A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization

Sriraman Tallam
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
tmsriram@cs.arizona.edu

Neelam Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

ABSTRACT

Software testing and retesting occurs continuously during the software development lifecycle to detect errors as early as possible and to ensure that changes to existing software do not break the software. Test suites once developed are reused and updated frequently as the software evolves. As a result, some test cases in the test suite may become redundant as the software is modified over time since the requirements covered by them are also covered by other test cases. Due to the resource and time constraints for re-executing large test suites, it is important to develop techniques to minimize available test suites by removing redundant test cases. In general, the test suite minimization problem is NP complete. In this paper, we present a new greedy heuristic algorithm for selecting a minimal subset of a test suite T that covers all the requirements covered by T. We show how our algorithm was inspired by the concept analysis framework. We conducted experiments to measure the extent of test suite reduction obtained by our algorithm and prior heuristics for test suite minimization. In our experiments, our algorithm always selected same size or smaller size test suite than that selected by prior heuristics and had comparable time performance.

Keywords - test cases, testing requirements, test suite minimization, concept analysis.

1. INTRODUCTION

Software testing accounts for a significant cost of software development. As software evolves, the sizes of test suites grow as new test cases are developed and added to the test suite. Due to time and resource constraints, it may not be possible to rerun all the test cases in the test suites every time software is tested following some modifications. Therefore, it is important to develop techniques to select a subset of test cases from the available test suite that exercise the given set of requirements. The test suite minimization problem can be stated as follows:

Problem Statement: Given a set T of test cases $\{t_1, t_2, t_3, ..., t_n\}$, a set of testing requirements $\{r_1, r_2, \cdots, r_m\}$ that must be covered to provide the desired coverage of the program, and the information about the testing requirements exercised by each test case in T, the test suite minimization problem is to find a *minimal* cardinality subset of T that exercises the same set of requirements as those exercised by the un-minimized test suite T.

In general, the problem of selecting a minimal cardinality subset of T that covers all the requirements covered by T is NP complete. This can be easily shown by a polynomial time reduction from the *minimum set-cover* problem [7] to the test suite minimization problem. Given a finite set of attributes X and m subsets $S_1, S_2, ..., S_m$ of these attributes, the minimum set-cover problem is to find the fewest number of these subsets needed to cover all the attributes. Since the minimum set-cover problem is NP complete in general, therefore heuristics for solving this problem become important. A classical approximation algorithm [5, 6] for the minimum set-cover problem uses a simple greedy heuristic. This heuristic picks the set that covers the most points, throws out all the points covered by the selected set, and repeats the process until all the points are covered. When there is a tie among the sets, one set among those tied is picked arbitrarily. To illustrate this classical greedy heuristic, let

Table 1: An Example showing the requirements exercised by test cases in a test suite.

	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	

us apply it to minimize the test suite $\{t_1, t_2, t_3, t_4, t_5\}$ shown in Table 1. The coverage information for each test case is shown by an X in the corresponding column in this Table. The greedy heuristic will first select the test case t_1, and throw out the requirements r_1, r_2 and r_3 from further consideration. Next, either of t_2, t_3, t_4 or t_5 could be picked since each of these cover one yet uncovered requirement. Let us say t_2 is selected. Now the requirement r_4 is thrown out. In the next step, t_3, t_4 and t_5 are tied. Let us say t_3 is selected in this step. Then the requirement r_5 will be thrown out. Finally the test case t_4 will get selected to cover the requirement r_6. Thus, the minimized suite generated by this heuristic consists of 4 test cases t_1, t_2, t_3 and t_4 . This example points out the drawback of this greedy heuristic. The redundant test case t_1 was selected because the decision to select t_1 was made too early. The choice of picking t_1 before picking any of the other test cases seemed a good decision at the time when t_1 was selected, however, it turned out to be not the best choice for computing the overall minimal test suite. The optimally minimized test suite for this example has only 3 test cases t_2, t_3 and t_4. The above classical greedy heuristic algorithm [5, 6] primarily exploits the *implications among the test cases* to

PASTE '05 Lisbon, Portugal

identify redundant test cases and exclude them from further consideration.

