The Eiffel Graph Library

Jimmy J. Johnson

August 2021

Table of Contents

# 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 recongition 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.

## 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]](#footnote-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.

## 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.

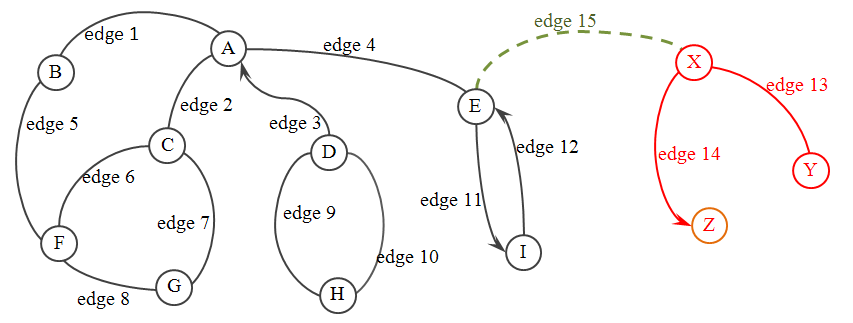


Figure -- A Simple Graph Example

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 the 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 – 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 – 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.

The code in Figure 5 shows 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\_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 – Add nodes and edges to the example graphs

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.)

*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 – Using class NODE and EDGE directly

*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 – 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 – 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 the edge between node E and node X is not in the graph and therefore 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.

## 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. 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.

# 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.

## 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.

## 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, in\_count, and out\_count. Features in\_capacity and out\_capacity, not to be confused with count, gives the number of edges the node is capable of containing 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.

## 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. 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.

## 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 referential\_integrity invariant checks that each edge contained in the node has\_connection to the current node. Also, a corresponding invariant in EDGE ensures that both nodes has\_connection to the current edge. The two invariants are show in Figure 8.

In addition the features in class GRAPH for adding nodes 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.

-- from EDGE

referential\_integrity: (is\_connected and

not node\_from.is\_unstable and not node\_to.is\_unstable) implies

(node\_from.has\_out\_edge (Current) and

node\_to.has\_in\_edge (Current))

-- from NODE

referential\_integrity: across edges as e all

(not e.item.is\_unstable and not is\_unstable) implies

e.item.has\_connection (Current) end

Figure – Referential Intergety Invariants from NODE and EDGE

*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\_can\_adopt\_child: a\_node.*can\_adopt* (a\_other\_node)

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 – Feature connect\_nodes from GRAPH

Figure 10 shows the implementation of feature adopt, from NODE, giving another example of a very heavily contracted feature.

adopt (a\_child: like node\_anchor)

-- Add a connection (EDGE) from Current to `a\_child'.

require

can\_adopt\_children: can\_adopt

local

b: BOOLEAN

do

b := mark\_unstable (true)

new\_edge.connect (Current, a\_child)

b := mark\_unstable (b)

ensure

mark\_unchanged: is\_marked\_unstable = old is\_marked\_unstable 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

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 – 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. When complete, the feature ensures that the connection between the two nodes is broken, however, the two nodes are still in the graph.

The heavily contracted features of the Graph, node, and edge classes maintain consistency between the nodes and edges in a graph. These classes are the top abstractions in the library’s hierarchy and provide the features for creating and maintaining the structure of a graph.

disconnect\_nodes (a\_node, a\_other\_node: like node\_anchor)

-- Remove any connections between the two nodes

require

has\_node: has\_node (a\_node)

has\_other\_node: has\_node (a\_other\_node)

do

-- Loop, because there could be multiple connections

from

until not a\_node.has\_node (a\_other\_node)

loop

check attached a\_node.find\_connection (a\_other\_node) as e then

-- because a\_node.has\_node (a\_other\_node) from above

e.disconnect

prune\_edge (e)

end

end

check

other\_node\_not\_connected: not a\_other\_node.has\_node (a\_node)

-- because of node/edge symetry and all edges between

-- the two nodes were disconnected.

end

notify\_dirty

ensure

not\_connected: not a\_node.has\_node (a\_other\_node)

other\_not\_connected: not a\_other\_node.has\_node (a\_node)

still\_in\_graph: has\_node (a\_node)

other\_still\_in\_graph: has\_node (a\_other\_node)

end

Figure – Feature is\_extendable\_node from GRAPH

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

# Beyond Structure

Given a consistent graph structure

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.

