© The Author 2005. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oupjournals.org
doi:10.1093/comjnl/bxh000

Using Coupling-based Weights for the Class Integration and Test Order Problem

Aynur Abdurazik, Jeff Offutt

Information and Software Engineering, George Mason University, Fairfax, VA 22030, USA
Email: aabduraz@gmu.edu,offutt@gmu.edu

During component-based and object-oriented software development, software classes exhibit relationships that complicate integration, including method calls, inheritance, and aggregation. Classes are integrated and tested in specific orders, where each class is added and tested one by one to see if it integrates successfully. A difficulty arises when cyclic dependencies exist – the functionality that is used by the first class to be tested must be mimicked by creating "stubs" (sometimes called "mock objects"), an expensive and error-prone operation. This problem is generally called the class integration and test order (CITO) problem, and solutions must be fully automated for integration and testing to proceed smoothly and efficiently. This paper describes new techniques and algorithms to solve the CITO problem. New results include improved edge weights to more precisely model the cost of stubbing, and the use of node weights, which allows more information to be used. These weights are derived from quantitative measures of couplings between the integrated and stubbed classes. Also, a new algorithm for computing the integration and test orders is presented. The technique is compared with an existing approach and found to be cheaper, get the same results when using edge weights exclusively, and yield better results when using node weights.

Received 30 August 2006; revised 00 Month 2006

1. INTRODUCTION

A common problem in inter-class integration testing of object-oriented software is to determine the order in which classes are integrated and tested [1]. When one class requires another to be available before it can be executed, we define this kind of relationship to be a *dependency*. These two classes can be characterized as *server* and *client* classes. The client class is being compiled or executed and the server class must be present. This dependency has a direction, and is based on one or more object-oriented relationships.

When there is no cycle in the dependency of classes or subsystems, the *class integration and test order* (CITO) problem can be solved by a simple reverse topological ordering of classes based on their dependencies. However, dependency cycles are common in real-world systems and when present, topological sorting is not possible [1].

To solve the CITO problem in the presence of cycles, the cycles must be broken. The effect of breaking a dependency cycle is that a *stub* must be created for the class that is no longer present, thus increasing the cost of integration testing. A *stub* is an incomplete class that provides the signatures necessary for full compilation and integration, but does not implement the full functionality (stubs are sometimes called *mock objects*). Our goal is to find an optimal order that minimizes the stubbing effort. Stubbing effort has many factors to consider, and, therefore, cannot be completely measured or estimated [2]. These factors include how complete the stub needs to be and whether it must perform a computation to be useful. For example, it may be possible for the stub to return random or fixed values, which is very easy to implement, or the stub may need to return more specific values, which will require more algorithmic complexity in the stub. Some class stubs may need only one method, whereas others may need more. More details can be found in Briand et al.'s paper [2]. It has also been suggested that creating stubs can be error prone and costly [3].

Coupling measurement captures class relationships. The main goal of this study is to use coupling measurement to estimate stubbing effort and develop

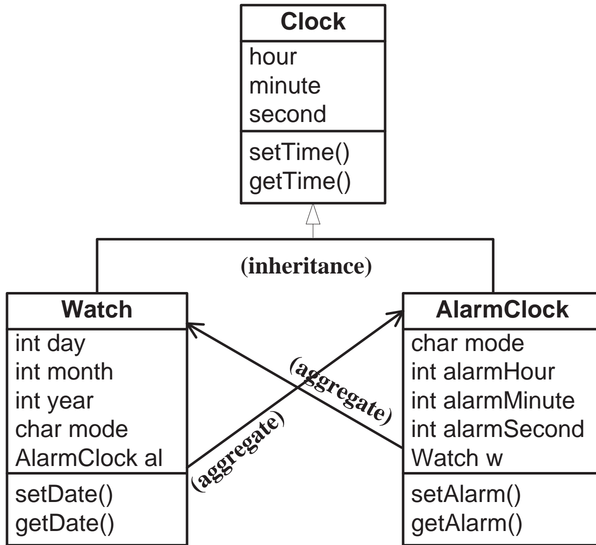

FIGURE 1. Simple Example 3-class Package With a Dependency Cycle

an efficient technique for finding an *optimal integration order*.

Figure 1 shows a simple example of three classes that have a dependency cycle. Class **Watch** and class **AlarmClock** inherit from class **Clock**. **Watch** aggregates an **AlarmClock** so it can supply an alarm in the **Watch**. **AlarmClock** also aggregates a **Watch** so it can supply a date in the **AlarmClock**. Although most design experts would not advocate this design, such aggregations represent common shortcuts that programmers make. Unfortunately, this forms a dependency cycle, where **Watch** and **AlarmClock** depend on each other. Most cycles involve more than two classes, and thus are less obvious.

The remainder of the paper discusses existing solutions, introduces our model, then presents results from a case study taken from Briand et al.'s paper [2]. Section 2 of this paper summarizes existing approaches to the class integration and test order (CITO) problem and Section 3 describes our model and algorithms. The algorithms are explained in detail with a running example. Section 4 presents a case study that uses the same system as used by Briand et al. [2] and Section 5 presents conclusions and discusses ideas for future work.

2. SUMMARY OF EXISTING SOLUTIONS

The class integration and test order problem has been addressed by several researchers and several solutions have been proposed. The solutions can be categorized into *graph-based* and *genetic algorithm-based* approaches. This section summarizes existing solutions and discusses their advantages and disadvantages.

In graph-based approaches, classes and their relationships in software are modeled as object relation diagrams (ORD) or test dependency graphs (TDG). An ORD or TDG is a directed graph $G(V, E)$ where V is a set of nodes representing classes and E is a set of edges representing the relationships among classes. The class integration and test order problem is to find an ordering of nodes in the graph so that the classes can be integrated and tested with minimum effort.

In most papers [1, 4, 5, 6], the testing effort is estimated by counting the number of test stubs that need to be created during integration testing. This method assumes that all stubs are equally difficult to write. Only one recent paper has tried to consider test stub complexity when estimating the testing effort [7].

In the genetic algorithm-based approach [2], inter-class coupling measurements and genetic algorithms are used in combination to assess the complexity of test stubs and to minimize complex cost functions.

Kung et al. [1] were the first researchers to address the class test order problem and they showed that, when classes do not have any dependency cycles, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph–a well known graph theory problem. In the presence of dependency cycles, they proposed a strategy of identifying strongly connected components (SCCs) and removing associations until no cycles remain. When there is more than one candidate for cycle breaking, Kung et al.'s approach chooses randomly. They mention that a possible solution would involve the use of the complexity of the associations involved in cycles.

Tai and Daniels proposed a number of properties for inter-class test ordering [5]. They assumed that aggregation and inheritance relations do not form cycles, but association relations may. Tai and Daniels defined a *major* and a *minor* level for classes, and sorted classes according to these levels. First, classes are assigned major level numbers according to the inheritance and aggregation relationships only. Then, within each major level, minor-level numbers are assigned based on the association relationships only. In this case, first, strongly connected components (SCCs) in a major level are identified, then each edge in a SCC is assigned a value, called *weight (e)*, which is defined as the sum of the number of incoming dependencies of the origin node of e and the number of outgoing dependencies of the target node of e. Edges with higher values are selected to break cycles. The hypothesis is that removing edges with higher values will break more cycles. However, Briand et al. [4] showed this hypothesis is not always true. Another problem is that their algorithm may break an association edge that crosses major levels but is not involved in any cycles [4].

Le Traon et al. assigned weights to each node in the ORD, then removed the incoming edges of the node with maximum weight [6]. This process is repeated until no cycle remains in the ORD. To assign weights, they first used Tarjan's algorithm to identify strongly connected components. In each SCC, edges are

partitioned into four classes: (1) *tree edges* lead from a node to an unvisited node, (2) *forward edges* are non tree-edges that go from a node to a descendent, (3) *frond edges* go from a node to an ancestor, and (4) *cross edges* are the remaining edges. The weight of a node is the sum of the number of incoming and outgoing frond edges.

