Galant: A Graph Algorithm Animation Tool*

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

July 26, 2016

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 creator 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 two key advantages: (i) the objects manipulated in an animation all have the same type; and (ii) graphs are ubiquitous and therefore provide 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.

1 Background

Algorithm animation has a long history, dating back at least as far as the work of Brown and Sedgewick [**?**, **?**] and that of Bentley and Kernighan [**?**] 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). These pictures are then composed into a sequence that can be 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; and the *animator*, who designs an animation. The latter may also be

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

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

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

referred to as a *developer* if the process of creating animations is integrated with that of implementing the animation system. An explorer is also an observer and an animator is both of the others. Rössling and Freisleben [**?**] 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 [**?**]) posits that, while students acting as observers or explorers of animations may gain some benefit, a student who takes on the role of animator has a much richer learning experience. This hypothesis has become Galant's major motivating factor — that it should simplify the task of the animator as much as possible while enhancing the ability to produce compelling 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) whether the user acts primarily as an observer; 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 used as educational tools see Bridgeman [**?**].

Observer-oriented (passive) systems. One general purpose animation program is ANIMAL [**?**, **?**]; 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 [**?**] 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. In that sense it would also be a tool. It suffers, however, from the fact that the animator must navigate a complex Java-based interface.

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

Libraries. AlgoViz [**?**] 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 [**?**, **?**, **?**] aims to create a textbook compilation of a variety of visualizations, mostly designed for observers.

Early work. Earlier animation tools/systems include GDR [**?**], John Stasko's Tango [**?**], Xtango, and SAMBA,[2] and, of course, the work of Brown and Sedgewick [**?**, **?**] and that of Bentley and Kernighan [**?**]. SAMBA, and to a lesser extent GDR, is especially notable for emphasis on simplifying the creation of animations so that students can easily accomplish them. Both are also tools by our definition. All of these suffer, however, from using old technology, and, except for GDR, they require off-line creation of problem instances and have no graph-algorithm specific implementations or graph creation interfaces to offer.

3 Galant features

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

3.1 GDR: Galant's predecessor

Galant is a successor to GDR [**?**, **?**] 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 animation creator 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

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

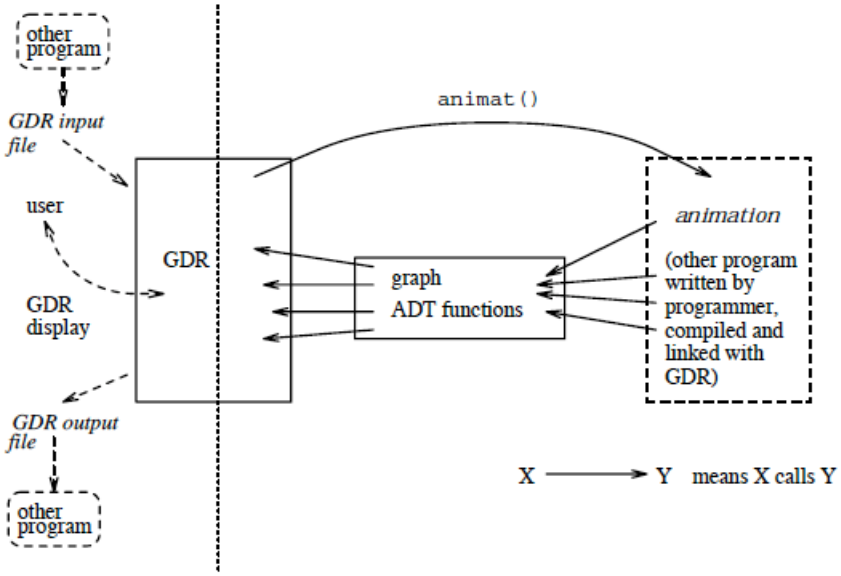


Figure 1: GDR design.