## GRAPH\_ITERATOR

Create an iterator for a graph 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 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.[[2]](#footnote-2)

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 – Comment for set\_breadth\_first from GRAPH\_ITERATOR

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 – Comment for set\_depth\_first from GRAPH\_ITERATOR

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 – 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 – 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 – Traversal Order for set\_bottom\_up and set\_leaf\_first

## 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.

## 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.

## 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.[[3]](#footnote-3) Figure 19 shows a simple graph and paths explored during a breadth-first traversal. Remember that a 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?



Figure – Node Deletion Affects Cursor Movement

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 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.[[4]](#footnote-4)

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.

## 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 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” 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.



Figure – Changing Traversal Method

## 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.[[5]](#footnote-5)

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.

## 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.

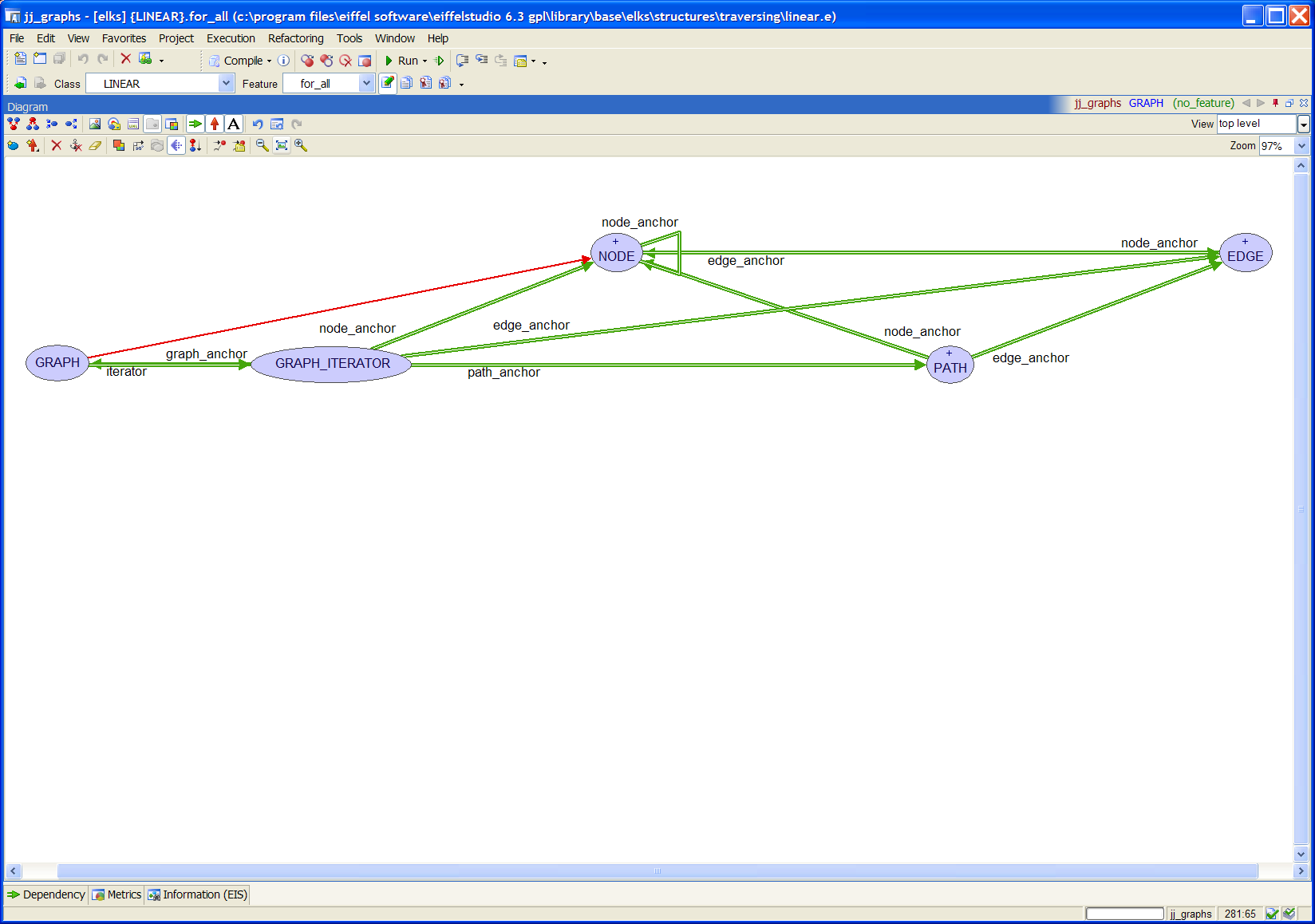


Figure – Upper-level Classes

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[[6]](#footnote-6).