Le Traon et al.'s approach is non-deterministic in two ways. First, different sets of edges can be labeled as *frond edges* depending on the different starting node. Second, the approach arbitrarily chooses a node when two or more nodes have the same weight. Thus, different runs of the algorithm result in different outcomes.

Briand et al. [8, 4] proposed a graph-based strategy for ordering classes for testing that combines Tai and Daniels and Le Traon et al.'s approaches. They first used Tarjan's algorithm to identify strongly connected components (SCCs). Next, weights are assigned to *association* edges in the SCCs. The weight of an edge is the *estimated* number of cycles that the edge may be involved in. Let $G_i(V_i, E_i)$ be a SCC of graph $G(V, E)$ and $v_1, v_2 \in V_i, e \in E_i$, and $e = v_1 \rightarrow v_2$. The estimated weight of edge e is $weight(e) = (v_1)_{in} \times (v_2)_{out}$, where $(v_1)_{in}$ is the number of incoming dependencies of node v_1 and $(v_2)_{out}$ is the number of outgoing dependencies of node v_2. Then, the edge with the highest weight value is removed. These steps are repeated until no SCC remains.

Briand et al.'s approach has the advantage over Le Traon's approach of not breaking inheritance and aggregation edges and also the weight computation for edges is more precise than Tai and Daniels' approach.

Subsequently, Briand et al. [2] used a genetic algorithm and coupling metric to try to break cycles by removing edges that will reduce the complexity of stub construction. A *genetic algorithm* is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. It is a search algorithm that locates optimal binary strings by processing an initially random population of strings using artificial mutation, crossover and selection operators, in an analogy with the process of natural selection [9]. Briand et al. conclude that composition and inheritance relationships should never be removed since, according to their heuristic, removal of these edges would likely lead to complex stubs. The complexity of stub construction for parent classes is induced by the likely construction of stubs for most of the inherited member functions [8]; moreover, inherited member functions must be tested in the new context of the derived class rather than the context of the parent class [10]. Their experiment showed that genetic algorithms can be used to obtain optimal results by using more complex cost functions and perform as well as graph-based algorithms under similar conditions.

Malloy et al. developed a *Class Ordering System* that is driven by a parameterized cost model [7]. They used a strategy similar to Briand et al.'s graph-based approach [4]. They defined six types of edges, *association*, *composition*, *dependency*, *inheritance*, *owned element*, and *polymorphic*. These edges are assigned weights of (2, 2, 20, 5, 20, 20) based on their estimation of the cost of stub construction for untested classes based on heuristics. For an ORD $G = (V, E)$, where V is a set of nodes representing classes and E is a set edges representing relationships among classes, their cost model $C = \langle W, f(e), w(m_{x,y}) \rangle$ is a 3-tuple where W is a set of weight assignments and $f(e)$ and $w(m_{x,y})$ are weight functions. When there is a cycle, the edge with the smallest weight is removed from the strongly connected component. When there is no cycle, the reverse topological sort of the nodes in the ORD is the order for integration test.

To summarize, the existing graph-based approaches use high level, course grained, estimates of test stub complexity. The GA approach must be run many times, greatly complicating the process. The algorithms described in this paper only run once and use more information to provide a more precise estimation of test stub complexity.

3. A NEW MODEL AND ALGORITHMS

This section introduces a new graph-based solution for the CITO problem. Our approach improves on other graph-based approaches in three ways. First, we model classes and their relationships with weights on **both nodes and edges**. Second, weights of nodes and edges are based on a **quantitative measure** of coupling. Last, we use algorithms that **incorporate edge and node weights** as well as the number of cycles in cycle breaking.

3.1. Modeling Class Integration & Test Order

As said in Section 2, dependencies among classes are usually modeled in graphs. The CITO problem then becomes finding an acyclic graph with minimum cost. The cost is usually modeled as the number of test stubs to be generated during the testing activities. This section uses a different abstraction for test dependencies among classes and a different cost model for the testing effort. We model the test dependencies among classes using a *Weighted Object Relation Diagram (WORD)*, and model the testing effort by computing test stub complexities using coupling information.

To develop testing and maintenance predictive models, we developed couplings based on explicit and implicit object-oriented class relationships. The UML diagrams encode relationships and by studying the use of UML, we were able to identify nine different types

of couplings that have complementary coupling connections in software implementations. They are *Association Coupling*, *Aggregation Coupling*, *Composition Coupling*, *Usage Dependency Call Coupling*, *Global Coupling*, *Inheritance Coupling*, *Interface Realization Coupling*, *External Coupling*, and *Exception Coupling*.

For each coupling type, coupling measures are defined to measure the dependencies between a server and client class in terms of four attributes: (1) the number of distinct variables used, (2) the number of distinct methods called (including constructors), (3) the number of parameters sent, and (4) the number of return value types. The coupling measures also include a coupling type indicator. For convenience of expression, these four measures are aggregated into one term using a "dot notation." The intent of the dot notation is to indicate that the four measurements are independent but related. The following equation represents a *coupling measure (CM)* for couplings between two classes c_i and c_j:

$$CM(c_i, c_j) = C.V_d.M_d.R_d.P_d, \quad (1)$$

where c_i and c_j represent two classes that are coupled together, and:

$$C = \begin{cases} 5 & \text{when coupling is based on inheritance} \\ & \text{or composition} \\ 1 & \text{for other coupling types} \end{cases}$$

V_d, the *number of distinct vars*, represents the number of distinct public vars of c_j that are directly used by c_i. M_d, the *number of distinct methods*, represents the number of distinct public methods of c_j that are called by c_i. R_d, the *number of distinct return types*, represents the number of distinct return types that appear in M_d. P_d, the *number of distinct parameters*, represents the number of distinct parameters that appear in M_d.

Equation 1 assigns different values to C to distinguish different coupling types in our measure so that they can be analyzed and used methodically when needed. As part of this research, the coupling measures are used to estimate how difficult it would be to create a stub for each class, called the *test stub complexity*.

We define two kinds of stub complexity. If a client class c_i depends on a server class c_j to function, we can quantify this dependency by identifying the scope of B used by A, as measured by coupling. We define this to be a *specific test stub complexity* of B to A. However, there can be other client classes that depend on B, and some uses of B can overlap uses of client classes. We define the sum of dependencies/usages from all other client classes to B as the *total stub complexity* of server class B. When we compute the total stub complexity of a class, we take into account the overlapping possibility of specific stubs. Thus, a total stub complexity of a class takes a value between the maximum and sum of several specific stubs complexity.

We use the *specific test stub complexity* and *total stub complexity* to assign weights to edges and nodes in our WORD. In a WORD, nodes represent classes and edges represent test dependencies among classes. Both nodes and edges are weighted. The node weight represents the *total stub complexity* of a class, and the edge weight represents the *specific stub complexity* of a server class to the client class that is connected by the edge. The graph can be acyclic or cyclic. If the graph is acyclic, then we can carry out integration testing in the reverse topological order of the graph. If the graph is cyclic, then we have to first break cycles. This forces us to create test stubs for the edges that were broken or nodes that were removed. We model the testing effort as the total complexity of stubs that are introduced during integration testing. The goal is to make the graph acyclic by removing certain edges and/or nodes, and the total weight of removed edges and/or nodes has to be minimum.

The model for the class integration and test order problem is defined as follows:

Let $G(V, E)$ be a node- and edge-weighted directed graph that models classes or components and their relationships. In the graph, nodes represent classes or components, and edges represent test dependencies among classes. The edge weights represent *specific stub complexities* and node weights represent *total stub complexities*. Our problem is to determine the nodes and edges with minimum total weight to remove so that there are no cycles in G.