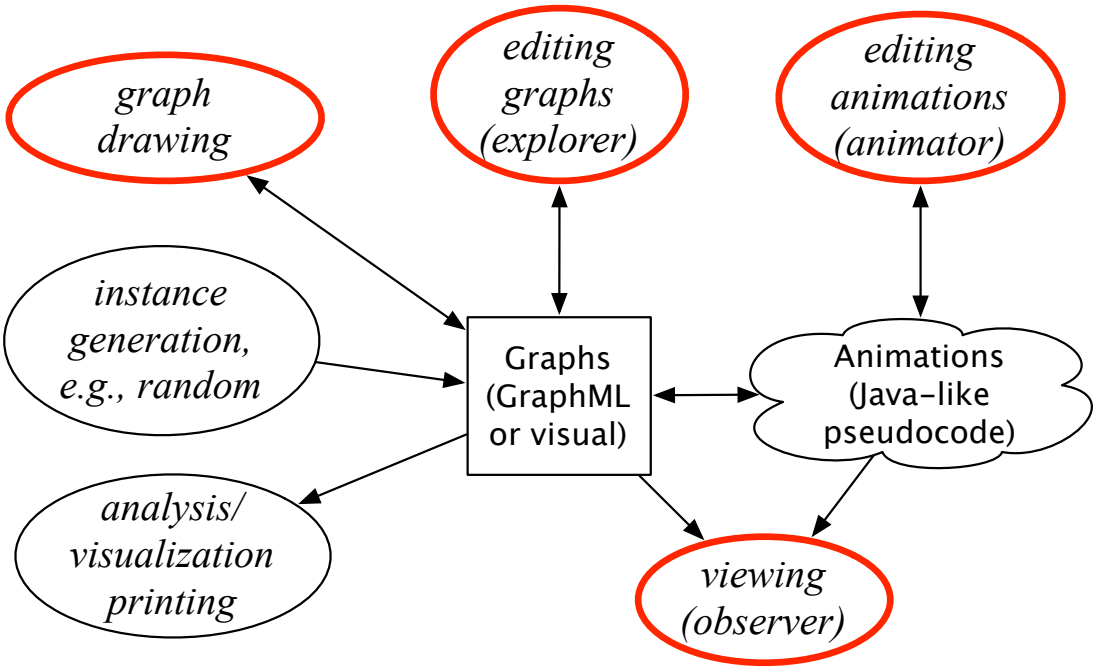


Figure 2: Illustration of Galant design and functionality. Arrows indicate data flow. All functions, shown as ovals can be performed easily outside of Galant. The ones with thick red borders are part of current Galant functionality.

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 (the one reported by Hu [**?**]) 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 [**?**] and the huge body of research carried on by the graph drawing community [**?**], can be used externally as well. Graphs used in Galant animations can be analyzed externally using tools such as Gephi [**?**].

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. 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 [**?**] 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 [**?**] 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. Translators to and from other formats are also available or can easily be constructed. We have written conversion scripts among the following formats: GraphML, gml [**?**], sgf, and dot (GraphViz [**?**]). The sgf (simple graph) format was devised by the first author as a lingua franca specific to layered graphs. It is similar to the `gr` format used in the 9th DIMACS implementation challenge [**?**] except that layer and position are used instead of x- and y-coordinates and there are no weights on the edges. When layered graphs are represented using dot files these are supplemented with *ord* files that give layer and position information for nodes – see Stallmann et al. [**?**].

3.3 For the creator

Here 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 or depth-first search.
- 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. [**?**], attests to the fact that we have succeeded.

A key advantage of the API design, not present in, for example, BALSA, is that it sits directly on top of Java. This allows the creator to develop arbitrarily complex algorithms using other Java class API's and ones devised by the creator. 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) or significant infrastructure for algorithms in a specific domain, as the first author has done for crossing minimization in layered graphs.

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 an algorithm implementation.

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 a force-directed graph drawing method (as described by Hu [**?**]) to adjust node placement.

A creator 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, 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 (i.e., additional Java classes), some of the resulting functionality will no doubt be migrated into its core. Or the core will be enhanced to accommodate new capabilities. Research on crossing minimization heuristics (e.g., [**?**]) has already benefitted greatly from use of Galant to study detailed behvior of these heuristics.

One drawback of the current implementation is that, unlike GDR, it provides no facility for user interaction other than stepping forward and back. For example, one might want the user to be able to select the next node to visit during a depth-first search or a node to insert or remove during an animation of a binary search tree. A query window like that of GDR would be a useful enhancement, although the semantics of repeating a display state during which a query occurs would need to be worked out (GDR has no such problem because it does not allow backward steps).

[Because these 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*[3], spoken navigation and haptic interfaces (see [**?**, **?**, **?**]). 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 [**?**]. 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 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.

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

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 path 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 path 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, select, 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.[4]

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 ids, therefore, need not be contiguous starting at 0, as, in general, they won't 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.

[4] 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. [**?**]. 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 to deal with tree edges that get relabeled as back edges when treated as bidirectional.

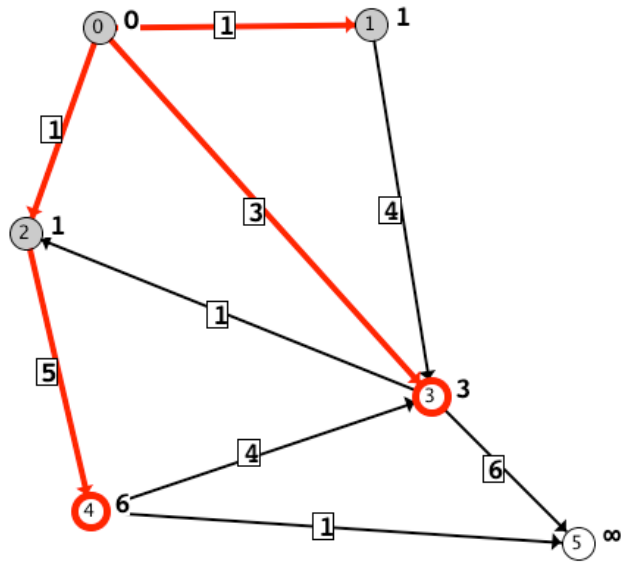


Figure 4: Dijkstra's algorithm on a directed graph.

