The Eiffel Graph Library

Jimmy J. Johnson

August 2021

Table of Contents

1. INTRODUCTION

Graph theory is useful and indispensable for solving problems in many diverse areas. Majeed and Rauf survey and cite numerous uses of graph theory from the Koinsber Bridge problem in 1735 to modern-day applications including social networks modelling, big data analysis, natural language processing, pattern recognition applications, quantum cryptograph, optimization problems, and others in various disciplines, including chemistry, biology, physics, and computer science {Majeed and Rauf, 2020, #51068}. The application of graph theory to these various areas requires a computer model to set up relationships between the entities under study.

The Eiffel Graph Library (TEGL) simplifies this modeling by providing ready-made containers, relieving the programmer of the burden of building low-level computer data structures, such as matrix, edge-list, or adjacency-lists representations, allowing the programmer to focus on the problem domain. TEGL's model uses standard graph terminology which is fairly intuitive, and its interface is straight-forward.

The next subsection defines terms; the following subsection provides a quick-start example of the interface.

1.1. DEFINITIONS

Intuitively, a **graph** is a set of **nodes** and a set of **edges**, where each edge joins one node to another node. An edge can be traversed from one node to the other node. A **weighted edge** contains a numeric cost of traversing that edge. A **labeled edge** is marked with a data **label**. A **valued node** contains a data **value**. A **weighted graph**, **labeled graph**, or **valued graph** contains the corresponding node or edge types. The definitions combine to from more complex graph definitions, such as a **valued_weighted_graph**, a graph which has values at each node weights on each edge.

A **directed edge** can only be traversed in one direction. A **cyclic edge** connects a node to itself.

A **walk**[1], is a sequence of one or more edges from one node leading to another node. The length of a path is the number of edges comprising the path. The cost of traversing a path is the sum of the costs of traversing each edge in the path. A cycle is a path that returns to the node from which it started.

These basic definitions give enough information for the next section to show a cursory overview and use of some of the top level interface classes in TEGL.

1.2. QUICK-START EXAMPLE

The Eiffel Graph Library contains class *VALUED_ GRAPH [V]* which models a *GRAPH* container holding nodes, where each *NODE* holds a value of type *V*. The graphs in Figure 1 serve as a quck-start example. The example shows two graphs which contain both directed and undirected edges.

[1] TEGL uses the term "walk" instead of "path" to prevent a conflict with the class PATH which represents a way to identify files or directories.

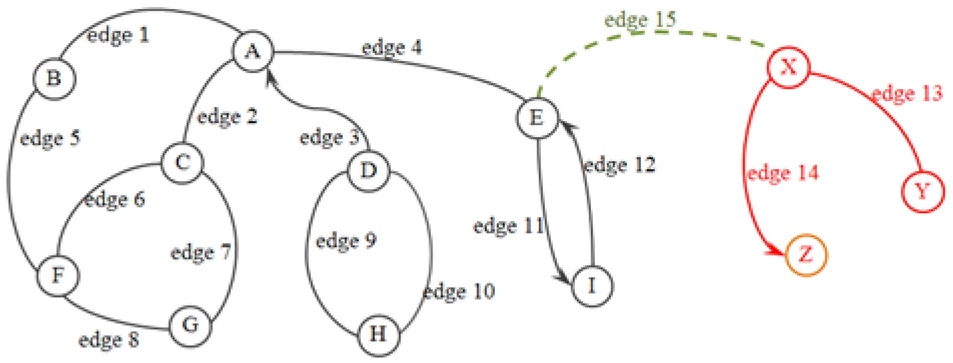

Figure 1 -- A Simple Graph Example

Figure 2 lists the attributes used in this example. The declarations show that each *GRAPH* is a container of nodes containing values of type *STRING*. To avoid CAT Calls, the graph and node declarations should be matching types (i.e. *VALUED_GRAPH* and *VALUED_NODE*) and have same type generic parameter.

```
graph_1: VALUED_GRAPH [STRING]
          -- Graph to hold nodes A through I

graph_2: VALUED_GRAPH [STRING]
          -- Graph to hold nodes X, Y, and Z

node_a: VALUED_NODE [STRING]
          -- Keep a reference to a node for use later

node_e: VALUED_NODE [STRING]
          -- Keep a reference to a node for use later
```

Figure 2 – Attribute declarations

Figure 3 depicts the code to create the two graph attributes and to set up the two graphs.

```
default_create
          -- Initialize the attributes.
     do
          create graph_1
          create graph_2
          create node_a ("A")
          create node_e ("E")
               -- Compare the string values not references to the strings
          graph_1.compare_objects
          graph_2.compare_objects
               -- Make graphs follow a total order
          graph_1.set_ordered
          graph_2.set_ordered
     end
```

Figure 3 – Initialize the attributes

Figure 4 shows how to add nodes and edges to a graph. Features *connect* and *connect_directed* find existing nodes containing the values passed in as parameters or create new nodes if it does not find nodes containing those values. The features then create an edge between the two nodes. Feature *connect_nodes* adds an edge between the two nodes and adds the nodes to the graph.

```
build_graph_1
        -- Add nodes and edges to the graph number 1.
    do
        graph_1.connect ("A", "B")
        graph_1.connect ("A", "C")
            -- Demonstrate various feature calls
        graph_1.connect_directed ("D", "A")
        graph_1.connect_nodes (node_a, node_e)
        graph_1.connect ("B", "F")
        graph_1.connect ("C", "G")
        graph_1.connect ("F", "G")
        graph_1.connect ("D", "H")
        graph_1.connect ("H", "D")
        graph_1.connect_directed ("E", "I")
        graph_1.connect_directed ("I", "E")
    end
```

Figure 4 – Add nodes and edges to the example graphs

The code in Figure 5shows an alternative method using classes *NODE* and *EDGE* directly. This code creates the nodes X, Y, and Z and an edge between X and Y. The call to feature *adopt* from class *NODE* creates the edge between X and Z. At this point the three nodes and two edges exist, but they are not in a graph. To add these nodes and edges to a graph use features *extend_node*, *extend_edge*, or *deep_extend_node* from class *GRAPH*.

```
build_graph_2
        -- Add nodes (X, Y, and Z) and edges to the graph number 2.
    local
        x, y, z: VALUED_NODE [STRING]
        e: VALUED_EDGE [STRING]
    do
            -- Create the nodes
        create x.make_with_value ("X")
        create y.make_with_value ("Y")
        create z.make_with_value ("Z")
            -- Make an {EDGE} that connects two nodes
        create e.connect (x, z)
        e.set_directed
            -- Connect two nodes using feature from {NODE}
        node_x.adopt (y)
            -- Add node X to the graph (does not add any edges)
        graph_2.extend_node (x)
            -- Add the edge to the graph (includes nodes X and Z)
        graph_2.extend_edge (e)
            -- Ensure node X and all its edges (included any connected nodes) are in the graph
        graph_2.deep_extend_node (x)
    end
```

Figure 5 – Using class NODE and EDGE directly

The code in Figure 6 adds the final edge to the structure, showing how to optain a particular node from a graphs using feature *find_node*. It then uses feature *adopt* from class *NODE* to add an edge between the node just found and a previously created node without inserting that edge into either graph. (Nodes or edges linked together but not in a graph are not visible to the graph, which is useful for various interation schemes shown later.)

```
link_graphs
        -- Add the edge connecting the two graphs.
    do
            -- Find the nodes (object equality)
        check attached graph_2.find_node ("X") as n then
                -- Connect two nodes using a feature from {NODE}
            node_e.adopt (n)
        end
    end
```

Figure 6 – Searching for a node in a graph

Having completed the structure of the two graphs, Figure 7 shows the use of an iterator to visit the nodes of one of the graphs. The iterator in the code below visits each node in depth-first order, printing the value of each visited node. The output is: A B F G C E I.

```
traverse_graph_1
        -- Visit nodes in graph number 1.
    local
        it: VALUED_GRAPH_ITERATOR [STRING]
        n: VALUED_NODE [STRING]
    do
            -- Obtain an iterator for the graph
        it := graph_1.iterator
            -- Set the node from witch to start traversals
        n := graph_1.find_node ("A")
        it.set_root_node (n)
            -- Set traversal policies
        it.visit_nodes
        it.set_depth_first
            -- Visit nodes
        from it.start
        until it.is_after
        loop
            n := it.node
            io.put_string (n.value + " ")
            it.forth
        end
    end
```

Figure 7 – Using an iterator to traverse a graph

Notice that the iterator does not visit nodes D and H, because the directed edge from node A to node D points the wrong direction. Also, the iterator does not visit the nodes in graph number 2 (node X, node Y, and node Z) because because the edge between node E and node X is not in the graph and therefor not visible to the graph from which the iterator was obtained. A subsequent section on iterators shows how to allow the iterator to traverse such an edge and visit nodes in other graphs.

1.3. DOCUMENT LAYOUT

Before exploring iterators in more detail, this paper first details the structural classes for building graphs, exploring The Eiffel Graph Library's extensive use of contracts and looking at how trees fit into TEGL's structure. Next this paper explores the iterator classes, describing the various traversal methods and policies. Having detailed the structural and iteration classes, subsequent sections cover some design decisions and ideas for future improvement.

2. PROVIDING THE STRUCTURE

The library models the containers (e.g. graphs, weighted graphs, and b-trees) around the basic concept of a graph as a collection of nodes which are connected by edges. The three main classes in the library which provide and maintain the structure of a graph are *GRAPH*, *NODE*, and *EDGE*. Class *GRAPH* and its descendents are intended to be used directly. Classes *NODE* and *EDGE* and their descendents are relegated to a support role.