3.1.1. Measuring Stub Complexity

We use coupling measures to assign weights to edges and nodes in the weighted object relation diagram (WORD). After all couplings are measured in the form of equation 1, coupling measures between the source and target node classes are aggregated into one measure, cm_{e_i}, which measures the coupling for edge e_i. The edge e_i represents a coupling between two classes, and the coupling measure is summed over the nine coupling types. This measure is defined as:

$$\begin{aligned} cm_{e_i} &= \{\sum_{k=1}^{9} CM_k(v_m, v_n) \\ &\quad \mid v_m, v_n \in V, e_i = v_m \rightarrow v_n, e_i \in E\} \\ &= max(C).\sum_{k=1}^{9} V_{d_k}.\sum_{k=1}^{9} M_{d_k}.\sum_{k=1}^{9} R_{d_k}.\sum_{k=1}^{9} P_{d_k} \quad (2) \end{aligned}$$

This measure will be used to compute the weight of an edge and represents a *specific test stub complexity* of a class.

The coupling measures on edges are then further aggregated to nodes. A coupling measure on a node is computed from the coupling measures of the incoming edges to the node. In the following formula, g is the number of incoming edges to the node. The

formula computes an interval bounded by the maximum coupling of all the incoming edges and the sum of the couplings from all the incoming edges ($[max, sum]$).

$$cm_{v_i} = \left\{ \left[max(cm_{e_{1,i}}, cm_{e_{2,i}}, ...cm_{e_{g,i}}), \sum_{l=1}^{g} cm_{e_{l,i}} \right] \mid v_i \in V, e_{l,i} = v_l \rightarrow v_i, e_{l,i} \in E \right\} \quad (3)$$

where $cm_{e_{1,i}}, cm_{e_{2,i}}, ...cm_{e_{g,i}}$ are coupling measures on the incoming edges of node v_i and the summation of coupling measures are the same as in equation 2. Equation cm_{v_i} takes a value between the maximum and sum of coupling measures on the incoming edges of node v_i. This measure will be used to compute the weight of a node and represents the *total test stub complexity* of a class.

Our rationale for introducing weights for nodes is that specific stubs for a class may overlap. It is possible that certain methods or variables of a server class can be used by a number of clients in the same way. In this case, creating one stub for the server class can satisfy the needs of several clients.

We use Briand et al.'s [2] method for estimating stubbing complexity from the coupling measures of edges and nodes. For a measure $Cplx()$, a complexity measure $\overline{Cplx()}$ is normalized as:

$$\overline{Cplx(i,j)} = Cplx(i,j) \ / \ (Cplx_{max} - Cplx_{min}) \quad (4)$$

where $Cplx(i,j)$ represents a complexity information matrix, $Cplx_{min} = Min\{Cplx(i,j), i,j = 1,2,...\}$ and $Cplx_{max} = Max\{Cplx(i,j), i,j = 1,2,...\}$. They use two coupling measures A() and M(), the number of locally defined variables and the number of methods, to compute overall stubbing complexity:

$$SCplx(i,j) = (W_A.\overline{A}(i,j)^2 + W_M.\overline{M}(i,j)^2)^{1/2} \quad (5)$$

where W_A and W_M are weights and $W_A + W_M = 1$. Thus, for a given test order o, with d dependencies to be broken, an overall stubbing complexity for the order o is computed as

$$OCplx(o) = \sum_{k=1}^{d} SCplx(k) \quad (6)$$

The principle of not breaking inheritance and composition edges was ensured by constraints in Briand et al.'s work. This paper takes a different approach, specifically, assigning higher values for inheritance and composition to the variable C in equation 1.

As shown in equation 1, our coupling measures use the number of parameters and the number of return value types in addition to the number of variables and the number of methods. We use the same normalization method as Briand et al., and include the additional coupling measures in the stubbing complexity estimation.

Using aggregated coupling measures on edges and nodes, a stubbing complexity is estimated as follows:

$$SCplx(i,j) = C + (W_V \times \overline{V}(i,j)^2 + W_M \times \overline{M}(i,j)^2 + W_R \times \overline{R}(i,j)^2 + W_P \times \overline{P}(i,j)^2)^{1/2} \quad (7)$$

where C is the first variable from equations 2 and 3, W_V, W_M, W_R, and W_P are weights and $W_V + W_M + W_R + W_P = 1$. The $\overline{V}(i,j)$, $\overline{M}(i,j)$, $\overline{R}(i,j)$, and $\overline{P}(i,j)$ values are computed from equation 4 using values from equations 2 and 3.

Our objective is to find an optimal integration and test order o by determining a set of k nodes and/or l edges to be removed to make the $WORD$ acyclic such that the sum of the stubbing complexities for these nodes and edges is minimum:

$$OCplx(o) = \sum_{i=1}^{k} SCplx_{node}(i) + \sum_{j=1}^{l} SCplx_{edge}(j) \quad (8)$$

3.2. Heuristic Algorithm for Breaking Cycles

This section presents **three** general algorithms for making a cyclic graph acyclic using simple weight assignments on edges, nodes, and both. The first algorithm uses edge weights, the second uses node weights and the third uses both.

3.2.1. Heuristic Algorithm for Breaking Cycles Using Edge Weights

A natural heuristic is to first find all strongly connected components in the $WORD$, then work on each SCC separately. The first algorithm follows this approach to find cycle-weights for each edge. The $cycle - weight$ *ratio* (CWR) for an edge is defined as the number of cycles the edge appears in (NC) divided by the weight of that edge Wt. The CWR is used to successively eliminate edges from the graph until all cycles are broken. The *cost* of breaking these cycles is then the sum of the edge weights of all the edges that were eliminated. Algorithm 1 summarizes this process and is illustrated through the example in Figure 2.

Figure 2 is based on an example in Briand et al.'s paper [2]. The edges represent general dependencies between two classes, not UML-specific relationships, with edge labels representing the specific stub complexity. Step 1 in Algorithm 1 finds one SCC in Figure 2, $\{E, A, C, H, D, B, F\}$. Steps 2 and 3 find the following 11 cycles in the SCC:

(1) $E \rightarrow F \rightarrow E$
(2) $E \rightarrow A \rightarrow C \rightarrow E$
(3) $E \rightarrow F \rightarrow D \rightarrow A \rightarrow C \rightarrow E$
(4) $E \rightarrow F \rightarrow D \rightarrow H \rightarrow C \rightarrow E$

Algorithm 1 Eliminating Cycles in WORD (V,E)

1: Find all SCCs in WORD
2: **for** (each $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$) **do**
3: find all cycles CYCLES (totalCycles)
4: **for** (each $e \in E_{scc_i}$) **do**
5: find the number of cycles that use e $(cardinal\{cycles - through - e\})$
6: compute the cycle-weight ratio
7: **end for**
8: **while** (totalCycles != 0) **do**
9: order all *edges* in descending order of their cycle-weight ratio
10: remove *edge* with highest cycle-weight ratio
11: totalCycle = totalCycle - number of cycles broken
12: update the number of cycles that use e $(cardinal\{cycles - through - e\})$ in the remaining edge set
13: recompute the cycle-weight ratio for the remaining edges
14: **end while**
15: **end for**

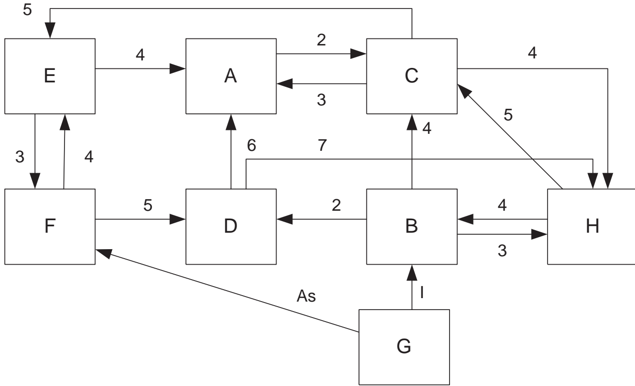

FIGURE 2. Example Weighted Object Relation Diagram (WORD)

(5) $E \to F \to D \to H \to B \to C \to E$
(6) $A \to C \to H \to B \to D \to A$
(7) $A \to C \to A$
(8) $C \to H \to C$
(9) $C \to H \to B \to C$
(10) $H \to B \to H$
(11) $H \to B \to D \to H$

Table 1 shows the results from steps 4 through 7 of Algorithm 1. After the initial computation of CWR values for edges, the algorithm works in the following steps:

A. Choose an edge from table 1 with maximum cycle-weight ratio and remove that edge from the *WORD*. At this point, the edge with the maximum cycle-weight ratio is $A \to C$ with a ratio of 2. Removing edge $A \to C$ breaks four cycles; 2, 3, 6, and 7; leaving seven.