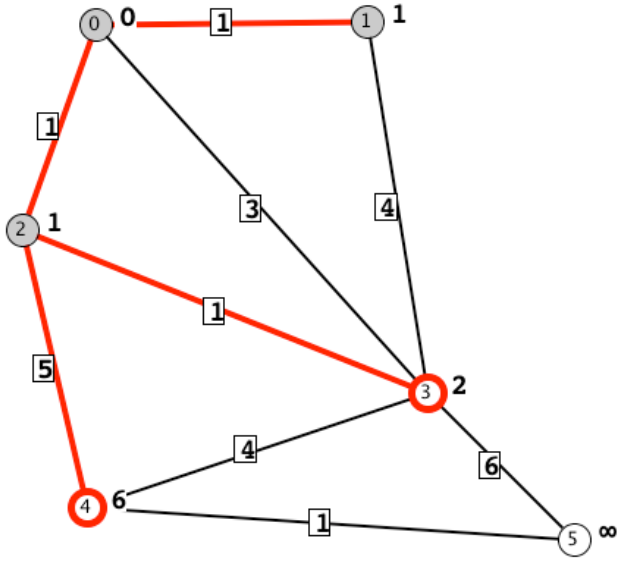
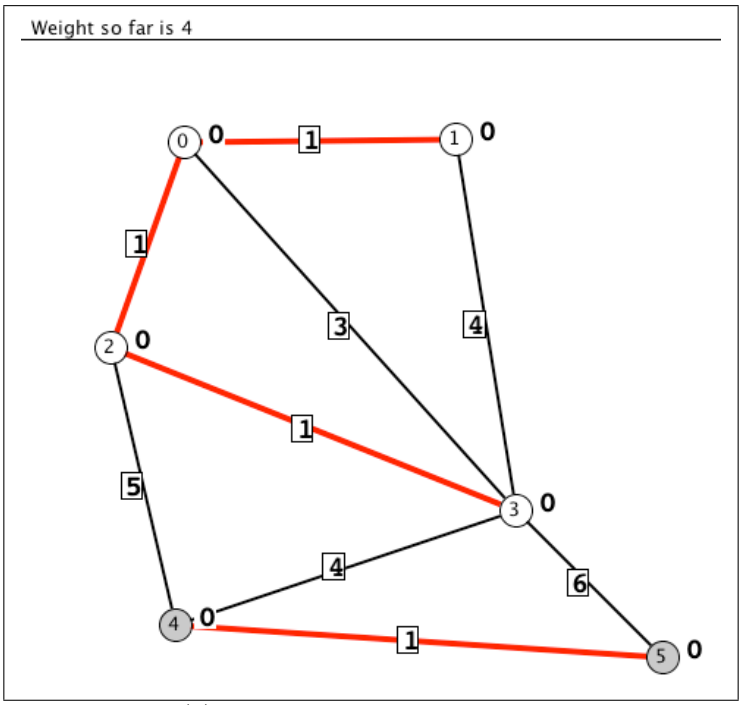


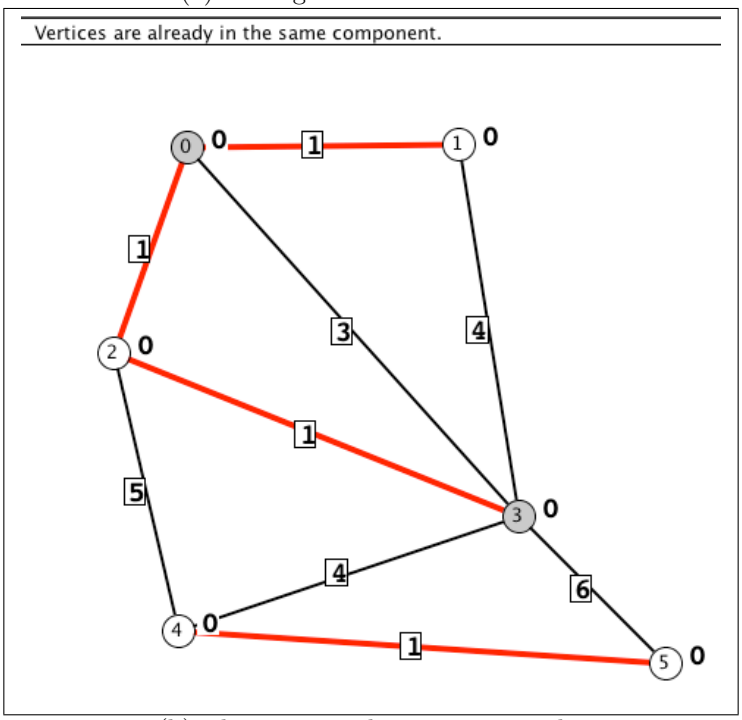
Figure 5: Dijkstra's algorithm on the same graph, undirected.