2.1. GRAPH

Class *GRAPH* is the top of the hierarchy. Features *default_create* and *make_with_capacity* are the creation features. Feature *default_create* creates a graph whose nodes can have multiple incoming and multiple outgoing edges, initialized to the *Default_in_capacity* and *Default_out_capacity*. Feature *make_with_capacity* allows the nodes to be created with enough room for the number of edges as given in the argument to the feature. In either case, the number of edges a node can contain can grow beyond this initial value.

A graph can be queried with *node_count* and *edge_count* to determine the number of nodes or edges, respectfully, that are in the graph. These values may differ, because multiple edges may lead to the same node. Features *has_node* and *has_edge* are boolean queries for testing whether a node or an edge is in the graph.

Nodes are added to the graph using feature *connect_nodes* which takes two nodes as arguments and creates a connection (i.e. an *EDGE*) between them. One of the preconditions, *is_connection_allowed* ensures the two nodes are in states that allow a connection between them. Feature *connect_nodes_directed* also creates an edge between the two nodes; the difference is that the new edge is a directed connection from the first node to the second node. The newly created edge can be obtained with *last_new_edge*.

Nodes and edges can also be added to a graph without creating a new edge using features *extend_node* and *extend_edge*. These features can be used to make a node or edge "visible" to the graph. Feature *prune_node* conversely removes a node and all its edges from the graph making them no longer "visible" to this graph; that is, they are no longer "in" the graph. The nodes and/or edges are not themselves changed; a node still has all its edges and the edges still connect the correct nodes. This allows a node having many edges to be "in" a graph, but perhaps some or all of its edges are not themselves in the graph. A node could also be in, and therefore manipulated by, more than one graph without harming its connections or the connections of other nodes. Similarly, feature *prune_edge* will remove an edge from the graph without disconnecting the edge. To disconnect and descard the edge between nodes that are in a graph, use feature *disconnect_nodes*. Use query features *has_node* and *has_edge* to determine if a node or an edge is in the graph.

Finally, feature *is_empty* is there for consistency with the kernel container classes and returns **True** if the graph contains no nodes or edges.

2.2. NODE

NODE is the most complex of the top structure-providing classes in the library. Besides providing *GRAPH* with the creation features, *default_create* and *make_with_order*, it provides the mechanics for creating and storing the edges to be seen by a graph. Feature *adopt*, taking one *NODE* as an argument, adds a child node by creating a new edge and connecting that edge from the current node to the argument node. Feature *disown*, again with one *NODE* as an argument, removes that child node by disconnecting any edges from the current node to the child. Complementary features *embrace* and *spurn* allow a node to take on a parent node (by creating an edge *from the parent to the current node*) or to remove a parent node, again by disconnecting the appropriate edge(s). There are also "deep" versions of these features.

The edges stored in a node can be accessed through features such as *i_th_edge*, *connections*, *connections_from*, *connections_to*, *children*, *parents*, *relations*, *ancestors*, and *descendants*. The number of edges in a node is obtained through features *count*, *child_count*, *in_count*, and *out_count*. Feature *order*, not to be confused with *count*, gives the number of edges the node is capable of containing (the capacity) at this moment.

Status features and their setters control the behavior of a node. If a node *can_adopt* then it is allowed to have out-going edges (i.e. to obtain children.) It is set by *allow_adopting* and *forbid_adopting*. If a node *can_embrace* then it is able to obtain an incoming edge (i.e. to obtain a parent.) If a node *can_embrace_multiple* then it is allowed to have more than one incoming edge (i.e. to have more than one parent.) The setter features are named similarly as above with the forbid and adopt prefix added to the feature name.

Other *CONTAINER*-type queries such as *is_empty*, *has_edge*, and *has_node* are also available.

2.3. EDGE

EDGE is the last of the three top-level, structural classes. Simply put, an edge is a connection between two nodes, a *node_from* and a *node_to*. If one of the nodes to which the edge connects is known, the other node can be accessed by feature *other_node*, taking as argument a reference to the known node.

Even though the feature names *node_from* and *node_to* imply the edge goes in one direction, that is not the intent. The names, chosen to distinguish the two nodes, simply seemed a better choice than names such as "node_1" and "node_2". If a directed edge is desired then give direction to an edge using feature *set_directed*. Feature *set_undirected* will return the edge to a directionless state. This status is reported by *is_directed*.

Three other status-reporting features, *is_connected*, *is_disconnected*, and *is_unstable* deserve attention. After creation an edge can be in one of three states. The first, *is_connected*, is the state normally expected where there is a node at both ends of the edge. Feature *connect*, which is also a creation feature, ensures the edge connects the two nodes passed as arguments. The other creation feature *default_create* will create an edge that is not connected to any nodes. While this might seem to be an odd state for an edge, it is in fact necessary, for example, when removing a child node from a parent node. (Remember that a child node is removed by calling *disconnect* on the appropriate edge.) The third state an edge might find itself in is when there is an edge at one end but not the other. This does not make sense in the context of graphs, but will develop as an intermediate state as one node and then the other is connected or disconnected from the edge. Feature *is_unstable* is used by the class invariant to distinguish this state, allowing the object to momentarily exist in this "semi-invalid" state. The invariant, though, ensures the edge ends up in one of the two stable states.

2.4. Contracts

Invariants and other contracts are used extensively in this library. Of particular interest is the way referential integrity is maintained between the two nodes in an edge and the edges contained in a node. Each node at the ends of an edge must contain that edge; if a node contains an edge then the node must be one of the nodes of that edge. In *NODE* the invariant calls *has_proper_connections* which checks all the edges contained in the node to ensure each edge *has_connection* to the current node. Also, the invariant in *EDGE* ensures that both nodes *has_connection* to the current edge. The two invariants are show in Figure 8.

```
-- from EDGE
    symmetrical_connections: is_connected implies
                (node_from.has_edge (Current) and node_to.has_edge (Current))
-- from NODE
    has_proper_connections: not is_unstable implies has_proper_connections
```

Figure 8 – Referential Intergety Invariants from *NODE* **and** *EDGE*

In addition the features for adding nodes to a graph (and connecting edges) are armed with many preconditions and postconditions. Recall feature *connect_nodes* from ***GRAPH***, shown in Figure 9, is used to add nodes to a graph by having the first node *adopt* the second node. Feature *extend_edge* is called near the end; it ensures that the *last_new_edge*, and hence the two nodes at the end of that edge, are added to and therefore in the graph. The pre- and post-conditions of *connect_nodes* illustrates some of the extensive use of contracts in this library.

```
connect_nodes (a_node, a_other_node: like node_anchor) is
        -- Make a connection between the two nodes, ensuring both nodes
        -- are visible to (i.e. "in") the graph.  The newly created edge
        -- connecting the two nodes can be accessed through `last_new_edge'.
        -- Side effect: if Current `is_ordered' then the two nodes will be
        -- ordered after this call, perhaps rearranging their edges.
    require
        node_exists: a_node /= Void
        other_exists: a_other_node /= Void
        node_can_adopt_child: a_node.can_adopt
        other_node_is_adoptable: a_other_node.is_adoptable
        other_node_is_adoptable: a_other_node.is_adoptable
        can_add_node: is_extendable_node (a_node)
        can_add_other_node: is_extendable_node (a_other_node)
        allowable_connection: is_connection_allowed (a_node, a_other_node)
    do
        a_node.adopt (a_other_node)
        if is_ordered then
            if not a_node.is_ordered then
                a_node.set_ordered
            end
            if a_other_node.is_ordered then
                a_other_node.set_ordered
            end
        end
        extend_edge (last_new_edge)
    ensure
        node_inserted: has_node (a_node)
        other_node_inserted: has_node (a_other_node)
        new_edge_in_graph: has_edge (last_new_edge)
        proper_from_connection_made: last_new_edge.originates_at (a_node)
        proper_to_connection_made: last_new_edge.terminates_at (a_other_node)
    end
```

Figure 9 – Feature *connect_nodes* from *GRAPH*

Figure 10 shows the implementation of feature *adopt*, from *NODE*, giving another example of a very heavily contracted feature. While not a true measure of the quality of the software, the number of lines dedicated to the contracts is much higher than the number of lines in the body.

```
adopt (a_child: like node_anchor)
        -- Add a connection (EDGE) from Current to `a_child'.
    require
        can_adopt_children: can_adopt
        child_exists: a_child /= Void
        child_is_adoptable: a_child.is_adoptable
    local
        e: like edge_anchor
    do
        e := new_edge
        e.connect (Current, a_child)
    ensure
        connection_exists: has_node (a_child)
        connection_to_exists: has_edge_to (a_child)
        count_increased: edge_count = old edge_count + 1
        out_count_increased: out_count = old out_count + 1
        in_count_unchanged: in_count = old in_count
        last_edge_originates_at_current: last_new_edge.originates_at Current)
        last_edge_terminates_at_a_child: last_new_edge.terminates_at (a_child)
    end
```

Figure 10 – Feature *adopt* from *NODE*

Features in the library rely very heavily on such contracts as these. Another example is in feature *disconnect_nodes* from *GRAPH* which requires that the two nodes being disconnected are contained in the graph; a graph should not change nodes it does not own. This check could have been done in the body and then no action taken if one or both of the nodes was not in the graph, but that would have required the graph to search [twice] just to determine if it should take action

One of the preconditions to *connect_nodes* from *GRAPH* deserves a closer look. Feature *is_extendable_node* is called for each node passed to *connect_nodes* to ensure the nodes which the graph is being asked to connect can be added to the graph. Feature *is_extendable_node*, shown in Figure 11, first checks to make sure the node is non-void.[2]

```
is_extendable_node (a_node: like node_anchor): BOOLEAN is
        -- Can `a_node' be put into and manipulated by Current?
        -- If `is_tree_mode' then Current can only manipulate nodes which
        -- have or can have only one parent.
        -- If Current is not acting as a tree it can take any non-void node
        -- and change the node.  This could violate the invariant of another
        -- graph if that other graph contains `a_node' and is acting as a
        -- tree; Current could ask a singly-parentable node to take on a
        -- second parent.
    do
        Result := a_node /= Void and then
                      ((not is_tree_mode) or else
                      (is_tree_mode and then a_node.can_embrace_multiple))
    end
```

Figure 11 – Feature *is_extendable_node* from *GRAPH*

Then, if the graph is acting as a tree it verifies that the node can have at most one parent. The assertions in these examples give a sampling of how extensively contracts are used in this library. The last one alludes to the use of contracts as they apply to trees.