TABLE 1. Cycle-weight Ratio for Edges in SCC $\{E, A, C, H, B, D, F\}$. $Wt.$ is the weight of the edge, NC is the number of cycles the edge appears in, and CWR is the cycle-weight ratio.

No.	Edge	Wt.	Cycles Invol-ved	NC	CWR
1	$\mathbf{A \to C}$	**2**	**{2, 3, 6, 7}**	**4**	**2**
2	$B \to C$	4	{5, 9}	2	0.5
3	$B \to D$	2	{6, 11}	2	1
4	$B \to H$	3	{10}	1	0.33
5	$C \to E$	5	{2, 3, 4, 5}	4	0.8
6	$C \to A$	3	{7}	1	0.33
7	$C \to H$	4	{6, 8, 9}	3	0.75
8	$D \to A$	6	{3, 6}	2	0.33
9	$D \to H$	7	{4, 5, 11}	3	0.43
10	$E \to A$	4	{2}	1	0.25
11	$E \to F$	3	{1, 3, 4, 5}	4	1.33
12	$F \to D$	5	{3, 4, 5}	3	0.6
13	$F \to E$	4	{1}	1	0.25
14	$H \to B$	4	{5, 6, 9, 10, 11}	5	1.25
15	$H \to C$	5	{4, 8}	2	0.4

B. Re-compute the cycle-weight ratio for the remaining edges. The result is shown in table 2. There are two edges that have same maximum cycle-weight ratio, 14 and 11 with ratios of 1. Our rule in this situation is to choose the edge that is involved in larger number of cycles. This is edge 14, $H \to B$. Removing $H \to B$ breaks four cycles; 5, 9, 10, and 11; leaving three.

C. Re-compute the cycle-weight ratio for remaining of edges in table 2. For space reasons, the result is not shown, but the edge with maximum cycle-weight ratio is now edge 11, $E \to F$. Removing edge $E \to F$ breaks two cycles, 1 and 4, leaving only cycle 8.

D. Re-compute the cycle-weight ratio for the remaining edges, and at this point the edge with maximum cycle-weight ratio is edge 7, $C \to H$. Removing $C \to H$ breaks cycle 8, and makes the *WORD* acyclic. Thus, we break all 11 cycles by removing four edges, $A \to C$, $H \to B$, $E \to F$, and $C \to H$. The total cost is 2+4+3+4 = 13.

3.2.2. Applying Algorithm 1 to A Special Case

A key difference between this research and previous research is the modeling of the cost of stubbing as edge weights. If we assign a weight of 1 to each edge, then our model is equivalent to the previous models. In addition, previous researchers modeled node weights as the sum of all incoming edges, which corresponds to our pessimistic approach. To facilitate comparison, we assign all edges in the graph in Figure 2 the weight 1, to see if our algorithm gets the same results as Briand's [4, 2].

TABLE 2. Cycle-weight Ratio for Edges in *SCC* $\{E, A, C, H, B, D, F\} - \{A \to C\}$

No.	Edge	Wt.	Cycles Involved	NC	CWR
2	$B \to C$	4	{5, 9}	2	0.5
3	$B \to D$	2	{11}	1	0.5
4	$B \to H$	3	{10}	1	0.33
5	$C \to E$	5	{4, 5}	2	0.4
6	$C \to A$	3	{ }	0	0
7	$C \to H$	4	{8, 9}	2	0.5
8	$D \to A$	6	{ }	0	0
9	$D \to H$	7	{4, 5, 11}	3	0.43
10	$E \to A$	4	{ }	0	0
11	**E → F**	**3**	**{1,4,5}**	**3**	**1**
12	$F \to D$	5	{4, 5}	2	0.4
13	$F \to E$	4	{1}	1	0.25
14	**H → B**	**4**	**{5, 9, 10, 11}**	**4**	**1**
15	$H \to C$	5	{4, 8}	2	0.4

TABLE 3. Cycle-weight Ratio for Edges in *SCC* $\{E, A, C, H, B, D, F\}$ **- All Edges Have the Same Weight**

No.	Edge	Wt.	Cycles Involved	NC	CWR
1	$A \to C$	1	{2, 3, 6, 7}	4	4
2	$B \to C$	1	{5, 9}	2	2
3	$B \to D$	1	{6, 11}	2	2
4	$B \to H$	1	{10}	1	1
5	$C \to E$	1	{2, 3, 4, 5}	4	4
6	$C \to A$	1	{7}	1	1
7	$C \to H$	1	{6, 8, 9}	3	3
8	$D \to A$	1	{3, 6}	2	2
9	$D \to H$	1	{4, 5, 11}	3	3
10	$E \to A$	1	{2}	1	1
11	$E \to F$	1	{1, 3, 4, 5}	4	4
12	$F \to D$	1	{3, 4, 5}	3	3
13	$F \to E$	1	{1}	1	1
14	**H → B**	**1**	**{5, 6, 9, 10, 11}**	**5**	**5**
15	$H \to C$	1	{4, 8}	2	2

Table 3 shows initial CWR values for edges. We briefly describe the process: first, edge $H \to B$ is chosen to be removed, breaking five cycles. The recomputation of CWR values for the remaining edges are not shown, but the next edge to remove is $E \to F$, breaking three cycles. Next, edge $A \to C$ is chosen, breaking two cycles. There is one cycle left, 8, and we can break it by either removing $H \to C$ or $C \to H$. Here we can apply the heuristic of not breaking an *Aggregation* relationship and choose $C \to H$ to remove. Thus, we removed four edges in total: $H \to B$, $E \to F$, $A \to C$, and $C \to H$. This result is the same as the result from Briand et al.'s graph-based research, although the algorithm is quite a bit cheaper and the edges were removed in a different order.

3.2.3. Heuristic Algorithm for Breaking Cycles Using Node Weights

Algorithm 1 assumed weights were assigned to edges. This subsection presents an algorithm for breaking cycles using weight assignments on nodes. Algorithm 2 is illustrated through the following examples.

Algorithm 2 Eliminating Cycles in WORD(V,E) Using Node Weights

1: Find all SCCs in WORD
2: **for** (each $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$) **do**
3: find all cycles CYCLES (totalCycles)
4: **for** (each $v \in V_{scc_i}$) **do**
5: find the number of cycles that use v ($cardinal\{cycles - through - v\}$)
6: compute the cycle-weight ratio
7: **end for**
8: **while** (totalCycles != 0) **do**
9: order all *nodes* in descending order of their cycle-weight ratio
10: remove *node* with highest cycle-weight ratio
11: totalCycle = totalCycle - number of cycles broken
12: update the number of cycles that use v ($cardinal\{cycles - through - v\}$) in the remaining node set
13: recompute the cycle-weight ratio for the remaining nodes
14: **end while**
15: **end for**

The node weights model the amount of effort needed to create a stub for that class, that will be used by all classes that use it (*user classes*). In a pessimistic approach, each user class will need completely different stub functionality, so each user class needs a completely independent stub. For example, one user class may call methods $M1()$ and $M2()$, and another may call methods $M3()$ and $M4()$. This situation is modeled by case 1, where the node weight is the **sum** of incoming edge weights. In an optimistic approach, all the user classes will need the exact same stub functionality, so one stub can satisfy all user classes. This situation is modeled by case 2, where the node weight is the **maximum** of incoming edge weights. In most situations, the reality is probably in between. So case 3 models the situation where the node weight is between the maximum and the sum of the incoming edge weights. Which choice to make depends on domain knowledge and probably needs to be made by the tester. Figures 3 and 4 show node weights for cases 1 and 2.