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

Node [] parent;

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

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

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

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

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

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

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

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

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

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

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



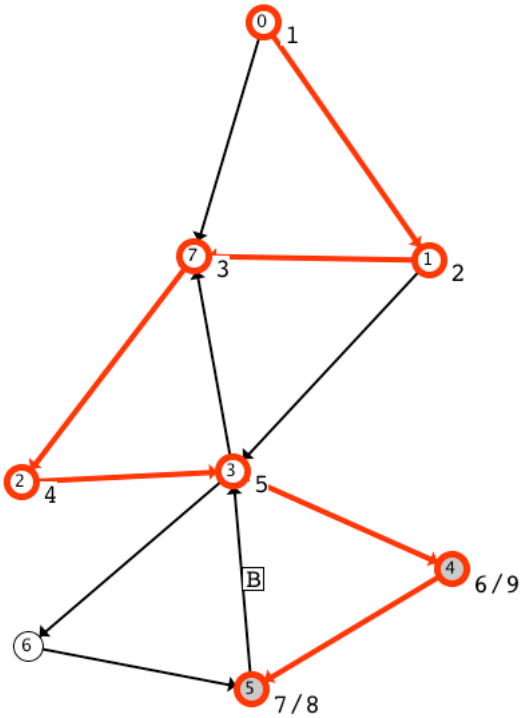
(a) An edge is added to the tree.



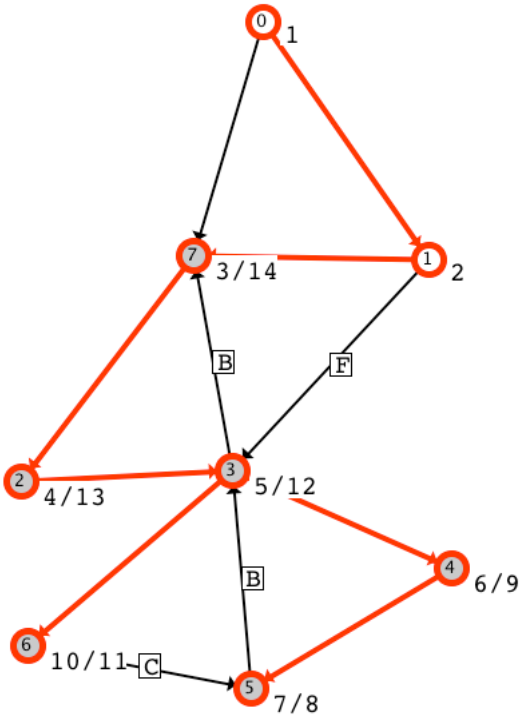
(b) The current edge creates a cycle.

