with a graph element argument checks that the argument is not null and reports a GalantException if it is. The second or third line in the stack trace refers to the point in the algorithm where the exception occurred. All exceptions, whether those caught as GalantException's with meaningful messages or those caught in the Galant implementation code, result in a stack trace on the console and a popup window. The latter allows to user to choose whether to continue, meaning that the algorithm is terminated and Galant returns to edit mode, or exit from Galant completely. Null pointer exceptions can almost always be tracked down by looking for the first reference to the algorithm in the stack trace, an item of the form

Galant.algorithm.code.compiled.*algorithm_name*.run(*algorithm_name*:*line_number*)

Exceptions can also occur in edit mode: when reading a graph from a file, when specifying a node or edge after a keyboard shortcut, or when giving the weight of a node or edge. A list of the most common Galant exceptions is in Table 11.

---

[14]The reason this is not the case with other queries is that errors may need to be handled before the algorithm can proceed. Synchronization between the query and the algorithm is not straightforward.

[15]The only known exception is a function definition where the parameters are placed on multiple lines.

message	type/source	explanation
programmer message m	runtime	animation program has encountered an **error**(m) call
programmer message m	runtime	user selected a node or edge not in the set specified by a query of the form **getNode**/**getEdge**(p, S, m), where p is the prompt, S the set, and m the error message
Nonexistent node or edge	runtime	a node/edge is null or does not exist in current state
Graph element has no weight	runtime	no weight was given, neither during editing nor earlier during execution
No edge with source v and target w exists	runtime	can occur when user responds to a query for an edge or the algorithm asks for a specific edge using **getEdge**(v, w)
Empty graph	runtime	animation attempts to get a node when there is none
No node with id i exists	runtime	called **getNodeById**(i) when no node with id i exists
Node has been deleted	runtime	called **getNodeById**(i) when node has been deleted
Attempt to removeMin from max heap (or vice versa)	runtime	priority queues can be initialized as either min heaps or max heaps; Galant checks to make sure the correct remove method is used
Attempt to add null node/edge to (priority) queue	runtime	what it says, prevents later problem with null element
Unable to compute center for node	any time	something got messed up with the x and y coordinates (or the layer information in case of a layered graph), e.g., with a **setPosition** call
Something went wrong when processing algorithm block	macro processing	probably some unbalanced parentheses, braces or brackets in the algorithm
Missing *right parenthesis, bracket, brace* in *code* . . .	macro processing	what it says; the *code* shows only the beginning of the code block in which the error occurred
macro name: Curly braces required	macro processing	missing curly braces after a function header
No compiler found, need a JDK	compiler	either no JDK is installed or **JAVA_HOME** is not set up correctly
Invalid tab - use untitled graph or untitled algorithm	text editor	attempt to save a file when there's something wrong with the current tab/panel (should not happen)
No text when invoking GraphMLParser	GraphML parser	attempting to parse empty file
Missing id for node	GraphML parser	no id specified for the node; nodes are required to have id's; they are optional for edges
Bad id	GraphML parser	the id of a node or edge is not an integer
Duplicate id i	GraphML parser	there is more than node/edge with id i
Missing source/target	GraphML parser	an edge has no source/target in its GraphML representation
Bad source/target id	GraphML parser	the source/target specified in the GraphML file is not a legal integer
Source/target node missing	GraphML parser	the integer id of the source/target does not correspond with any node
Bad weight	GraphML parser	the weight of a node/edge is not a valid floating point number
Bad x/y-coordinate	GraphML parser	the x or y coordinate of a node is not a legal integer
Missing/bad layer/positionInLayer	GraphML parser	something is wrong with layer or positionInLayer of a node in a layered graph

Table 11: Galant exceptions.

D Known Bugs and Annoyances

The following are known bugs/problems with Galant. Some of them, marked * have been observed only on a Mac (on my Mac specifically).

Input/output

- If a graph has been saved via a File→Export in the graph window, runtime attributes such as hidden may cause problems when the graph is read and displayed. The issue has been fixed for the particular case of hidden, but not related ones such as hiddenLabel and hiddenWeight. Nonstandard attributes are saved as strings only even if the algorithm treated them as numbers or Booleans.
  The best workaround when an exception occurs during reading of a graphml file is to move the file to another location. Otherwise the exception will occur each time you open Galant.
- * Occasionally there is an exception involving the user interface with a stack entry that says something about GlyphView. This can happen at any time. Just hit Continue when the error window pops up (usually a good policy if you're not sure what the error is about and it appears not to be affecting your session).

Text editing (of programs or graphs)[16]

- There is no "undo" mechanism, either for the text editor or the editor in the graph window.
- * 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.
- * Occasionally Galant simply hangs; it appears that you can exit (quit) from the file menu of the graph window or from the the Galant menu on the task bar. Oddly, when Galant hangs, you can bring it back to life by executing a command in a separate terminal window.
- Some preferences, such as syntax highlight colors, require the user to exit Galant before they take effect.
- * The Open/Save preference that specifies the default directory for files does not persist from one session to the next (unlike all other preferences).
- 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)

- Editing is mode-driven: the effect of a mouse action is determined by which of the four modes (select, create node, create edge, delete) is selected on the toolbar. This has unpleasant consequences if, for example, the user forgets that she is in delete mode.
- 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

---

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

button is pressed. *If you delete nodes or edges when the smart reposition button is highlighted, an "unexpected exception" is likely to occur.*

- 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.
- It is not possible to change a weight from nothing to 0. You need to set it to 1 first and then back to 0.
- 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).

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.
- If user edits the graph and runs an animation without saving the graph file first, the animation runs on the previous version of the graph.
- 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.
- * If you try to manipulate anything in the text window (e.g., click on File) when an algorithm is running, Galant hangs. You must quit Galant completely without saving anything.

E   Feature Requests

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

1. *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.

2. *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.

3. *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.

4. *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.

5. *Mouse-based selection during animation.* Currently, an animation can ask the user for the id of a node or the id's of endpoints of an edge. It would be nice if the user could click on the node or edge instead.

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.