[2] Conversion to void-safe code would elliminate the need for this and many other checks.

2.5. Trees

GRAPH, *NODE*, and *EDGE* are the top nodes in the library's hierarchy and provide the features for creating and maintaining the structure of a graph. One would expect a library such as this to contain a tree class, since a tree also imposes further structure onto a graph. For starters, a tree must be a connected and acyclic graph. This could have been obtained by the addition of a tree class, but for various reasons this class was not included.

As Meyer wrote in the first edition of Object Oriented Software Construction [**Error! Reference source not found.**], finding the classes is not always easy. It would seem as if a tree class would naturally follow from the definitions in graph theory because a tree "is a" graph with special properties. But, carrying this logic to the absurd, one would end up not just with a tree class but with classes for directed graphs, undirected graphs, multigraphs, colored graphs, weighted graphs, disconnected graphs, cyclic and acyclic graphs and a cornucopia of other classes in a combinatorial explosion when these properties must be combined.[3] Instead of providing separate classes for all these combinations, this library uses the status features in class *GRAPH* to provide some of the desired properties.

Fix next paragraph; no longer correct.

Feature *is_cyclic* was discussed above. In addition feature *is_tree_mode* is true if a graph is behaving as a tree. If a graph *is_acyclic* and is not *is_disconnected*, then the graph can be considered a tree and can be forced into a "tree mode" with a call to *set_tree_mode*. This will ensure that future additions to the graph comply with the rules for a tree. In addition this property can be set at creation time by using one of the creation features *make_singly_parentable* or *make_singly_parentable_with_order*, inherited from class *NODE*, to make a graph whose nodes can have at most one parent. These features set *can_embrace_multiple* to false and ensure all nodes added have that property as well. (Remember that *is_extendable_node* from *GRAPH* only allows nodes which, themselves can have at most one parent if the graph can only have one parent.)

If *is_tree_mode* is true for a graph then the *root_node* attribute becomes available. Remember that a *GRAPH* object, in this case a tree, is itself one of the nodes in the tree. Depending on the order of connections, the tree could grow up by adding new parents, moving the root up, or down by adding children, leaving the root where it was. So, the *root_node* is not necessarily, and most likely is not, the *GRAPH* object. The feature will continue up the *parent_edge* (from *NODE*) until there is not a parent. That node then is the root of the tree.

If *is_tree_mode* is true for a graph then other measurement features become available. Feature *height* gives the length, in number of edges, of the longest path from the *root_node* down to a leaf node; feature *weight* gives the number of leaf nodes in the graph; feature

[3] Development of this library went through a version which did have a tree class, but when generic containers were added the number of classes and therefore the complexity of the interface grew disproportionally to the benefit gained. Renaming of the creation procedures and changes in feature exports were also overly complicated. The current implementation of class *GRAPH* which must acount for tree properties is more complex but the interface is simplified.

breadth gives the number of nodes at a particular level in the graph; and feature *node_depth* gives the distance, again in number of edges, that a particular node is from the *root_node*.

The invariants of class *GRAPH*, shown in Figure 12, ensure a graph maintains tree properties if it was asked to by a call to *set_tree_mode*. In order to maintain tree properties

```
invariant

     tree_mode_implication: is_tree_mode implies is_tree
     is_tree_implication: is_tree implies (is_asyclic and then
                                             not is_disconnected and then
                                             not has_multi_parented_node)
```

Figure 12 – Invariant from *GRAPH*

during node additions to a graph for which *is_tree_mode* is true the preconditions to the connection features must also be "modified".

Referring back to Figure 9, in the preconditions of *connect_nodes* from *GRAPH* we see a call to feature *is_connection_allowed*, shown in Figure 13, which forces the caller to guarantee that nodes being connected in [a graph behaving as] a tree meet tree-node conditions. First the nodes must both be extendable into the graph as seen earlier with *is_extendable_node* for a graph. The first node must be able to adopt a child node and the

```
is_connection_allowed (a_node, a_other_node: like node_anchor): BOOLEAN is
          -- Is Current allowed to make a connection between the two nodes?
          -- If `is_tree_mode' then one of the nodes must already be in the graph.
          -- (Remember, Current is always "in the graph" so we can start by
          -- adding connections to Current.)
do
          if is_extendable_node (a_node) and then is_extendable_node (a_other_node) and then
                    a_node.can_adopt and then a_other_node.is_adoptable then
               if is_tree_mode then
                    Result := (has_node (a_node) or else has_node (a_other_node))
                              -- no need to check for induced cycle because the only way to
                              -- get a cycle is to allow nodes to obtain more than one parent;
                              -- this is prevented by `is_extendable_node' which makes sure a
                              -- node can not embace multiple parents.
               else
                    Result := True
               end
          end
     end
```

Figure 13 – Feature ***is_connection_allowed*** **from** *GRAPH*

second node must be able to accept a parent. This ensures the node properties. Finally, if *is_tree_mode*, at least one of the two nodes must already be in the graph in order to maintain the not *is_disconnected* invariant. A graph that is a tree, then, will be a connected-acyclic graph whose nodes have at most one parent and which restricts the types of nodes added so they also comply with tree properties.

So, having explored some of the features and contracts of *NODE*, *EDGE*, and most of the features of *GRAPH*, it is time to cover, as promised earlier, the one other important feature of *GRAPH*.

3. BEYOND STRUCTURE

At this point we have seen how to build a graph using feature *connect_nodes* from *GRAPH* and *adopt* from *NODE*, but there has been no discussion about how individual nodes or edges in the graph can be accessed beyond maintaining a reference to the nodes as they are passed into the graph. The individual nodes and edges in the graph can be reached by using an iterator. Feature *iterator* in class *GRAPH* makes a new object of type *GRAPH_ITERATOR*, an external iterator allowing several types of traversals over a graph with the ability to get the currently pointed-to *node* or *edge*.

3.1. GRAPH_ITERATOR

An iterator for a graph is created with feature *make*, taking an object of type *GRAPH* and the start point for the traversal is set in *make* or later by a call to *set_root_node* whose argument node must be in the graph for which the iterator was created. The creation feature calls *set_breadth_first* (Figure 14) allowing the graph to be traversed by visiting

```
set_breadth_first is
            -- Make Current traverse the graph in breadth-first order.
            -- Begin at the root node and explore all the neighboring nodes.  Then for each of
            -- those nearest nodes, it explores their unexplored neighbor nodes, and so on.
            --            1
            --          / | \
            --       2   3   4
            --      / |       | \
            --    5  6      7  8
            --   / |          | \
            -- 9 10       11  12
```

Figure 14 – Comment for *set_breadth_first* from *GRAPH_ITERATOR*

each of the nodes in "breadth-first" order using the normal *start*, *forth*, *back*, *is_after*, etc. pattern. The different traversal methods can be set by calls to features shown in the figures below.[4]

```
set_depth_first is
            -- Make Current traverse the graph in depth-first order.  Start at the root
            -- and explore as far as possible along each branch before backtracking.
            -- (http://en.wikipedia.org/wiki/Depth-first_search)
            --            1
            --          / | \
            --       2   7   8
            --      / |       | \
            --    3  6      9  12
            --   / |          | \
            -- 4  5       10  11
```

Figure 15 – Comment for *set_depth_first* from *GRAPH_ITERATOR*

[4] Describing the order in which the nodes are traversed naturally produces a tree-like representation, however the traversals do not produce trees just a list of paths.

```
set_post_order is
        -- Make next `sort' be in post-order (i.e. visit all children then the parent)
        --           12
        --         / | \
        --       5   6   11
        --      /|        |\
        --    3  4       9  10
        --   /|          |\
        --  1  2        7  8
```

Figure 16 – Comment for *set_post_ordert* from *GRAPH_ITERATOR*

```
set_in_order is
        -- Make Current traverse the tree in `in_order' fashion.  Traverse the left
        -- subtree, visit the root, traverse the right subtree, etc.  This will visit the
        -- lowest leaf, parent, remaining leaves, parent, etc.
        --           6
        --         / | \
        --       4   7   11
        --      /|        |\
        --    2  5       9  12
        --   /|          |\
        --  1  3        8  10
```

Figure 17 – Comment for *set_in_order* from *GRAPH_ITERATOR*

Figure 18 shows two other traversal methods not usually encountered in the literature on graphs but may be useful in some applications. The first is a bottom-up traversal which visits nodes furthest away from the *root_node* first, then visits the nodes one level closer to the root, and so on. The second is a leaf-first traversal which visits nodes by moving down the branches in order until reaching a leaf.

```
set_bottom_up
        --          12
        --        / | \
        --      9  10  11
        --     /|       |\
        --   5  6      7  8
        --  /|         |\
        -- 1  2       3  4
```

```
set_leaf_first
        --          12
        --        / | \
        --      9  4  11
        --     /|      |\
        --   8  3    10  7
        --  /|        |\
        -- 1  2      5  6
```

Figure 18 – Traversal Order for *set_bottom_up* and *set_leaf_first*

3.2. Other Traversal Settings

So far in the discussion of class *GRAPH* and the description of possible traversal methods provided by *GRAPH_ITERATOR* there was an assumption that each node to be visited was "in" the graph; that is not always required in this library. There has been no mention of a class encompassing the concept of directedness (i.e. a "directed graph" class) as found in most graph-theory literature and other graph libraries; the descusion below will show how directedness can be added to a graph. Finally, the traversal policies, from breadth-first traversal to leaf-first traversal, have assumed the object of the traversal was to visit each node in a graph with no mention of the edge traversed or path explored; this class allows these other traversal policies.