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();
    setLabel(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 ) {
            setLabel(e, "B");
        }
        else if ( finish[id(w)]
                  > discovery[id(v)] ) {
            setLabel(e, "F");
        }
        else {
            setLabel(e, "C");
        }
        endStep();
    }
    time = time + 1;
    finish[id(v)] = time;
    beginStep();
    mark(v);
    setLabel(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 );
        }
    }
}
```



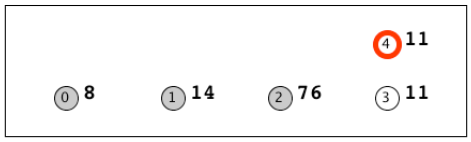
After first non-tree edge is labeled.



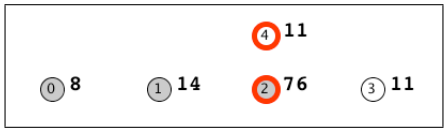
After all but one non-tree edges have been labeled.

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

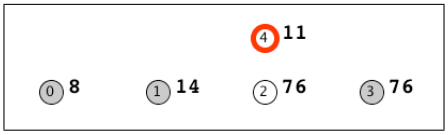
```
for ( int i = 1; i < n; i++ ) {
    double x = A[i]; toInsert.setWeight( A[i] );
    toInsert.setX( xCoord[i] );
    endStep();
    int j = i - 1;
    while ( j >= 0 && A[j] > x ) {
        beginStep();
        toInsert.setX( xCoord[j] );
        nodes[j].setSelected( true );
        endStep(); beginStep();
        A[j+1] = A[j]; nodes[j+1].setWeight( A[j] );
        nodes[j].unMark();
        nodes[j].setSelected( false );
        nodes[j+1].mark();
        endStep();
        j = j - 1;
    }
    beginStep();
    A[j+1] = x; nodes[j+1].setWeight( x );
    nodes[j+1].mark();
}
```



(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.

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 (selected).

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.

B User documentation

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

B.1 Overview

Galant provides three major components across two windows:

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

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

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

B.2 Workspace

Opened graph and algorithm files are displayed in the text window, which has tabs that allow the user to switch among different algorithms/graphs. New algorithms can be created using the "page" icon at the top right of the window and new graphs 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 preferences. Algorithm files have the extension `.alg` and graph files the extension `.graphml`.

B.3 Graph editing

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

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

The graph window, as illustrated in Fig. 10 has a toolbar with four sections:

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

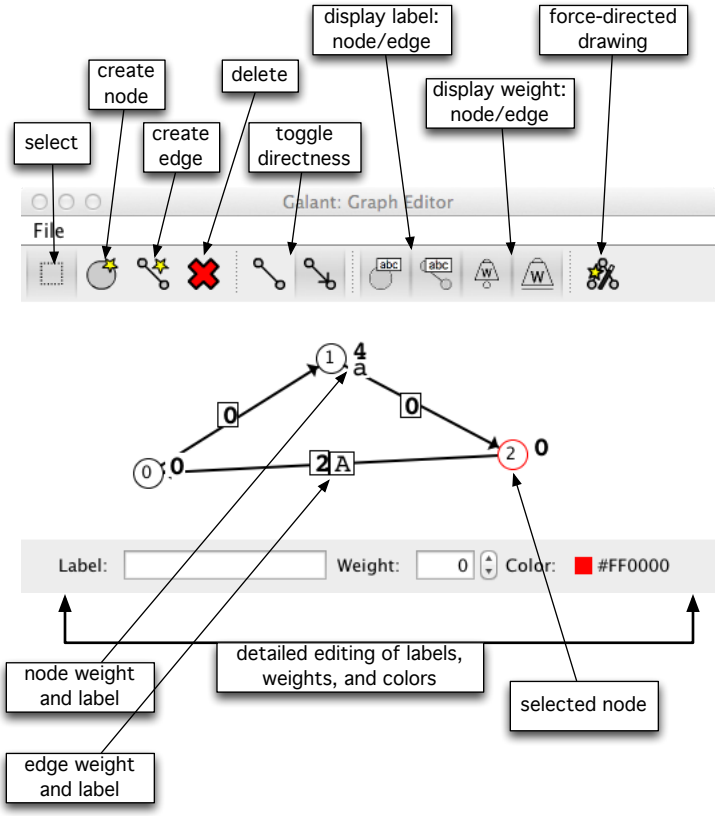


Figure 10: The Galant graph window with annotations.

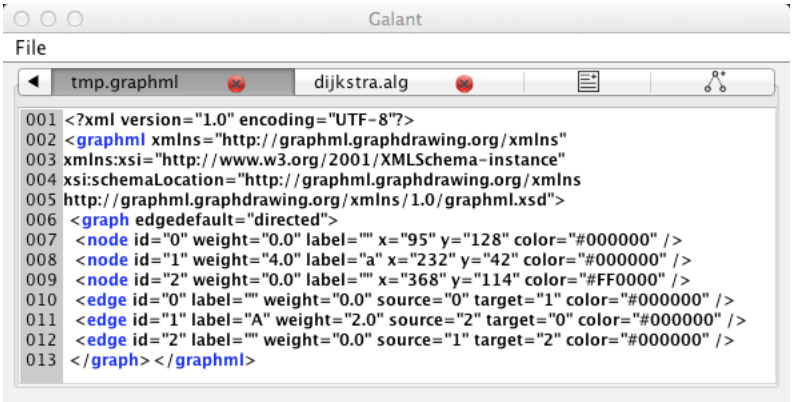


Figure 11: The text window with the GraphML representation of the graph in Fig. 10.

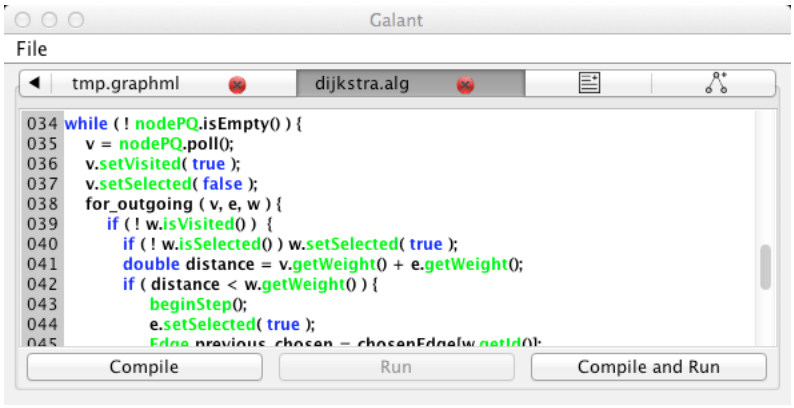


Figure 12: The text window showing Dijkstra's algorithm.

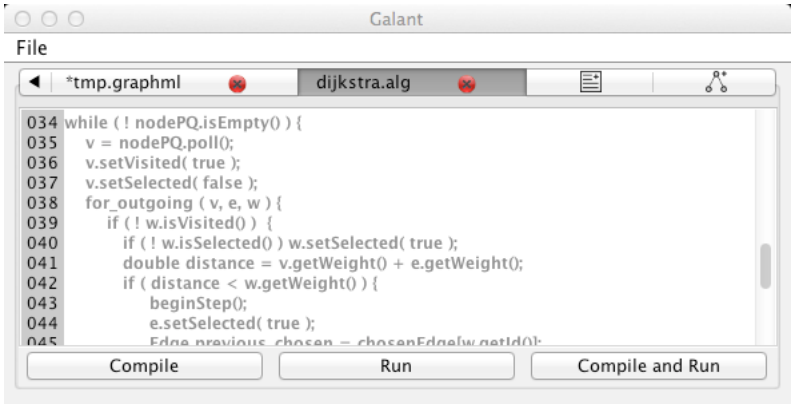


Figure 13: The text window when Dijkstra's algorithm is running.

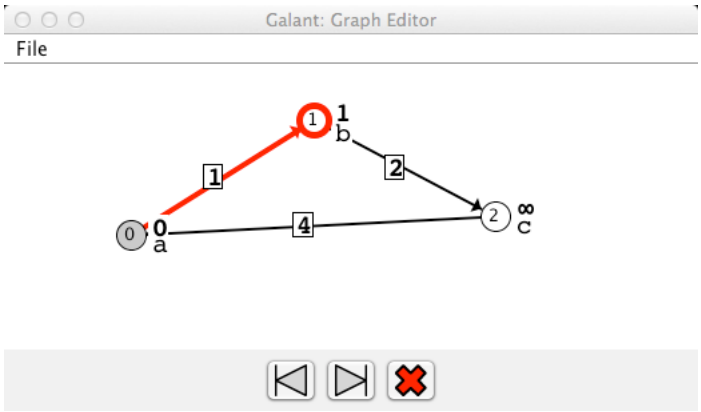


Figure 14: The graph window when Dijkstra's algorithm is running.

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

2. **Directedness toggles** – These change both the interpretation and the display of the graph between directed and undirected. Pressing the undirected (line between two dots) button causes all edges to be interpreted as undirected: this means that, when the code calls for all incoming/outgoing edges, all incident edges are used. Undirected edges are displayed as simple lines.

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

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

4. **Force directed drawing button** – Applies Hu's force directed algorithm [**?**] to the graph. *Caution: this operation cannot (currently) be undone.*

Keyboard shortcuts for graph editing operations are as follows:

ctrl-n – create a new node in a random position
ctrl-e – create a new edge; user is prompted for id's of the nodes to be connected
ctrl-i – do a smart repositioning 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

B.4 Algorithm editing

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

B.5 Animating algorithms

To animate an algorithm the code for it must be compiled and then run via the algorithm controls – the bottom tabs on the text window shown in Figs. 12 and 13. The algorithm runs on the *active graph*, the one currently displayed on the graph window. If there are errors in compilation these will show up on the console (terminal from which Galant was run) and in a dialog box that allows the user to ask for more details and decide whether to exit Galant or not. The console also displays the what the code looks like after macro replacement in case obscure errors were the result of unexpected macro expansion. Line numbers in the macro-expanded code match those of the original so that all errors reported by the Java compiler will refer to the correct line number in the original Galant code. Runtime errors also open the above mentioned dialog box.