# 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.

## 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.

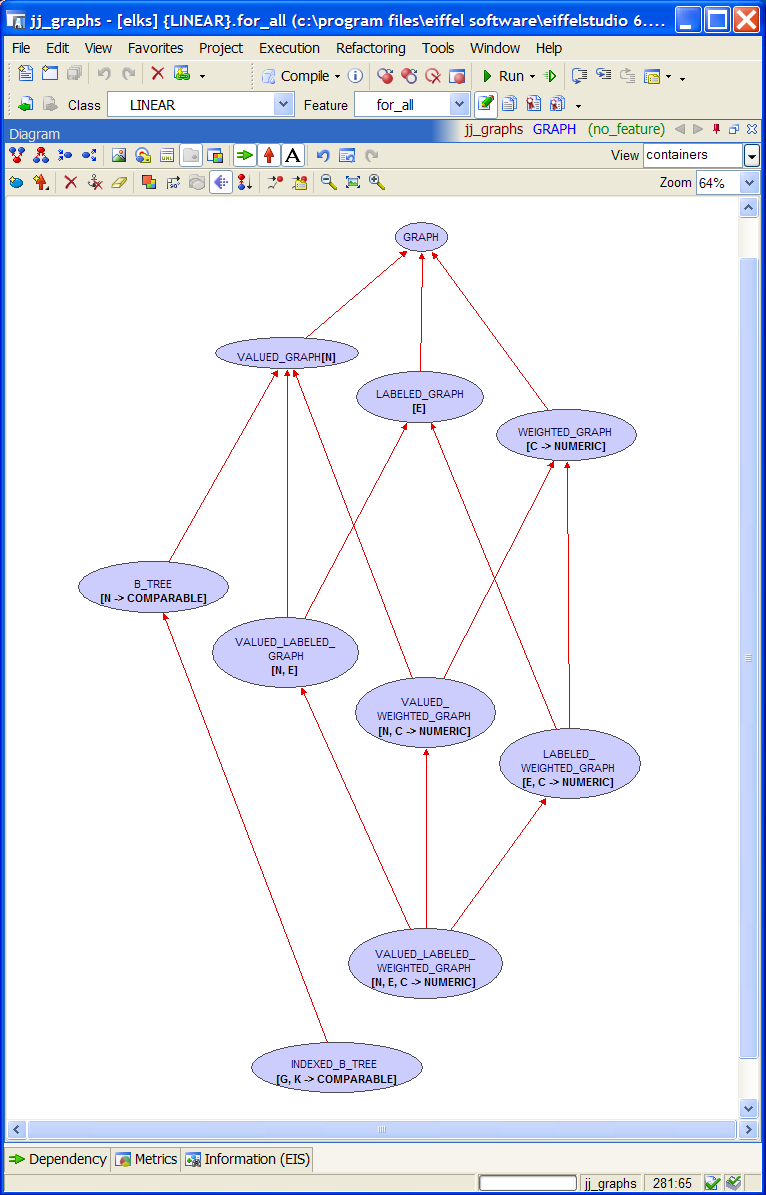


Figure – Containers

Class B\_TREE [N ->COMPARABLE] models a b-tree whose values of type N are stored in order. The 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 – 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!!