Normally an iterator is created from a graph and the iterator is used to visit nodes that are "in" the graph starting at a *root_node* that is also "in" the graph. In this case feature *is_seeing_reachables* is in its default state of false. To change this, use feature *see_reachables* which allows nodes connected to an arbitrary node to be visited without first putting all the nodes into a graph. This allows the iterator to "see" nodes that are connected to the *root_node* but may not be "in" the *graph*. This need could arise if one wishes to visit all nodes connected to a node that is in two graphs with some of its descendents in one graph and other descendants in another graph. Feature *see_visibles* returns the iterator to the default state.

Directedness was touched briefly in the discussion of feature *connect_nodes_directed* from class ***GRAPH*** and in the discussion of class ***EDGE*** and its features *set_directed*, *set_undirected*, and *is_directed*. This library does not contain a class which models a directed graph; this "directedness" is delegated to each edge in the graph, allowing the user to mix directed and undirected edges in one graph. An example application would be the modeling of a map of city streets containing both two-way and one-way streets. The traversal features check *is_directed* for each edge as it is encountered to determine if a traversal down that edge from the current node to the node at the other end of the edge is allowed. If a truly directed graph is desired then *inspect_children* will cause the traversal to proceed only from the *node_from* of each edge to the *node_to* of each edge; *inspect_parents* will cause the traversals to go the other way. The default setting for a graph is to *inspect_relations* which causes the traversal to treat the two nodes of an edge equally, giving the traversal the opportunity to move in either direction down the edge. Nevertheless, an *edge* which *is_directed* can be traversed in only one direction.

But this directedness is not the only factor used by the traversals to determine whether or not to visit a node. So far the policy has been to *visit_nodes*, stepping to each node reachable by the chosen traversal method until all nodes have been visited and visiting a node no more than once. But some applications require not only each node to be visited but also requires each edge in the graph to be traversed which may lead to nodes being visited multiple times. Other applications may require each possible path (or walk) to be explored, allowing multiple visits to nodes and repeated traversals of edges. Class ***GRAPH_ITERATOR*** exports status-setting features which will modify whatever traversal method is in use at the time (e.g. breadth-first, depth-first, etc.) to accommodate these needs. Feature *traverse_edges* causes the traversal to continue until all reachable edges have been traversed and *explore_paths* allows the traversal to continue until all reachable paths have been explored.

3.3. PATH

The concept of a path is encapsulated in the final support class called, predictably enough, ***PATH***. A path begins at the *first_node*, set by the creation feature *make*, and proceeds down a list of edges to the *last_edge* and *last_node*. The *cost* of a path is simply the number of edges in the path (the number of edges to traverse along this path to travel from the *first_node* to the *last_node*.) Edges are added to a path with *extend*. Because a path is an unbroken list of edges from one node to another, any new edge to be added "should" have one node in common with the *last_node*. A precondition that the edge *has_connection_to* or *has_connection_from* the *last_node* would be desirable, but this is not possible as that would require a strengthening of the precondition to *extend* which hales from *COLLECTION*. So, to check if a new edge is valid, a check statement is added in the version of *extend* redefined here in *PATH*.

The discussion of class *PATH* gives some insight into the implementation of the traversal methods of an iterator. As the iterator moves forward to the next edge (in order to obtain the next node) a new path is built and added to a list of paths already explored. The current *edge* and *node* is obtained by accessing the *last_edge* and *last_node* of the last path added to the list. In other words, each path in the internal list gives one trail of edges that were traversed in order to get from the *root_node* of the iterator to the *last_node* of that path, with the last path in the list pointing to the current trail of edges. This "current" path is access using feature *path*.

3.4. Moving Back and Forth With Iterators

It is easy to see, then, that moving the iterator backwards is simply a matter of getting the previous path from the internal list. Or is it? If the graph has changed between the last call to *forth* and this call to *back*, the path now referenced may be invalid; that is, one of the nodes visited or edges traversed along the *path* may no longer be in the graph.[5] Figure 19 shows a simple graph and paths explored during a breadth-first traversal. Remember that a

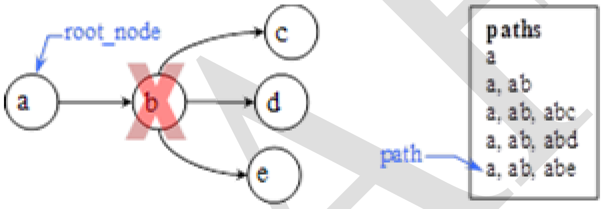

Figure 19 – Node Deletion Affects Cursor Movement

PATH is a *start_node* followed by a list of edges. In the figure, *path* references an object with *start_node* "a" and a list containing an edge from "a" to "b" and an edge from "b" to "e". The iterator has visited the nodes in the order "a-b-c-d-e", so the "current" *node* would be node "e", the last node of the *path*. Calling *back* would move the *path* cursor back to path "abd", causing the iterator to return node "d" on a call to *node*. But if node "b" was deleted prior to the call to *back*, to which path should the *path* cursor be moved and which *node* should be the current one?

Class *GRAPH_ITERATOR* and its descendents implement feature *back* so that invalid paths are ignored. In the example any path going through node "b" (paths "abd", "abc", and "ab") are no longer valid, so a call to back would move the *path* cursor back to path "a". Feature *node* would then return the same node as feature *root_node*.

This example illustrates a problem with using an external iterator instead of an internal one as used in class *LINKED_LIST* and others from the base library. The structure [of the graph] could change and the iterator would not know it. In fact, this problem arises not just for feature *back* but all features of class *GRAPH_ITERATOR* that rely on the current

[5] This is analogous the backward traversal of a *LINKED_LIST* where a previously accessed item may no longer be in the list, so moving backwards would give a different "previous" item or may even go off the front of the list.

position. Feature *node*, *edge*, and *path* are three of these. To prevent an iterator from returning a value that is invalid, such as a node that is no longer in the graph, an edge that is not connected to two nodes, or a path containing a disconnected edge, the iterator can check the validity of the current path and keep searching until it finds a valid path.[6]

This path validity checking can slow the iteration immensely, so this checking is turned off by default; it is assumed that most iterations go from *start* until *is_after* with no mid-iteration changes to the graph. This status is queried with feature *is_validate_mode* and is turned on and off with features *set_validate_mode* and *set_unsafe_mode*. When *is_validate_mode* is true the path validity checking will be done.

The use of external iterators may have made the implementation of the iterator classes more difficult, but it simplified the development of the structural classes because those classes do not have to worry about maintaining a "cursor" as the structure changes. In addition, it allows a graph to be traversed in various ways by multiple iterators at the same time and allows the traversal method to be changed in mid-course.

3.5. Mode Changes During Graph Traversals

Setting the traversal method of an iterator using features such as *set_breadth_first*, *set_depth_first*, and so on as described in sections 3.1 and 3.2 does not lock the iterator forever in that mode. Figure 20 depicts a graph and the list of paths created when the graph was traversed from beginning to end, but with a change mid-course in the traversal

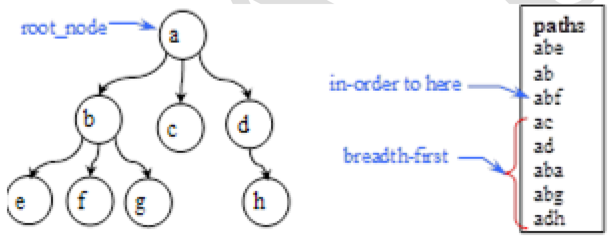

Figure 20 – Changing Traversal Method

method. The graph was traversed in order from the root node through nodes "e" and "b" to node "f" via paths "abe", "ab", and then "abf". At that point the traversal method was changed to breadth-first by calling *set_breadth_first*. Since breadth_first_forth looks for the children of the first "queued" node, the children of "a" are visited next, producing paths "ac" and "ad", but node "b" is not visited again. Next in breadth-first order are the children of "b". Nodes "e" and "f" have already been visited, so they are not visited again. It looks at this point as if node "g" should be next. It is not so obvious, though, that node "b" has *two* unvisited children. Remember that if the graph *is_visiting_relations*, which is the default setting, the direction of the edges is ignored meaning that node "a" is also an unvisited child of node "b". If nodes "a" and "b" were connected before nodes "b" and "g" then node "a" would normally be visited first, then, as the last child of node "b", node "g"

[6] A different implementation for preventing invalid cursor movement through a changing graph was considered. A proposal to allow a graph to know [i.e. keep a reference to] each iterator created for that graph was discarded because every change to the graph would require reshufling of each iterator referenced which was deemed to place too much overhead on a graph.

would be visited. The breadth-first traversal of the remaining nodes is straight forward. Node “c” has no children so the one child of node “d” is visited to complete the traversal.