When the user initiates execution of an animation by pushing the Run button the animation program steps forward until displays the next animation event or, if a `beginStep()` call has marked the start of a sequence of events, until it reaches the next `endStep()` call. It then pauses execution and waits for the user to decide whether to step forward, step backward, or exit. A step forward resumes execution while a step backward returns the display to a previous state. The algorithm resumes execution 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. 14) or the right/left arrow keys. During the execution of the animation, all graph editing functions are disabled. These are re-enabled when the user exits the animation by pressing the red **X** button or the esc (escape) key on the terminal.

B.6 Preferences

Galant preferences can be accessed via the `File->Preferences` menu item or by using the keyboard shortcut ctrl-P (cmd-P for Mac). Preferences that can be edited are:

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

C Programmer guide

Animation programmers can write algorithms in notation that resembles textbook pseudocode. 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)`.

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

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

Without the `beginStep`/`endStep` override, this initialization would require the observer to click through multiple steps (one for each node) before getting to the interesting part of the animation.

Functions and macros for the graph as a whole are shown in Table 1. Also included are a few functions that are global to an algorithm without any direct connection to a graph.

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

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

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

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

is immune to array out of bounds errors.

C.1 Node and edge methods

Nodes and edges have 'getters' and 'setters' for a variety of attributes, i.e.,
`set`a`(`$\langle a's$ `type`$\rangle$ `x)`
and
$\langle a's$ type$\rangle$ `get`a`()`, where a is the name of an attribute such as `Color`, `Label` or `Weight`. A more convenient way to access these standard attributes omits the prefix `get` and uses procedural syntax: `color(`x`)` is a synomym 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`)`.

print(String s)	prints s on the console; useful for debugging
display(String s)	writes the string s at the top of the window
String getMessage()	returns the message currently displayed on the message banner
error(String s)	prints s on the console with a stack trace; also displays s in popup window with an option to view the stack trace; the algorithm terminates and the user can choose whether to terminate Galant entirely or continue interacting
List⟨Node⟩ getNodes() NodeList nodes()	returns a list of the nodes of the graph; type NodeList is built into Galant and equivalent to the Java List⟨Node⟩
List⟨Edge⟩ getEdges() EdgeList edges()	returns a list of edges of the graph; return type EdgeList is analogous to NodeList
for_nodes(v) { *statement; ...* }	equivalent to for (Node v : getNodes()) { *statement; ...* }; the statements are executed for each node v
for_edges(e) { *statement; ...* }	analogous to for_nodes
Integer numberOfNodes()	returns the number of nodes
Integer numberOfEdges()	returns the number of edges
getStartNode()	returns the first node in the list of nodes, typically the one with smallest id; used by algorithms that require a start node
isDirected()	returns true if the graph is directed
setDirected(boolean directed)	makes the graph directed or undirected depending on whether `directed` is true or false, respectively
Node addNode() Node addNode(Integer x, Integer y)	returns a new node and adds it to the list of nodes; the id is the smallest integer not currently in use as an id; attributes such as weight, label and position are absent and must be set explicitly by appropriate method calls; the second version sets the position of the node at (x,y)
addEdge(Node source, Node target) addEdge(int sourceId, int targetId)	adds an edge from the source to the target (source and target are interchangeable when graph is undirected); the second variation specifies id's of the nodes to be connected; as in the case of adding a node, the edge is added to the list of edges and its weight and label are absent
deleteNode(Node v)	removes node v and its incident edges from the graph
deleteEdge(Edge e)	removes edge e from the graph

Table 1: Functions and macros that apply to the whole graph or to an algorithm more generally.