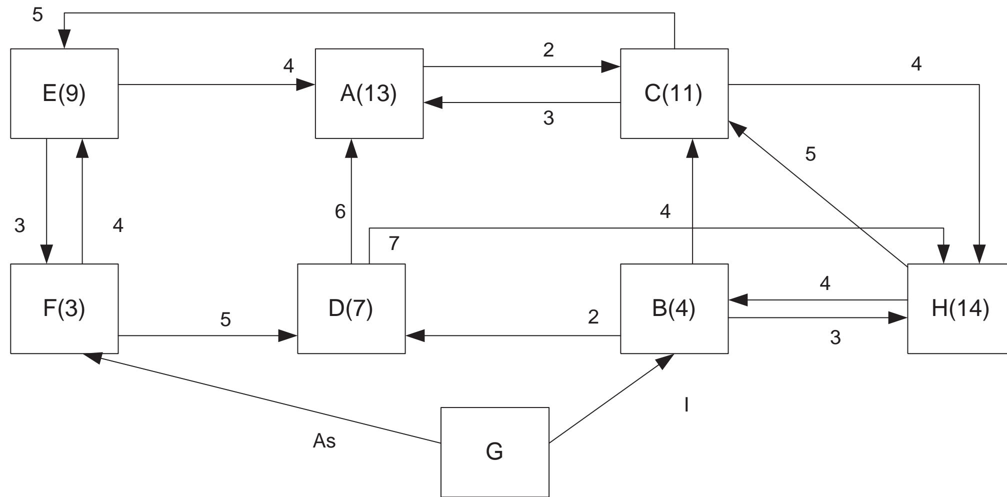

FIGURE 3. WORD - Node Weight is the Sum of Incoming Edge Weights (Algorithm 2, case 1)

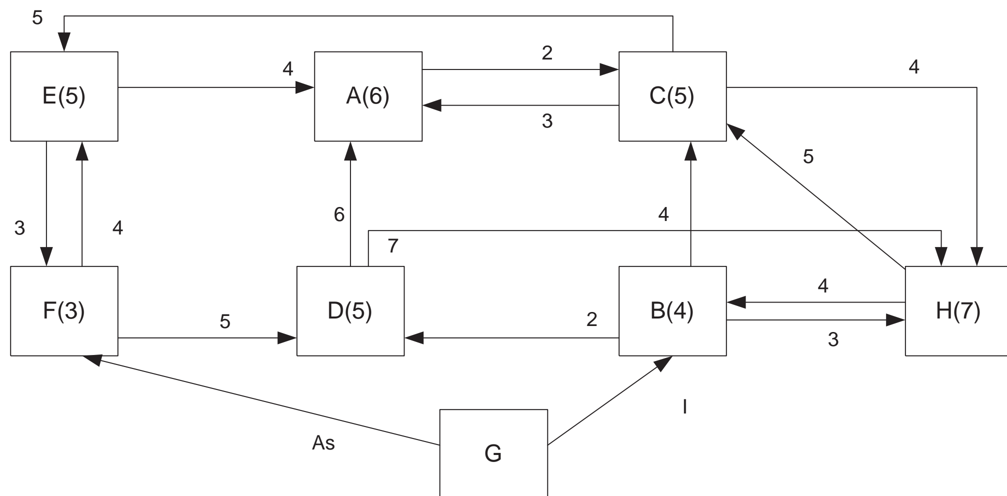

FIGURE 4. WORD - Node Weight is the Maximum of Incoming Edge Weights (Algorithm 2, case 2)

Case 1 is the same as considering all edges only, which was shown in the previous subsection. Cases 2 and 3 are illustrated through examples.

Case 2:

The node weight is defined as the maximum incoming edge weight. Table 4 shows the results from applying steps 4 through 7 of algorithm 2 on Figure 4. After the initial computation of CWR values for nodes, the algorithm follows the following steps:

A. Choose a node with maximum cycle-weight ratio in table 4 and remove that node from the *WORD*. The node with maximum CWR is C with a CWR of 1.6. Removing C breaks eight cycles; 2, 3, 4, 5, 6, 7, 8, and 9; leaving three.
B. Re-compute the cycle-weight ratio for the remaining edges in table 4. The result is shown in table 5. The node with maximum CWR value is node

TABLE 4. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\}$ in Figure 4

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	6	$\{2, 3, 6, 7\}$	4	0.67
2	B	4	$\{5, 6, 9, 10, 11\}$	5	1.25
3	C	5	$\{2, 3, 4, 5, 6, 7, 8, 9\}$	8	1.6
4	D	5	$\{3, 4, 5, 6, 11\}$	5	1
5	E	5	$\{1, 2, 3, 4, 5\}$	5	1
6	F	3	$\{1, 3, 4, 5\}$	4	1.33
7	H	7	$\{4, 5, 8, 9, 10, 11\}$	6	0.86

B. Removing node B breaks two cycles, 10 and 11, leaving cycle 1.

TABLE 5. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\}$ -{C} in Figure 4

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	5	$\{11\}$	1	0.2
5	E	5	$\{1\}$	1	0.2
6	F	3	$\{1\}$	1	0.33
7	H	7	$\{10, 11\}$	2	0.29

TABLE 6. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\}$ -{C, B} in Figure 4

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	6	$\{\}$	0	0
4	D	5	$\{\}$	0	0
5	E	5	$\{1\}$	1	0.2
6	**F**	3	$\{1\}$	1	0.33
7	H	7	$\{\}$	0	0

C. Re-compute the cycle-weight ratio for the remaining nodes in table 5. The result is shown in table 6. The node with the maximum cycle-weight ratio is F. Removing F breaks cycle 1 and makes the *WORD* acyclic.

Thus, removing three nodes, C, B, and F, made the graph acyclic with a total cost of 12. This is a slightly lower cost than with algorithm 1 (cost of 13).

Case 3:

The node weight is assumed to be between the maximum and the sum of the incoming edge weights. Table 7 shows the result from steps 4 to 7 of algorithm 2 on Figure 5. After the initial computation of CWR values for nodes, the algorithm works in the following steps:

A. Choose a node with maximum cycle-weight ratio in table 7 and remove that node from the *WORD*. Both C and F have the same maximum CWR. Our rule in this situation is to choose the node that is involved in larger number of cycles (node C). Removing C breaks eight cycles; 2, 3, 4, 5, 6, 7, 8, and 9; leaving three.

B. Re-compute the cycle-weight ratio for the remaining edges in table 7. The result is shown in table 8. The node with maximum CWR value is node B. Removing node B breaks two cycles, 10 and 11, leaving one.

C. Re-compute the cycle-weight ratio for the remaining nodes in table 8. The node with maximum cycle-weight ratio is F. Removing F breaks cycle 1 and makes the *WORD* acyclic.