## 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].

## Covariance and Catcalls

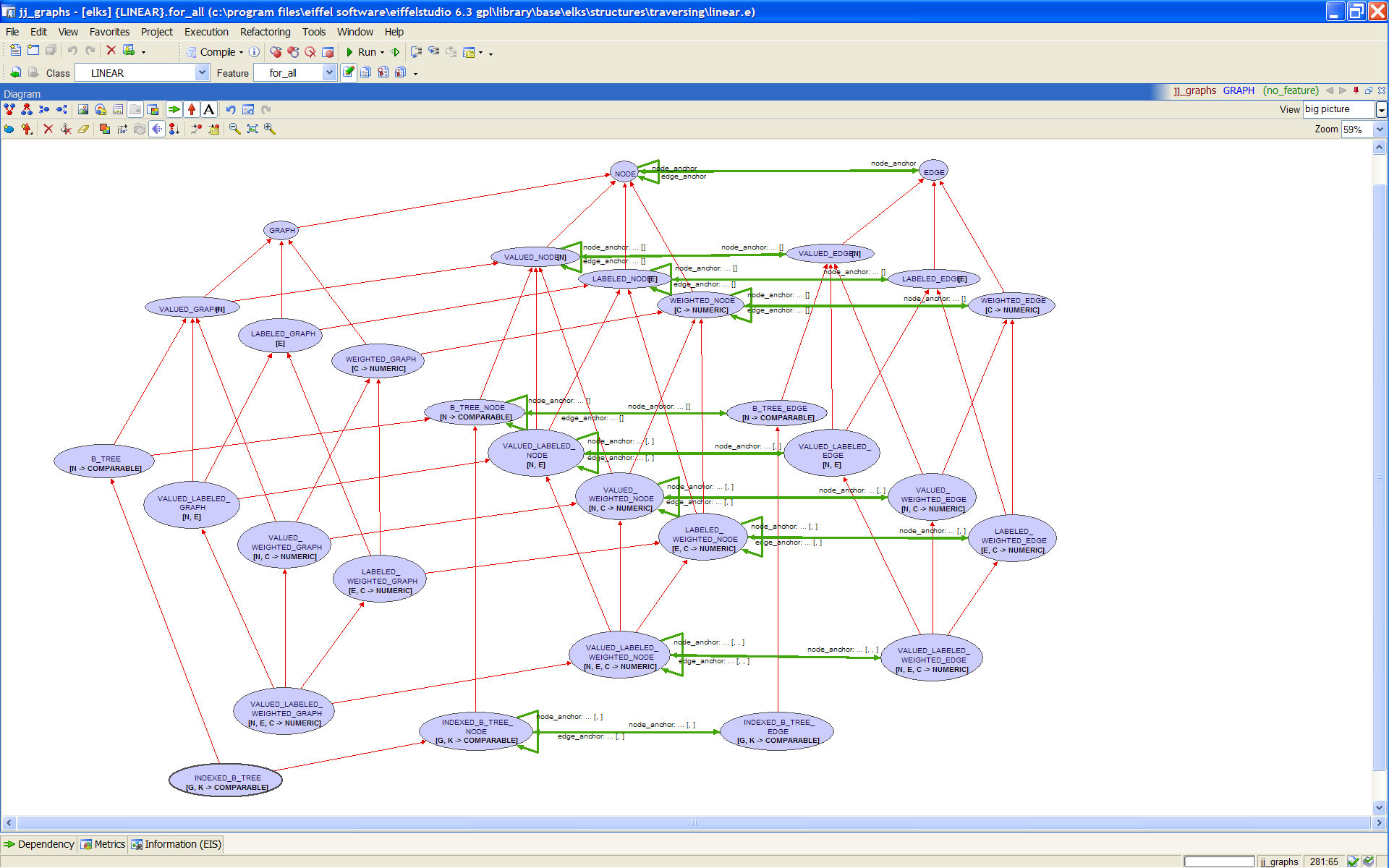


Figure – The Eiffel Graph Library

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 – 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.

# 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.

## VALUED\_GRAPH vrs. LINKED\_LIST

## B\_TREE vrs. SORTED\_TWO\_WAY\_LIST

## INDEXED\_B\_TREE vrs. HASH\_TABLE

# 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 …

# Notes on Implementation

## 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……

## 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.

## 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 – 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 – An “Anchor feature” redefined

## 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 – 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.[[7]](#footnote-7)

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.

## 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 –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 –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.[[8]](#footnote-8)



Figure – B\_TREE\_NODE Structure

## 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 –Inheritance for INDEXED\_B\_TREE

class

INDEXED\_B\_TREE\_NODE [G, K -> COMPARABLE]

inherit

B\_TREE\_NODE [B\_TREE\_CELL [G, K]]

Figure –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.[[9]](#footnote-9)



Figure – INDEXED\_B\_TREE\_NODE

## 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

# To Be Done

## This File

* Fix diagrams so they don’t cover a footnote.

## 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.

## 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.

## 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?

## 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.

## 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?

## 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.

## VALUED\_NODE

* Add replace\_value? Why do this?

## 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.

## 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?]

## 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).

## 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.

## 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.

# 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.

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. [↑](#footnote-ref-1)
2. 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. [↑](#footnote-ref-2)
3. 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. [↑](#footnote-ref-3)
4. 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. [↑](#footnote-ref-4)
5. 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. [↑](#footnote-ref-5)
6. 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. [↑](#footnote-ref-6)
7. 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. [↑](#footnote-ref-7)
8. 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. [↑](#footnote-ref-8)
9. 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. [↑](#footnote-ref-9)