Logical attributes: functions and macros

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

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

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

- `List<Edge> getIncidentEdges()` returns a list of all edges incident to this node, both incoming and outgoing.

- `List<Edge> getOutgoingEdges()` returns a list of edges directed away from this node (all incident edges if the graph is undirected).
- `List<Edge> getIncomingEdges()` returns a list of edges directed toward this node (all incident edges if the graph is undirected).
- `Node travel(Edge e)` returns the other endpoint of `e`.

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

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

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

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

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

Geometric attributes

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

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

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. Aside from the setters and getters: `setWeight(double)`, `Double getWeight()`, `setLabel(String)` and `String getLabel()`, the programmer can also manipulate and test for the absence of weights/labels using `clearWeight()` and `boolean hasWeight()`, and the corresponding methods for labels. The procedural variants

in this case are `setWeight(Node,double)`, `Double weight(Node)`,[5] `label(Node,String)`,[6] and `String label(Node)`

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

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

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

Although the specific colors for displaying selected nodes or edges are predetermined, the animation implementation can modify the color of a node boundary or an edge, thus allowing for many different kinds of highlighting. The `setColor` and `getColor` methods use String arguments using the RGB format #RRGGBB; for example, the string `#0000ff` is blue. There are several predefined color constants:

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

Of the attributes listed above, weight, label, color and position can be accessed and modified by the user as well as the program. In all cases, modifications by execution of the animation are ephemeral – the graph returns to its original state after execution.

C.2 Additional programmer information

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

Definition of Functions/Methods

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

`function` *[return_type] name* `(` *parameters* `) {`
    code_block
}

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

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

is equivalent to

[5] The *get* is omitted here for more natural syntax.
[6] A natural syntax that resembles English. However, `setLabel(Node,String)` is also allowed.

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

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

Data Structures

Galant provides some standard data strutures for nodes and edges automatically. Each of these is supplied as a single instance.

- `nodeQ` – a queue of elements of type `Node`; provides the the methods of a Java Queue:
  - `offer(n)` – puts node `n` on the queue
  - `poll()` – returns and removes the node at the front of the queue
  - `peek()` – returns the node at the front of the queue but does not remove it
  - `isEmpty()` – returns `true` if the queue is empty
- `edgeQ` – a queue of elements of type `Edge`; methods are the same as those for `nodeQ`
- `nodeStack` – a stack of elements of type `Node`; provides methods of a Java Stack: `push(Node n)`, `pop()`, `peek()`, and `empty()`
- `edgeStack` – a stack of elements of type `Edge`; same as `nodeStack`
- `nodePQ` – a priority queue of elements of type `Node`; provides standard priority queue methods:
  - `offer(n)` – puts node `n` on the priority queue; the priority is defined to be the weight of `n`
  - `poll()` – returns and removes the node with the lowest priority
  - `peek()` – returns the node with the lowest priority but does not remove it
  - `isEmpty()` – returns `true` if the priority queue is empty

  Unfortunately, the only way to implement the decreaseKey operation is to remove and reinsert the node.
- `edgePQ` – analogous to `nodePQ`

Galant also provides classes corresponding to the above instances; these are

- `NodeQueue`
- `EdgeQueue`
- `NodeStack`
- `EdgeStack`
- `NodePriorityQueue`
- `EdgePriorityQueue`

Global Variables

One awkward feature of Galant's implementation is that global variables must be declared `final`. For arrays and other structures this is not a problem, except for the annoyance of the syntax. So, for example,