TABLE 7. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\}$ in Figure 5

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	9	$\{2, 3, 6, 7\}$	4	0.67
2	B	4	$\{5, 6, 9, 10, 11\}$	5	1.25
3	C	6	$\{2, 3, 4, 5, 6, 7, 8, 9\}$	8	1.33
4	D	6	$\{3, 4, 5, 6, 11\}$	5	0.83
5	E	7	$\{1, 2, 3, 4, 5\}$	5	0.71
6	F	3	$\{1, 3, 4, 5\}$	4	1.33
7	H	10	$\{4, 5, 8, 9, 10, 11\}$	6	0.60

TABLE 8. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{C\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	9	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	6	$\{11\}$	1	0.17
5	E	7	$\{1\}$	1	0.14
6	F	3	$\{1\}$	1	0.33
7	H	10	$\{10, 11\}$	2	0.20

TABLE 9. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{C, B\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	9	$\{\}$	0	0
4	D	6	$\{\}$	0	0
5	E	9	$\{1\}$	1	0.11
6	F	3	$\{1\}$	1	0.33
7	H	10	$\{\}$	0	0

Thus, we break all 11 cycles by removing three nodes, C, B, and F. The total cost is 6+4+3 = 13. In this example, the cost is the same as the cost of using algorithm 1.

3.2.4. *Heuristic Algorithm for Breaking Cycles Using Node and Edge Weights*

This section presents an algorithm for breaking cycles using weight assignments on both nodes and edges. The algorithm is shown in Algorithm 3 and illustrated through examples in Figures 4 and 5.

Recall that there are three possible weights for a node. The first, where the node weight is equal to the sum of the incoming edge weights, gives the same result as considering only edge weights. Hence, two cases are considered: (1) node weights are the maximum

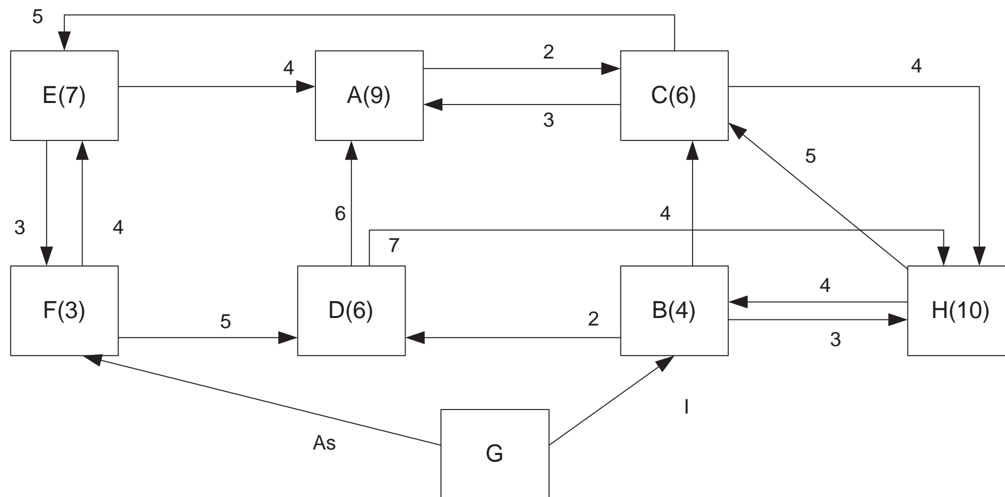

FIGURE 5. WORD - Node Weight is between the Maximum and Sum of Incoming Edge Weights (Algorithm 2, case 2)

Algorithm 3 Eliminating Cycles in WORD (V,E) Using Node and Edge Weights

1: Find all SCCs in WORD
2: **for** (each $scc_i(V_{scc_i}, E_{scc_i}) \in SCCs$) **do**
3: find all cycles CYCLES (totalCycles)
4: **for** (each $v \in V_{scc_i}$) **do**
5: find the number of cycles that use v ($cardinal\{cycles - through - v\}$)
6: compute the cycle-weight ratio
7: **end for**
8: **for** (each $e \in E_{scc_i}$) **do**
9: find the number of cycles that use e ($cardinal\{cycles - through - e\}$)
10: compute the cycle-weight ratio
11: **end for**
12: **while** (totalCycles != 0) **do**
13: order all *nodes* and *edges* in descending order of their cycle-weight ratio
14: remove the *node* or *edge* with the highest cycle-weight ratio
15: totalCycle = totalCycle - number of cycles broken
16: update the number of cycles that use v ($cardinal\{cycles - through - v\}$) or that use e ($cardinal\{cycles - through - e\}$) in the remaining node and edge sets
17: recompute the cycle-weight ratio for the remaining nodes and edges
18: **end while**
19: **end for**

of incoming edge weights, and (2) node weights are between the sum and maximum of incoming edge weights. Figures 4 and 5 are used for illustration.

Figure 4 shows node weights as the maximum of incoming edge weights. Previous tables 1 and 4 provide initial CWR values for edges and nodes. The rest of algorithm 3 works as follows:

A. Choose an edge or a node with maximum CWR from both tables 1 and 4. Then remove whichever has the greater CWR value from the *WORD*. Among edges, $A \rightarrow C$ has a maximum CWR value of 2, and among nodes, C has a maximum CWR value of 1.6. Thus, the edge $A \rightarrow C$ is chosen. Removing edge $A \rightarrow C$ breaks four cycles; 2, 3, 6, and 7; leaving seven.
B. Re-compute the cycle-weight ratio for the remaining edges and nodes in tables 1 and 4. The result for edges is the same as in table 2. The result for nodes is shown in table 10. Note that the weight of the node associated with the removed edge is also recomputed. The new node weight is reduced by the removed edge weight. From tables 2 and 10, choose a node or a edge with maximum CWR, which is node C. Removing node C breaks four cycles; 4, 5, 8, and 9; leaving three.
C. Compute the cycle-weight ratio for the remaining edges in table 2 and nodes in table 10. Note that removing a node also removes all edges associated with the node. The results are shown in tables 11 and 12. There are two edges and one node that have the same maximum value 0.5. Our rule in this situation is to choose the node that is involved in the larger number of cycles. Thus B is chosen. Removing B breaks two cycles, 10 and 11, leaving one cycle, 1.
D. Compute the cycle-weight ratio for the remaining edges and nodes in tables 11 and 12. The results are not shown because they are simple and can be

TABLE 10. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{5, 9, 10, 11\}$	4	1
3	C	3	$\{4, 5, 8, 9\}$	4	1.33
4	D	5	$\{4, 5, 11\}$	3	0.60
5	E	5	$\{1, 4, 5\}$	3	0.60
6	F	3	$\{1, 4, 5\}$	3	1
7	H	7	$\{4, 5, 8, 9, 10, 11\}$	6	0.86

TABLE 11. Cycle-weight Ratio for Edges in $SCC\{E, A, C, H, B, D, F\}$ -$\{A \rightarrow C, C\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
3	$B \rightarrow D$	2	{ 11 }	1	0.5
4	$B \rightarrow H$	3	{ 10}	1	0.33
8	$D \rightarrow A$	6	{ }	0	0
9	$D \rightarrow H$	7	{11}	1	0.14
10	$E \rightarrow A$	4	{ }	0	0
11	$E \rightarrow F$	3	{1 }	1	0.33
12	$F \rightarrow D$	5	{ }	0	0
13	$F \rightarrow E$	4	{1}	1	0.25
14	$H \rightarrow B$	4	{10,11 }	2	0.5

TABLE 12. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C, C\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	6	$\{\}$	0	0
2	B	4	$\{10, 11\}$	2	0.5
4	D	5	$\{11\}$	1	0.20
5	E	5	$\{1\}$	1	0.20
6	F	3	$\{1\}$	1	0.33
7	H	7	$\{\}$	0	0

seen in tables 11 and 12. According to the rule, F is removed, completing the cycle removing process.

In conclusion, the graph is made acyclic by removing one edge, $A \rightarrow C$, and three nodes; C, B, and F. The total cost is 2 + 3 + 4 + 3 = 12. In this example, this is the same cost as using node weights as the maximum of incoming edge weights and not considering edge weights.