3.6. NODE AND EDGE ORDERING IN A GRAPH

In the example traversal of the graph in Figure 20 above, the order in which node “a” and node “g” was visited depended on the order in which they were connected to node “b”. The default setting for graphs is to place the edges into the nodes in the order they are connected to that node (i.e. the edges are added at the end of the list of edges.) Sometimes a sorted order is desired where the children of a node should be in alphabetical order. Other graphs may require the edges to be ordered with the shortest edges first. To build a graph with this ordering use feature *set_ordered* which sorts any edges already in the graph and places new edges in their proper sequence. Feature *sort* is also available to order edges already in the graph, but subsequent edge additions will be placed at the end of the edge list in the nodes.[7]

This ordering ability implies that there is a comparison operation going on between nodes and edges. Both classes, *NODE* and *EDGE*, inherit from *COMPARABLE* and redefine feature *“<”*. At this level in the hierarchy an edge is less than another edge if the node from which it originates (*node_from*) is less than the other’s originating node. If the originating nodes are equal then the terminating nodes (*node_to*) are compared.

This begs the question, “How are nodes compared?” A node is less than another node if the number of edges in the node is greater than the number of edges in the other node. This places nodes with the most children first. If the number of edges in the nodes is the same then each edge in the node is compared to the corresponding edge in the other node using the edge comparison described above. These comparison operations become more straight forward further down in the class hierarchy where edges have costs and nodes can take on values; these costs and values can be compared. But before seeing these descendent classes, let us look at the hierarchy of the classes described up to this point.

3.7. THE BIG PICTURE (…SO FAR)

Figure 21 depicts the library as described so far showing the structural class *GRAPH* and its support classes *NODE*, *EDGE*, *PATH*, and *GRAPH_ITERATOR*.

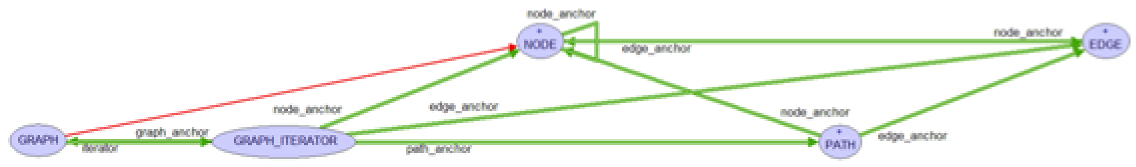

Figure 21 – Upper-level Classes

[7] These features do have the side affect of changing each node that is in the graph; the original edge ordering in the nodes is lost.

None of these classes are deferred; each is fully usable as is. However, a graph with simple nodes and edges while interesting to theorists is not of much practical use in programs. So now it is time to reveal the rest of the library[8].

4. The Rest of the Story

The classes seen so far allow a user to build a graph complete with nodes and edges, and the iterator classes, along with the path classes, allow various traversals of the graph. But missing is the ability to put data or items into a graph and use the graph's structure to organize the data. One approach could be to inherit from *NODE* and/or *EDGE* which would allow the objects, which now conform, to be manipulated by an object of type *GRAPH*. But, this approach puts a heavy burden on the developer. An easier way, similar to the use of the base container classes, such as *LINKED_LIST* and *ARRAY*, would be to use this library's generic container classes—the descendents of *GRAPH*.

4.1. Containers, Finally

The generic descendent classes of *GRAPH* are parameterized around the way data can be stored in a graph. Data can be held in the nodes or on the edges. The data on the edges is either simply that, information, thought of as a label or name for the edge, or it is a numerical value representing the cost of traversing that edge. The generic parameters *N*, *E*, and *C* are used to differentiate these three ways of looking at the data. *N* is used for the **N**ode-data to be stored in the nodes, *E* is used for **E**dge-labels, and *C* stands for the **C**ost of traversing an edge.

Figure 22 shows all the container classes along with their generic parameters. Below *GRAPH* is class *VALUED_GRAPH [N]*. This would be for a graph holding nodes containing data of type *N*. For example, the diagram in Figure 22 could be modeled as *VALUED_GRAPH [CLASS_NAMES]*. Class *LABELED_GRAPH [E]* is for a graph where the edges contain some type of data but there is no real "cost" to traverse the edges. (Exploring a path, of course, has a cost which is predicated on the number of edges in the path; the longer the path the higher the cost of walking it.) A *VALUED_GRAPH [COLOR]* might be used in a map coloring problem. Class *WEIGHTED_GRAPH [C]* is for graphs which have a traversal cost associated with each edge.

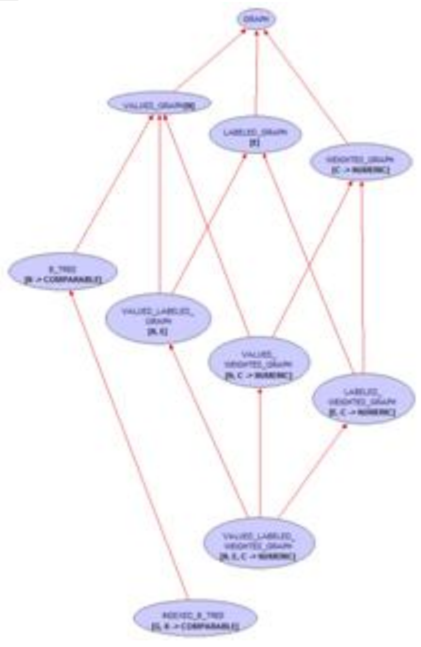
GRAPH
VALUED_GRAPH[N]
LABELED_GRAPH [E]
WEIGHTED_GRAPH [C -> NUMERIC]
B_TREE [N -> COMPARABLE]
VALUED_LABELED_GRAPH [N, E]
VALUED_WEIGHTED_GRAPH [N, C -> NUMERIC]
LABELED_WEIGHTED_GRAPH [E, C -> NUMERIC]
VALUED_LABELED_WEIGHTED_GRAPH [N, E, C -> NUMERIC]
INDEXED_B_TREE [G, N -> COMPARABLE]

Figure 22 – Containers

Class *B_TREE [N ->COMPARABLE]* models a b-tree whose values of type *N* are stored in order. The

[8] The "anchor" attributes, depicted in Figure 21 as client links, are covariantly redefined features to which other featuers and result types are anchored. The use of these anchor features simplified development and will be discussed later.

creation procedures for class *B_TREE* sets the graph to tree mode as discussed above. Class *INDEXED_B_TREE [G, K -> COMPARABLE]* stores items of type *G* indexed by a key of type K. This class is analogous to a *HASH_TABLE [G, K -> HASHABLE]* except the keys do not have to be hashable.

A Note about b-trees

If items already in a b-tree are changed the tree may not then be ordered correctly. If an item must be changed it may be better to first remove it from the tree, change it, then insert it back into the tree. This will ensure it ends up in the correct spot. Another alternative is to set_validate_mode. This will make the tree ensure the items are in the correct order and if not it will reorganize the tree. If the tree is large this could take a long time to complete.

Two of these data typing properties are combined in the descendents at the second level in the hierarchy in classes *VALUED_LABELED_GRAPH [N, E]*, *VALUE_WEIGHTED_GRAPH [N, C]* and *LABELED_WEIGHTED_GRAPH [E, C]*.

All three come together in class *VALUED_LABELED_WEIGHTED_GRAPH [N, E, C]*. This bottom-most class could be used to model a highway system where *N*, the node-data type, could be a *CITY*; *E*, the edge-label type, a *STREET*; and *C*, the edge-cost type, a *REAL*. The declaration for this hypothetical class would be *MAP [CITY, STREET, REAL_32]*. The last parameter, the *REAL32* value, might represent the "difficulty of driving a particular street" and could be computed from information such as the length between two cities, the road condition, or the number of lanes which would be stored in each street. The corresponding support classes for the node, edge, and path-types would automatically follow. Figure 23 shows sample code for this example.

```
local
    New_york: CITY
    map: VALUED_LABELED_WEIGHTED_GRAPH [CITY, STREET, REAL_32]
    iterator: GRAPH_ITERATOR
    route: PATH
    travel_difficulty: REAL_32
do
    …
        -- Assume all objects have been created and items added to the graph
    iterator := map.iterator
    iterator.set_start_node (New_york)
        -- Move forth until finding the destination city.
    …
    route := iterator.path
    travel_difficulty := route.cost
end
```

Figure 23 – Example Libray Usage

Near the end of the code the "travel_difficulty" entity is assigned the *cost* of exploring (or walking) the selected route. As stated this cost could be calculated from the street objects stored in the map. This is achieved by using feature *set_cost_agent* from …

Path or weighted path?

Fix me!!