```
final int [] theArray = new int[ graph.getNodes.size() ];
```

will work as expected: the entries of `theArray` can be modified as the animation progresses. Not so with scalars. The workaround is along the lines of

```
class GlobalVariables {
    public int myInt;
    public double myDouble;
```

```
}
final GlobalVariables globals = new GlobalVariables();
```

and then the globals need to be referred to as `globals.myInt` and `globals.myDouble`, respectively.

D Known Bugs and Annoyances

Input/output

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

Text editing (of programs or graphs)[7]

- 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 – although the tabs bar can be "scrolled", this may be impossible if there are other intervening graphs and algorithms.
- If you attempt to do anything in the text window (File menu or tabs) while an algorithm is running, Galant hangs; it appears that you can quit it from the file menu of the graph window, however.

Graph editing (in graph panel or via keyboard shortcuts)

- 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.
- A change in color of an edge does not appear to take place while the edge is selected for editing – its color stays blue during that time. Only when focus is no longer on the edge does it change color.
- If a node is selected for editing (other than immediately on creation) its weight is set to 0 when the spinner shows up. Initially a node has no weight at all; this should continue to be the case unless the user specifies a weight.
- If user attempts to change weight/label of a node/edge and clicks on the graph panel in the middle of the operation, the change is lost. There is no "ok" prompt as with color.
- When user creates a new edge via keyboard shortcut, there is no obvious way to enter the weight and label.

Compilation and execution

- The macro preprocessor is oblivious to comments. Thus, the placement or contents of a comment may cause it to throw an exception. Such exceptions have been known to occur when comments appear inside functions or contain constructions that are subject to macro expansion. For example, sometimes the word `for` in a comment causes problems.
- Every once in a while an exception occurs when an error-free algorithm is executed or when Galant intitally fires up, but it is possible to step through the animation normally after hitting the Continue button.
- The `function` construct appears not to work if the arguments or return value are arrays (or lists?). Macro expansion for this construct is complex.
- 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.
- There is no way for the user to change anything about the display while stepping through the animation, such as, for example, changing visibility of labels and weights, or positions of nodes to make important features more visible.

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

- There is no way for the animation program to specify that labels and/o weights should be displayed or hidden.
- 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 the user must click on the graph panel before starting the animation; for some reason, there is no reaction to the shortcuts if the first mouse click is on one of the arrows at the bottom of the graph window.

E Planned Enhancements

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

- A programmer will be able to ask the user to select a node or edge via a text query. The appropriate incantations will take the form
  - `getNode(`*`message`*`)`, where *message* is a prompt (string); a popup window asks the user for the id of a node; the function will return the actual node; there is an error dialog if no node with that id exists.
  - `getEdge(`*`message`*`)` – same as `getNode` except that the user is prompted for two node id's; an error dialog arises if either id is non-existent or the edge is non-existent.
- New types/classes `NodeSet` and `EdgeSet` will be added. Both will be concrete classes and will have conversion constructors from `NodeList` and `EdgeList`, respectively. Eventually lists, queues, stacks and priority queues of nodes and edges will all be concrete and all graph, node and edge methods that return containers will return one of these. For example, `getNodes()` will return a `NodeList` instead of a `List<Node>`.

  When combined with the previous enhancement we can add functions such as

  `getEdge(`*`prompt, set, message`*`)`,

  where *prompt* is as before, *set* is a set from which the edge must be selected (e.g., the set of edges incident on a node) and *message* is an error message to be given if the edge exists but is not a member of the given set.