Figure 5 shows node weights for case 3, between the maximum and the sum of incoming edge weights. Previous tables 1 and 7 provide initial CWR values for

TABLE 13. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{A \rightarrow C\}$

No.	Node	Wt.	Cycles Invol-ved	NC	CWR
1	A	9	$\{\}$	0	0
2	B	4	$\{5, 9, 10, 11\}$	4	1
3	C	4	$\{4, 5, 8, 9\}$	4	1
4	D	6	$\{4, 5, 11\}$	3	0.50
5	E	7	$\{1, 4, 5\}$	3	0.43
6	F	3	$\{1, 4, 5\}$	3	1
7	H	10	$\{4, 5, 8, 9, 10, 11\}$	6	0.60

edges and nodes. The rest of algorithm 3 works as follows:

A. Choose an edge or a node with maximum CWR from both tables 1 and 7. Then remove whichever has the greater CWR value from the *WORD*. Among edges, $A \rightarrow C$ has a maximum CWR value of 2, and among nodes, C and F have a maximum CWR value of 1.33. Thus, edge $A \rightarrow C$ is removed. Removing edge $A \rightarrow C$ breaks four cycles; 2, 3, 6, and 7; leaving seven.
B. Re-compute the cycle-weight ratio for the remaining edges and nodes in tables 1 and 7. The result for edges is the same as in table 2. The result for nodes is shown in table 13. Note that the weight of the node associated with the removed edge is also recomputed. The new node weight is reduced by the removed edge weight. From tables 2 and 13, choose a node or a edge with maximum CWR value. When more than one node or edge has the same maximum CWR value, our rule is to choose a **node** that is involved in the larger number of cycles. B and C have the same weight and are also involved in the same number of cycles. In this situation, we arbitrarily chose B. Removing B breaks four cycles; 5, 9, 10, and 11; leaving three.
C. Re-compute the cycle-weight ratio for the remaining edges in table 2 and nodes in table 13. Note that removing a node also removes all edges associated with the node. The results are shown in tables 14 and 15. Node F is removed, which breaks two cycles, 1 and 4, leaving one cycle, 8.
D. Re-compute the cycle-weight ratio for the remaining edges and nodes in tables 14 and 15. The results are not shown because they are simple and can be seen in tables 14 and 15. According to the rule, C is removed, and this completes the cycle removing process.

In conclusion, the graph is made acyclic by removing one edge, $A \rightarrow C$, and three nodes; B, F, and C. The total cost is 2 + 4 + 3 + 4 = 13. This is the same cost as using only edge weights.

TABLE 14. Cycle-weight Ratio for Edges in $SCC\{E, A, C, H, B, D, F\}$ **-**$\{A \to C, B\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
5	$C \to E$	5	{4 }	1	0.2
6	$C \to A$	3	{ }	0	0
7	$C \to H$	4	{8 }	1	0.25
8	$D \to A$	6	{ }	0	0
9	$D \to H$	7	{4}	1	0.14
10	$E \to A$	4	{ }	0	0
11	**E → F**	**3**	**{1,4}**	**2**	**0.67**
12	$F \to D$	5	{4}	1	0.2
13	$F \to E$	4	{1}	1	0.25
15	$H \to C$	5	{4, 8}	2	0.4

TABLE 15. Cycle-weight Ratio for Nodes in $SCC\{E, A, C, H, B, D, F\} - \{A \to C, B\}$

No.	Node	Wt.	Cycles Involved	NC	CWR
1	A	9	{}	0	0
3	C	4	{4,8}	2	0.5
4	D	6	{4}	1	0.17
5	E	7	{1,4}	2	0.29
6	F	3	{1,4}	2	0.67
7	H	10	{4,8}	0	0

Using both node and edge weights are similar to using only node weights. This is because node weights are computed using edge weights.

3.3. Algorithm for Ordering Classes for Integration Testing

Once cycles are broken by automation, the integration tester needs a specific ordering of the classes, especially for classes that appear in different SCCs. Algorithm 4 describes an approach for ordering classes for integration and testing. The algorithm first generates a *precedence table* for nodes in the WORD, then finds all strongly connected components (SCCs) in the weighted object relation diagram (WORD). A *precedence table* is indexed by the number or name of nodes in the WORD, and shows the nodes that are connected to a node through outgoing edges from the node. Then, each SCC is compressed into a node, and the multiple edges between SCCs are combined into one edge. The result is a compressed WORD, which is acyclic and directed. The example in Figure 6 represents a WORD with three SCCs, $\{1, 2\}$, $\{3, 4\}$, and $\{5\}$. Figure 7 shows the resulting compressed WORD, $WORD_{comp}$. Algorithm 4 finds the reversed topological order, O_{SCCs}, for the $WORD_{comp}$ to be (1) $\{5\}$, (2) $\{3, 4\}$, and (3) $\{1, 2\}$.

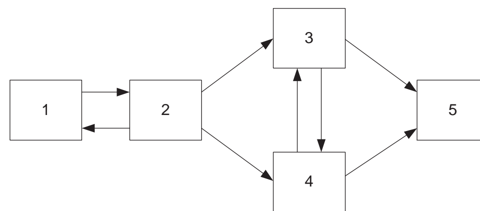

FIGURE 6. Finding Overall Test Order in a WORD

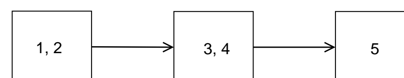

FIGURE 7. Compressed WORD

Each strongly connected component scc_i is made acyclic using Algorithm 1, resulting in the subgraph $scc_{i-acyclic}$. A test stub needs to be developed for each edge that was removed. Suppose edges $1 \to 2$ and $3 \to 4$ are removed from SCCs $\{1, 2\}$ and $\{3, 4\}$. The result is that test stubs need to be created for the classes represented by nodes 2 and 4. Algorithm 4 is then used to generate a reverse topological order, O_{scc_i}, for each $scc_{i-acyclic}$. In this example, SCCs $\{1, 2\}$ and $\{3, 4\}$ have reverse topological orders of 1, 2 and 3, 4. Integration and testing proceeds according to this order. For each node in O_{SCCs}, first, $scc_{i-acyclic}$ is restored, and the nodes that are included are tested according to the order O_{scc_i}. For Figure 6, the integration and test order is 5, 3, 4, 1, 2. Before a node is tested, the *precedence table* is checked. If a node was connected to an edge that was removed, the corresponding test stub must be included in the test order. For example, when node 3 is tested, the precedence table indicates that node 3 is connected to an untested node. Thus, the test stub for node 4 is included at this point.

4. CASE STUDY

This section provides a preliminary evaluation of the model and algorithms by comparing results with the same project, the ATM system, used by Briand et al. [2]. Briand et al. chose the number of broken dependencies, attribute couplings, method couplings, and a combination of attributes and methods as four cost functions to produce an integration test order, and compared the results to decide which cost function gives the best result. Our approach is first to use dependencies, attribute coupling measures, method coupling measures, and a combination of attribute and method coupling measures as weights on edges and apply our algorithm to check what kind of result can be obtained under similar situations. Then, we collect coupling data from the implementations using coupling definitions and coupling measures defined in Section 3.1, construct the weighted object relation diagram

TABLE 16. Coupling Measures for Edges in SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$

No.	8	9	10	11	12	13	14	15
8		1.0.0.0.0	1.0.2.1.3					
9	1.0.1.1.3							
10	1.0.6.4.4	1.0.1.1.1		1.0.3.5.3	1.0.1.1.0	10.1.1.0	1.0.1.1.0	1.0.1.1.0
11	1.0.1.1.3	1.0.0.0.0	1.0.1.1.0					
12	1.0.4.3.11	1.0.4.3.11	1.0.2.2.0	5.0.0.0.0				
13	1.0.4.3.6	1.0.4.3.14	1.0.2.2.0	5.0.0.0.0				
14	1.0.3.2.6	1.0.3.3.12	1.0.2.2.0	5.0.0.0.0				
15	1.0.2.1.6	1.0.3.3.10	1.0.2.2.0	5.0.0.0.0				

Algorithm 4 Ordering Classes for Integration and Testing

1: Generate *precedence table* for nodes in the WORD
2: Find all SCCs in WORD
3: Generate an acyclic compressed version of WORD, $WORD_{comp}$, by representing each scc_i as a single node and by compressing multiple edges between every two nodes
4: Find reverse topological order, O_{SCCs}, of nodes in $WORD_{comp}$
5: **for** (each $scc_i \in SCCs$) **do**
6: make scc_i acyclic by using Algorithm 1 and record removed edges
7: find reverse topological order, O_{scc_i}, for nodes in the acyclic $scc_{i-acyclic}$
8: **end for**
9: Start testing according to the order of SCCs in O_{SCCs}
10: **for** (each ordered $scc_i \in SCCs$) **do**
11: test nodes in the order O_{scc_i}
12: **end for**

(WORD), and compute the edge weights for SCCs in the WORD using equations 2 and 7.