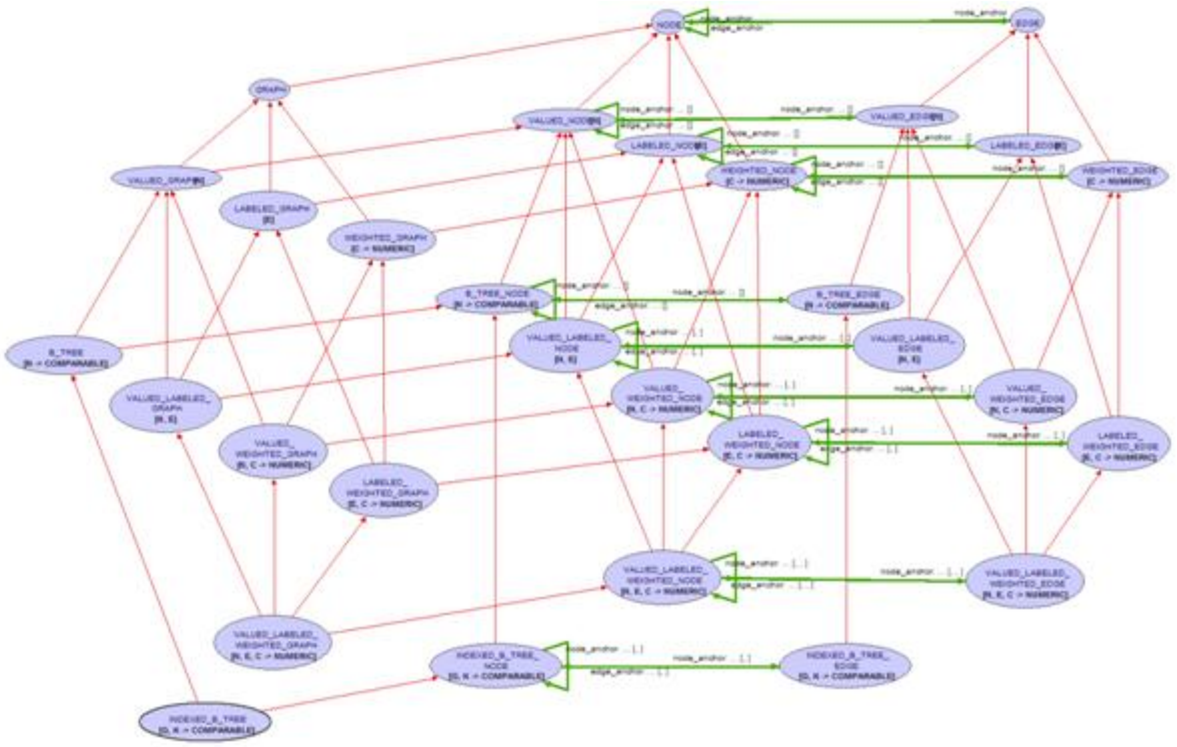
NODE
EDGE
node_anchor
edge_anchor
GRAPH
VALUED_GRAPH[G]
LABELED_GRAPH [K]
WEIGHTED_GRAPH [C -> NUMERIC]
VALUED_NODE[G]
VALUED_EDGE[G]
LABELED_NODE[K]
LABELED_EDGE[K]
WEIGHTED_NODE [C -> NUMERIC]
WEIGHTED_EDGE [C -> NUMERIC]
B_TREE [K -> COMPARABLE]
B_TREE_NODE [K -> COMPARABLE]
B_TREE_EDGE [K -> COMPARABLE]
VALUED_LABELED_GRAPH [G, K]
VALUED_LABELED_NODE [G, K]
VALUED_LABELED_EDGE [G, K]
VALUED_WEIGHTED_GRAPH [G, C -> NUMERIC]
VALUED_WEIGHTED_NODE [G, C -> NUMERIC]
VALUED_WEIGHTED_EDGE [G, C -> NUMERIC]
LABELED_WEIGHTED_GRAPH [K, C -> NUMERIC]
LABELED_WEIGHTED_NODE [K, C -> NUMERIC]
LABELED_WEIGHTED_EDGE [K, C -> NUMERIC]
VALUED_LABELED_WEIGHTED_GRAPH [G, K, C -> NUMERIC]
VALUED_LABELED_WEIGHTED_NODE [G, K, C -> NUMERIC]
VALUED_LABELED_WEIGHTED_EDGE [G, K, C -> NUMERIC]
INDEXED_B_TREE [G, K -> COMPARABLE]
INDEXED_B_TREE_NODE [G, K -> COMPARABLE]
INDEXED_B_TREE_EDGE [G, K -> COMPARABLE]

Figure 24 – The Eiffel Graph Library

4.2. The Big Picture

Figure 24 depicts the hierarchy with the container classes on the left and two support classes *NODE* and *EDGE* with their descendents at the middle and right. Classes *PATH* and *GRAPH_ITERATOR* and their descendants are not shown in order to avoid clutter.

All the container classes have an *iterator* anchored to a *path* as shown for the upper-level classes in Figure 21.

The green client relation links, named *xxx_anchor*, between the node classes and the edge classes are features that are not exported and should not be called even by the enclosing class; they were used to ease development of the library. The features are covariantly redefined in descendents so all features taking, in this case, arguments of type *NODE* or *EDGE* will follow the redefinition of the class. For example, feature *connect_nodes* from *GRAPH* takes two nodes as arguments. The arguments are defined to be of type **like** *node_anchor*. So if feature *connect_nodes* is called on an object of type *VALUED_GRAPH [STRING]*, it would expect the two arguments to be of type *VALUED_NODE [STRING]*, and the edge eventually created between the two nodes will be of type *VALUED_EDGE [STRING]*.

4.3. Covariance and Catcalls

The flexibility gained by this covariant redefinition greatly simplified the development of the library but, as described in [3], this can lead to the "risk of run-time type failures known as *catcalls*." Figure 25 shows code illustrating this problem. Entity integer_node conforms to *NODE* so it can be passed as a parameter in a call to *connect_nodes* on target g, because g, as a *GRAPH* simply expects objects of type *NODE* as arguments. The problem though, is that g is attached to a *VALUED_GRAPH [STRING]* object which expects two *VALUED_NODE [STRING]* arguments. The compiler will not catch this and this will most likely lead to a segmentation violation error.

```
cause_cat_call is
        -- Attempt to produce a cat call on the covariantly redfined classes
    local
        string_node: VALUED_NODE [STRING]
        integer_node: VALUED_NODE [INTEGER]
        string_graph: VALUED_GRAPH [STRING]
        g: GRAPH
    do
        create graph.make_with_value ("the graph")
        create string_node.make_with_value ("a node")
        create integer_node.make_with_value (123)
        g.connect_nodes (graph, string_node)                -- okay
        g := string_graph
        g.connect_nodes (graph, integer_node)               -- produces a catcall
    end
```

Figure 25 – Catcall Illustration

Class *B_TREE* is even more problematic in respect to catcalls because it changes the export status of many features. For example, a b-tree cannot call *connect_nodes*, because as items are added to the tree, the tree itself is responsible for creating the nodes in which items will be placed. So, a polymorphic assignment could not only pass in the wrong type of nodes but could also call features that should not be allowed for that target.

If possible avoid the type of polymorphic assignments depicted in Figure 25. For example, when accessing a node from an object of type *B_TREE [STRING]* assign the node to an entity of type *B_TREE_NODE [STRING]* not an entity of type *NODE*.

Section 3.2 said that an object of type *GRAPH_ITERATOR* could be created which is not associated with a particular graph. That could also lead to Catcalls. If an object of type *GRAPH_ITERATOR* was created to explore descendants of an object of type *B_TREE_NODE*, then the nodes obtained from the iterator may appear as simply a *NODE* type and not as a *B_TREE_NODE*, giving entities access to improper features.

This shows that care should be exercised when using the library. The intent of the library is for the descendent classes of *GRAPH* to be used as containers to hold the real objects of interest. The support classes such as *NODE*, *EDGE*, and *PATH* and their descendents are to be used only in rare cases. Also, while some polymorphic assignments should work in some cases, try to match the container types to the support types as closely as possible.

5. PERFORMANCE

If this catcall situation is avoided and this library is used as described, systems requiring graphs or trees could be easily modeled. But how do these classes fair in the performance arena? To answer this question the classes were compared to comparable classes in the elks structures cluster.

5.1. VALUED_GRAPH VRS. LINKED_LIST

5.2. B_TREE VRS. SORTED_TWO_WAY_LIST

5.3. INDEXED_B_TREE VRS. HASH_TABLE

6. CONCLUDE ME

…it adds much flexibility to the development of new systems and provides functionality not found in the Eiffel libraries up to now. The library is a practical implementation of …

7. NOTES ON IMPLEMENTATION

7.1. FEATURE "<" IN NODE AND EDGE

When thinking of a node as containing a value it is easy to envision that the comparison of two nodes would be done by comparing the value in the two nodes. Or, with an edge containing a cost or a "value on the edge", comparisons would be done based on the costs in the two edges. So, in the first approximation of this library I attempted to defer feature *infix "<"* in the support classes, *NODE*, *EDGE*, and *PATH*.

But, I wanted all the classes to be effected. Finish this section……

7.2. JJ_ARRAYs

This library relies on container classes from another cluster called "jj_support". These classes are all descended from *ARRAYED_LIST* because arrays may tend to be faster in the b-tree classes than a linked structure. Class *JJ_ARRAYED_LIST* changes feature *prune* to remove the first occurrence of an item starting at the beginning of the list, not from the current position. I felt that was more intuitive and prevented having to call *start* every time an item needed deleting. The class keeps the one-argument feature *replace* (renamed), but adds a two-argument *replace*. Instead of replacing the "current" item, it goes to the position of an item and then replaces the item with the new one. Again, this seemed more intuitive for the situations called for in this library.

