1 Galant Programmer Documentation

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.

1.1 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)`,[1] `label(Node,String)`,[2] 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.

[1] The *get* is omitted here for more natural syntax.

[2] A natural syntax that resembles English. However, `setLabel(Node,String)` is also allowed.

1.2 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

```
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.

This is an external document. It appears both as an appendix in the technical report and as a separate document.