The ATM system has 21 classes and eight form a strongly connected component that has 30 cycles [2]. Table 16 shows the coupling measures in the format of equation 1. Table 17 shows the different edge weights that are used in this evaluation. In particular, the columns labeled Dependency, # of Attributes, # of Methods, and A & M show the edge weights obtained from Briand et al.'s four cost functions. The last column shows edge weights that are computed from Table 16 using equations 4, 2, 3, and 7. The constraints of not breaking inheritance and composition edges are achieved by assigning 5 to variable C in equation 1 for inheritance and composition, and 1 for the others.

In all five approaches, seven dependencies were removed. When using weights in the columns labeled # of Attributes, # of Methods, A & M, and A & M-new of Table 17, exactly the same set of edges were removed. Hence, the stubbing cost for these approaches are equal. Although seven dependencies were broken when using the existence of dependencies as a cost function, the stubbing cost may vary because the edge weights are the same and thus cannot reflect any stubbing cost at the time of deciding to choose an edge to remove between two equal weight edges.

TABLE 17. Different Weights for Edges in SCC $\{8, 9, 10, 11, 12, 13, 14, 15\}$

No.	Edge	Dep.	# of Attr.	# of Meth.	A & M	A & M-new
1	$8 \to 9$	1	13	1	0.71	1
2	$8 \to 10$	1	9	2	0.53	1.22
3	$9 \to 8$	1	13	7	1	1.17
4	$10 \to 8$	1	13	7	1	1.66
5	$10 \to 9$	1	13	2	0.74	1.13
6	$10 \to 11$	1	0	0	0	1.57
7	$10 \to 12$	1	2	2	0.23	1.13
8	$10 \to 13$	1	2	2	0.23	1.13
9	$10 \to 14$	1	3	2	0.26	1.13
10	$10 \to 15$	1	1	2	0.21	1.13
11	$11 \to 8$	1	13	2	0.74	1.17
12	$11 \to 9$	1	13	1	0.71	1.00
13	$11 \to 10$	1	9	2	0.53	1.13
14	$12 \to 8$	1	13	4	0.81	1.60
15	$12 \to 9$	1	13	4	0.81	2.59
16	$12 \to 10$	1	9	2	0.53	2.01
17	$12 \to 11$	1	∞	∞	∞	5.00
18	$13 \to 8$	1	13	4	0.81	1.50
19	$13 \to 9$	1	13	4	0.81	2.62
20	$13 \to 10$	1	9	2	0.53	2.01
21	$13 \to 11$	1	∞	∞	∞	5.00
22	$14 \to 8$	1	13	3	0.77	1.39
23	$14 \to 9$	1	13	3	0.77	2.58
24	$14 \to 10$	1	9	2	0.53	2.01
25	$14 \to 11$	1	∞	∞	∞	5.00
26	$15 \to 8$	1	13	2	0.74	1.29
27	$15 \to 9$	1	13	3	0.77	2.58
28	$15 \to 10$	1	9	2	0.53	2.01
29	$15 \to 11$	1	∞	∞	∞	5.00

The results indicate that when we consider the stub complexity as weights on edges, graph-based algorithms

can produce results as good as those produced by genetic algorithms. However, the GA approach must be run many times, greatly complicating the process. The new algorithms in this paper only run once. They can also use more information, specifically node weights and shared stubbing, and thus can provide a more precise estimation of test stub complexity. A larger empirical evaluation needs to be carried out to verify that this result generalizes.

5. CONCLUSIONS AND FUTURE WORK

This paper presents an improved technique and algorithms to automatically solve the CITO problem. The technique uses weights to represent the cost of creating stubs. This has been done before, but the weights in this research are derived from quantitative analysis of couplings, thus obtaining more precise results. These weights are placed on a *Weighted Object Relation Diagram (WORD)*, which represents classes as nodes and relationships as edges. This paper also introduces the idea of applying weights to nodes to estimate the cost of removing the nodes. If a class is used by multiple classes, then all or part of the same stub for that class may be shared among all classes that use it, thus reducing the cost of stubbing. A question that arises is how much does shared stubbing reduce the cost? This question breaks down into two parts, (1) how often does shared stubbing occur, and (2) how much savings do we get from recognizing the shared stubbing. In our experience, shared stubbing occurs most of the time a class is used by more than one client. Determining the savings would quite difficult, and it seems likely that this would be very application dependent. We hope to investigate this question in future empirical work.

The weight of a node is at least as high as the maximal weight of all incoming edges (assuming total sharing of the stub), and no higher than the sum of the weights of all incoming edges (assuming no sharing of the stub).

New algorithms to solve the CITO problem are introduced. These algorithms use edge and node weights. They were compared with algorithms by previous researchers, and found to be just as effective if edge weights are ignored. Overall, the results in this paper improve the ability for developers to automate the CITO problem. An open question is how well these results will scale to large systems and we hope to work on that problem soon. In the future, we plan to complete automation of this work and then carry out detailed experiments to fully assess the value of the technique.

6. ACKNOWLEDGMENTS

We would like to thank Prof. Lionel Briand and his colleagues for sharing their experimental materials with us.

REFERENCES

[1] Kung, D., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1995) A test strategy for object-oriented programs. *19th Computer Software and Applications Conference (COMPSAC 95)*, Dallas, TX, August, pp. 239 –244. IEEE Computer Society Press.

[2] Briand, L., Feng, J., and Labiche, Y. (2002) Using genetic algorithms and coupling measures to devise optimal integration test orders. *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, Ischia, Italy, pp. 43–50. IEEE Computer Society Press.

[3] Beizer, B. (1990) *Software Testing Techniques*, 2nd edition. Van Nostrand Reinhold, Inc, New York NY. ISBN 0-442-20672-0.

[4] Briand, L. C., Labiche, Y., and Wang, Y. (2003) An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, **29**, 594–607.

[5] Tai, K.-C. and Daniels, F. (1997) Test order for inter-class integration testing of object-oriented software. *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, Santa Barbara CA, pp. 602–607. IEEE Computer Society.

[6] Traon, Y. L., Jéron, T., Jézéquel, J.-M., and Morel, P. (2000) Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, **49**, 12–25.

[7] Malloy, B. A., Clarke, P. J., and Lloyd, E. L. (2003) A parameterized cost model to order classes for class-based testing of C++ applications. *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, Denver, Colorado, pp. 353–364. IEEE Computer Society Press.

[8] Briand, L., Labiche, Y., and Wang, Y. (2001) Revisiting strategies for ordering class integration testing in the presence of dependency cycles. Technical report sce-01-02. Careleton University.

[9] Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

[10] Harrold, M. J. and McGregor, J. D. (1992) Incremental testing of object-oriented class structures. *14th International Conference on Software Engineering*, Melbourne, Australia, May, pp. 68–80. IEEE Computer Society Press.