I also wanted the ability to optionally store the nodes and edges in a sorted order. Class *JJ_SORTABLE_ARRAY* provides a *sort* feature and features for inserting in order is desired. Feature *item_position* for finding an item in an array or for finding the position where a new item should go was indispensable for *JJ_B_TREE*.

Finally, class *JJ_ARRAYED_STACK* behaves just like its ancestor, the base class *ARRAYED_STACK*, but allows items other than the top one to be inspected. This was needed for class *PATH* which needs to move through or access each edge in the path but must only add or delete edges from one end.

Iteration through these arrays was performed using an integer counter with feature *i_th*. I found that using *start*, *item*, *forth* would sometimes violate contracts because an invariant or other called feature would iterate over the same structure that was currently being iterated over.

7.3. Covariant Redefinitions With "Anchors"

I wanted to avoid the proliferation of classes seen in other attempts at a graph library. Sometimes there are classes for acyclic graphs, directed graphs, undirected graphs, disconnected and connected graphs, and so on. This "class explosion" occurred to some extent in this library because a particular node, edge, and path type had to be matched to the graph type, but through the use of the "anchor features" the library structure remained overall simple and the redefinitions were brainless (except for *INDEXED_B_TREE* which became quite complicated.)

To ease development "anchor features" were used in all the classes to provide types for the nodes, edges, paths, and graphs. For example, class *EDGE* has nodes at each end, *node_from* and *node_to*, which are defined as "**like** *node_anchor*" shown in Figure 26. In *B_TREE_EDGE* feature *node_anchor* is redefined as in Figure 27 to be of type *B_TREE_NODE* so that an edge in a b-tree will connect only b-tree nodes.

All the anchor features have the form shown in Figure 26 and are redefined in descendant classes similarly to that shown in Figure 27.

```
feature {NONE} -- Anchors (for covariant redefinitions)

    node_anchor: JJ_NODE is
            -- Anchor for features using nodes.
            -- Not to be called; just used to anchor types.
            -- Declared as a feature to avoid adding an attribute.
        do
            check
                do_not_call: False
                        -- Because give no info; simply used as anchor.
            end
        ensure then
            void_result: Result = Void
        end
```

Figure 26 – Example of an "Anchor feature"

```
feature {NONE} -- Anchors (for covariant redefinitions)

    node_anchor: JJ_B_TREE_NODE [N] is
            -- Anchor for features using nodes.
            -- Not to be called; just used to anchor types.
            -- Declared as a feature to avoid adding an attribute.
        do
            check
                do_not_call: False
                        -- Because give no info; simply used as anchor.
            end
        ensure then
            void_result: Result = Void
        end
```

Figure 27 – An "Anchor feature" redefined

7.4. Assertion Checking

Eiffel allows an object to violate invariants during features as long as it is restored on exit. Nevertheless, great effort was put into maintaining the invariants for each object at all times. Unavoidably however, there were places in the code where assertion checking had to be turned off. This was generally in features which are selectively exported. For example, in *B_TREE* during the splitting or combining of nodes the number of edges in a node and the number of items in a node could violate one or more of the invariants. As items are being shifted from one node to another there are times when the number of items falls below the minimum number of items per node as required by the b-tree's order. Also, since the edges and items must be shifted one at a time, the number of edges versus the number of items breaks the invariant that says the number of edges for non-leaf nodes is always one more than the number of items. Turning assertion checking off during development was not an option, so these "questionable" sections of code were bracketed with calls to *check_assert* from *ISE_RUNTIME* as shown in Figure 28 in the **if** statement. The second call to *check_assert* restores the assertion checking to what it was before the feature turned it off.

```
feature {B_TREE} -- Implementation (Insertion/deletion features)

    prune_value (a_node: like node_anchor; a_value: like item_anchor)
            -- Remove `a_value' from `a_node'.
        require
            node_exists: a_node /= Void
            value_exists: a_value /= Void
            node_has_value: a_node.has_item (a_value)
        local
            b: BOOLEAN
            n: like node_anchor
            v: like item_anchor
        do
            if a_node.is_leaf then
                b := {ISE_RUNTIME}.check_assert (False)
                a_node.value.prune (a_value)
                if a_node.item_count < minimum_item_count and then not a_node.is_root then
                        restore_node_count (a_node)
                end
                b := {ISE_RUNTIME}.check_assert (b)
            else
                n := a_node.successor_node (a_value)
                v := n.i_th (1)
                a_node.value.replace (a_value, v)
                prune_value (n, v)
            end
        ensure
            enough_items: a_node.is_root or else a_node.item_count >= minimum_item_count
        end
```

Figure 28 – Feature with Assertion Checking Disabled

In feature *prune_value* in Figure 28, for the nested **if** statement beginning, "**if** a_node.*item_count* < *minimum_item_count* …" to be true, by definition the invariant for a b-tree node would be violated; the number of items in the node has fallen below the minimum number required. But the point of this part of the feature is to restore that count when it is too low. The only way to check it is to turn the assertion checking off. Once *restore_node_count* has done its job, assertion checking can be turned back on.[9]

Also, during a node deletion from a graph, the edges are disconnected. One of these disconnected edges could be in a path stored in an iterator. When a feature of that path is called, and it will be in *GRAPH_ITERATOR*, the invariant from *PATH* which says the path is made of an unbroken chain of edges will be violated. So, the section of code in the *GRAPH_ITERATOR* class that checks the list of paths for validity must also be bracketed with *check_assert* as described above.

7.5. B_TREE

In order to follow the normal graph theory classification where a tree "is-a" graph, class *B_TREE* inherits from *GRAPH* indirectly through *VALUED_GRAPH [N]*. It inherits from *VALUED_GRAPH* in such a way that it makes the *value* in each node be an array instead of a single value. Specifically a *JJ_SORTABLE_FIXED_ARRAY [N]* is used as the generic parameter to *VALUED_GRAPH [N]* as shown in Figure 29.

```
class
    B_TREE [N -> COMPARABLE]
inherit
    B_TREE_NODE [N]
    VALUED_GRAPH [JJ_SORTABLE_FIXED_ARRAY [N]]
```

Figure 29 –Inheritance for B_TREE

The corresponding node and edge classes and the anchor features, shown in Figure 30, are redefined similarly to the other container and support classes.

```
class
    B_TREE_NODE [N -> COMPARABLE]
inherit
    VALUED_NODE [JJ_SORTABLE_FIXED_ARRAY [N]]
feature {NONE} -- Anchors (for covariant redefinitions)
    node_anchor: B_TREE_NODE [N]
    edge_anchor: B_TREE_EDGE [N]
    value_anchor: JJ_SORTABLE_FIXED_ARRAY [N]
    item_anchor: N
```

Figure 30 –Inheritance for B_TREE_NODE

As with all the node classes the *node_anchor* is the same type as the class. The *value* in the node is now an array as defined by feature *value_anchor*. A new feature, *item_anchor*, is defined in order to type each item in the array. This fixes the number of items in each node at creation time. The *edge_set* is still there from *NODE* but the number of edges is restricted by the invariant to always be one more than the number of items in the node.

Figure 31 shows the structure of an example b-tree of integers. Each node contains an array with *order*-1 items and a list of *order* number of edges pointing to the child nodes.[10]

[9] In hindsight, feature *prune_value* may not need to be exported to *B_TREE* so the invariants of *B_TREE* which check each node in the tree for compliance with b-tree properties would not be called; however the qualified call "a_node.*item_count*" would still check the invariants required for a *B_TREE_NODE*.

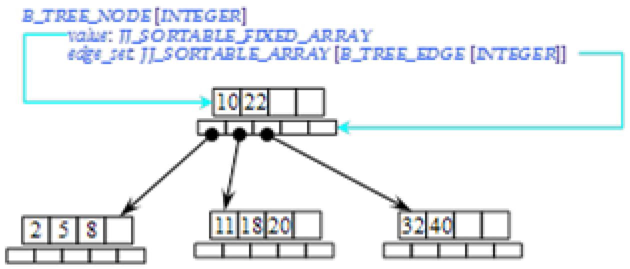
B_TREE_NODE [INTEGER]
value: JJ_SORTABLE_FIXED_ARRAY
edge_set: JJ_SORTABLE_ARRAY [B_TREE_EDGE [INTEGER]]
10 22
2 5 8
11 18 20
32 40

Figure 31 – B_TREE_NODE Structure

7.6. INDEXED_B_TREE

An *INDEXED_B_TREE* is built from a *B_TREE* whose items are of type *B_TREE_CELL*. The items of interest in the tree are contained in the cell and indexed by a *key* value. An item can be accessed with feature *item* or one of its aliases.

```
class
    INDEXED_B_TREE [G, K -> COMPARABLE]
inherit
    INDEXED_B_TREE_NODE [G, K]
    B_TREE [B_TREE_CELL [G, K]]
```

Figure 32 –Inheritance for INDEXED_B_TREE

```
class
    INDEXED_B_TREE_NODE [G, K -> COMPARABLE]
inherit
    B_TREE_NODE [B_TREE_CELL [G, K]]
```

Figure 33 –Inheritance for INDEXED_B_TREE_NODE

Error! Reference source not found. shows the structure of a single node in an indexed b-tree holding string values which are indexed by integer keys.[11]

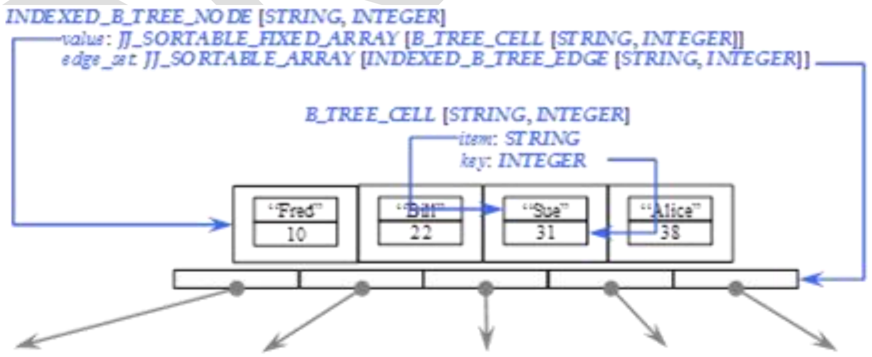
INDEXED_B_TREE_NODE [STRING, INTEGER]
value: JJ_SORTABLE_FIXED_ARRAY [B_TREE_CELL [STRING, INTEGER]]
edge_set: JJ_SORTABLE_ARRAY [INDEXED_B_TREE_EDGE [STRING, INTEGER]]
B_TREE_CELL [STRING, INTEGER]
item: STRING
key: INTEGER
"Fred"
10
"Bill"
22
"Sue"
31
"Alice"
38

Figure 34 – INDEXED_B_TREE_NODE

[10] The figure shows all the nodes containing an array of references to child nodes. In actuality, the *edge_set* of nodes in a b-tree are actually Void until the first child edge is added to the node. So, the *edge_set* of leaf nodes will always be Void.

[11] For simplicity the figure shows the strings as values stored with the keys in the cells; however, the cells really contain a reference to an object of type *STRING*. Child nodes are not shown.

7.7. Iterators

As an iterator progresses, it needs to mark node as visited, edges as traversed, and paths as explored. At first the visited_nodes feature was implemented as a list into which nodes were inserted. A node was considered visited if it was in visited_nodes. To determine if a node had been visited required a traversal of this set. For efficiency, instead of letting the GRAPH_ITERATOR keep track of visitations, this was delegated to NODE (and EDGE). When a node is visited, it is marked as was_visited_by (an_iterator). Instead of an iterator keeping track of the nodes it has visited, a node keeps track of the iterators which have visited it. Determination of node visitation should be faster this way; the number of

8. To Be Done

8.1. This File

- Fix diagrams so they don't cover a footnote.

8.2. All classes

- Remove all "is" keywords from features.
- Make void-safe and remove all "require /= Void" preconditions, etc.
- Fix all class names in comments to have braces around them.
- Make *NODE*, *EDGE*, and *PATH* (?) export modification features to *GRAPH* and *GRAPH_ITERATOR* only and add status feature *is_locked* (and corresponding setters) to *GRAPH_ITERATOR* to prevent changes to a graph during iteration. This would allow an iterator to assume the graph has not changed and therefore no need to check if a path is valid. No because one iterator may lock it but another could unlock, invalidating the assumption.

8.3. Test Classes

- Make a graphical display to show a graph and let the user dynamically create a graph.
- Make a test suite to check all traversals, moving forward and backwards while the graph is changing. Change traversal modes in the middle of a traversal to see what happens.
- Test the speed of insertions, deletion, access, and traversals on a *B_TREE* against a *SORTED_TWO_WAY_LIST*. Test *INDEXED_B_TREE* against a *HASH_TABLE*.

8.4. GRAPH

- Should I change N, E, and C (**N**ode-data, **E**dge-label, and edge-**C**ost) to V, L, and W (node-**V**alue, edge-**L**abel, and edge-**W**eight)? Or maybe use V, L, and C because the "weight" of a graph is the sum of the costs of all the edges. But "weighted graph" is the term used for graphs having costs in the edges. ???

- Ensure the graph will create nodes that have the same properties as the graph. Added nodes do not necessarily require this but when the graph makes the nodes they should. Tree nodes will have more restrictive properties.
- Allow *root_node* to not be in graph (perhaps due to a deletion by the graph) which would invalidate all paths and the iterator would be *is_before*(?) or *is_after*
- Add *minimal_spanning_tree* etc. – should this return copies or references? A minimal spanning tree of a graph returns a similar graph, not a tree, in order to keep the types correct. Minimal spanning "tree" will be a misnomer for this feature; it really returns a graph with the minimal number (or shortest) edges.
- Should ***GRAPH*** always contain itself? In *minimal_spanning_graph* I need a list of graphs as containers and putting the current graph into itself means each graph has an extra node. Think about this.
- Add *shortest_path* between two nodes? Or should this be in ***NODE***?
- Revisit *set_alphabetical* – does this make sense? Yes, sort the `nodes' and `edges' sets.
- Feature *root_node* – what happens if it is deleted from the graph? What to use for root node then?
- Add feature *distance (a_node, a_other_node)* to ***GRAPH***?

8.5. GRAPH_ITERATOR

- Finish *minimal_spanning_graph* for directed situations.
- Ensure all traversals keep track of *queue_index*? Why not use *pending_paths* in place of *queue_index* for all traversals?
- Fix *back* to call *breadth_first_back* when traversing that way.
- Fix *breadth_first_back*. Decrementing the *queue_index_position* every time is wrong!
- Fix *breadth_first_back* so it updates *visited_nodes*, and *traversed_edges*.
- Fix *visited_nodes* and *traversed_paths* to account for *is_shortest_first* traversals. The nodes and edges may be in the paths but not yet visited or traversed. A node is not "visited" until all its edges have been traversed.
- Fix comment in *shortest_first_forth*.
- Implement *shortest_first_back* and call it from *back*.
- Feature *in_order_start* – should it set the *queue_index_position*? What do I do with *queue_index_position* for the traversal methods that don't use this index? What should it be when traversal method changes to one that does?
- Implement *visit_longest* for use by *bottom_up_start* and *bottom_up_forth*.
- Implement *unreachable_nodes*.

8.6. B_TREE

- In *split* I create new nodes and then check the status of is_comparing_objects. Why not create a *new_node* and then call *copy_status* on that node? I would need a feature *new_valued_node* to accommodate a ***VALUED_GRAPH***.
- Revisit *is_in_order* – export it? Rename *is_sorted*? What happens if an item changes? Provide *sort* feature? Provide feature *is_allowing_changes*? If true then must sort else assume checked in precondition.
- Should any traversal methods other than the *set_in_order* be exported? How do you traverse a b-tree breadth-first, etc?

8.7. NODE

- Fix invariants. I think the can_embrace_multiple implications are wrong.
- Be consistent with feature naming; sometimes I use "connections" and sometimes the word "edges". Pick one.

8.8. VALUED_NODE

- Add *replace_value*? Why do this?

8.9. B_TREE_NODE

- In features *successor_node* and *predecessor_node* the post-condition, "result_is_descendent_of_Current" should be true but something is wrong, perhaps with *descendants*.

8.10. WEIGHTED_PATH

- Add (agent?) to get the cost associated with an edge, such as in ***EDGE [STREET]*** where street has feature that determines the length, in miles, of the street and that agent is passed to ***WEIGHTED_PATH*** in a feature like *set_cost_agent*. [Or can this go in PATH?]

8.11. EDGE

- Implement feature *set_cost_agent* to allow a user to provide a method for calculating the *cost*. However, this conflicts with the current concept of the generic parameter *E*; in ***WEIGHTED_EDGE [E]*** feature *cost* returns *E*. A change here would also affect feature *"<"* or *check_edges*.
- Features *node_from* and *node_to* could be given names more in line with graph theory, such as "head/tail" (refers more to the edge not the nodes) or "direct successor/predecessor" (a "successor/predecessor" refer to any node reachable from a node--what I called ancestors/descendents).

8.12. WEIGHTED_EDGE

- Feature *check_nodes* looks to see if the *cost* in each edge is an object of type ***COMPARABLE***. But an object of type ***NUMERIC*** is NOT comparable. How will this affect a graph of integers? [Answer: it will not; INTEGER inherits from COMPARABLE.] There is a similar comparison in *LABELED_EDGE* but that seems okay.

8.13.LABELED_NODE

- Fix post-conditions to *deep_adopt_labeled*.
- Fix pre- and post-condition for all features.
- Fix the features to account for *is_tree_mode*. Connecting and disconnecting edges is not so simple any more.

9. Last Check

- Ensure all the class names, feature names, code segments, and figures agree with the actual code.
- Update all field codes for the figure and footnote numbering and the cross references.

References

[1] Bertrand Meyer: Object Oriented Software Construction, 1988.

[2] Bertrand Meyer: *Eiffel*: *The Language*, *third edition*, work in progress at www.inf.ethz.ch/~meyer/ongoing/etl/, user name *Talkitover*, password *etl3*, consulted October 2007.

[3] Mark Howard, et. al.: *Type-safe Covariance: Competent Compilers Can Catch All Catcalls*, Draft — Version of 27 April 2003.

[4] Mary E. S. Loomis: Data Management and File Processing, 1983, ….fix me

[5] Reinhard Diestel: Graph Theory, Electronic Edition 2000, Springer-Verlag New York 1997, 2000.

[6] Olivier Jeger: Extending the Eiffel Library for Data Structures and Algorithms: EiffelBase, Masters Thesis, ETH Zurich, October 2004.

[7] Weisstein, Eric W. "Graph." From MathWorld—A Wolfram Web Resource. http://mathworld.wolfram.com/Graph.html.

[8] http://en.wikipedia.org/wiki/Glossary_of_graph_theory.