Another heuristic (referred to as HGS algorithm from here onwards) to minimize test suites was developed by Harrold, Gupta and Soffa in [8]. Given a test suite T and a set of testing requirements $r_1, r_2, \cdots, r_n$ that must be exercised to provide the desired testing coverage of the program, the technique considers the subsets $T_1, T_2, \cdots, T_n$ of T such that any one of the test cases t_j belonging to T_i can be used to test r_i. The HGS algorithm first includes all the test cases that occur in T_i's of cardinality one in the representative set and marks all T_i's containing any of these test cases. Then T_i's of cardinality two are considered. Repeatedly, the test case that occurs in the maximum number of T_i's of cardinality two is chosen and added to the representative set. All unmarked T_i's containing these test cases are marked. This process is repeated for T_i's of cardinality 3, 4, $\cdots$, max, where max is the maximum cardinality of the T_i's. In case there is a tie among the test cases while considering T_i's of cardinality m, the test case that occurs in the maximum number of unmarked T_i's of cardinality $m+1$ is chosen. If a decision cannot be made, the T_i's with greater cardinality are examined and finally a random choice is made. Let us consider the example in Table 2. Each row in Table 1 shows the requirements exercised by that test case. In this example, $T_1 = \{t_1, t_2\},\ T_2 = \{t_1, t_3\},\ T_3 = \{t_2, t_3, t_4\},\ T_4 = \{t_3, t_4, t_5\},\ T_5 = \{t_2, t_6, t_7\}$. Since there is no T_i of cardinality one, the HGS heuristic considers

Table 2: Another Example showing the requirements exercised by test cases in a test suite.

	r_1	r_2	r_3	r_4	r_5
t_1	X	X			
t_2	X		X		X
t_3		X	X	X	
t_4			X	X	
t_5				X	
t_6					X
t_7					X

T_1 and T_2 (each with cardinality two) and selects the test case t_1. Next, T_3, T_4 and T_5 (each with cardinality three) are considered. The tie between t_2, t_3 and t_4 is broken arbitrarily say in favor of selecting the test case t_2. Now only r_4 remains to be exercised, i.e. T_4 is still unmarked. Any of test cases t_3, t_4 or t_5 can be selected at this stage. Let us say t_3 is now selected. Thus, the reduced test suite selected by the HGS heuristic for this example is $\{ t_1, t_2, t_3 \}$. However, the requirements exercised by t_1 are also exercised by t_2 and t_3 and hence the test case t_1 is redundant. This redundant test case was selected because the decision to select t_1 was made too early.

Agrawal [1, 2] used the notion of dominators, superblocks and megablocks [1, 2] to derive coverage implications among the basic blocks to reduce the coverage requirements for a program. Similarly, Marre and Bertolino [13] use a notion of entities subsumption to determine a reduced set of coverage entities such that coverage of the reduced set implies the coverage of the un-reduced set. These works [1, 2, 13] exploit only the *implications among the coverage requirements* to generate a reduced set of coverage requirements.

We explored the concept analysis framework in an attempt to derive a better heuristic for test suite minimization. Concept analysis [3] is a technique for classifying *objects* based upon the overlap among their *attributes*. We viewed each test case in a test suite as an object and the set of requirements covered by the test case as its attributes. We observed that the concept lattice exposed the implications among the test cases in a test suite *as well as* the implications among the requirements covered by the test suite. Thus, it presented a unified framework to identify steps of a new greedy test suite minimization algorithm that iteratively exploits the implications among the test cases *and* the implications among the requirements to reduce the context table. Although the steps of our algorithm were inspired by the concept analysis framework, we implemented the steps directly on the original context table (such as Table 1) that provides the mapping between the test cases and the requirements covered by each test case in the un-minimized suite. As a result, we developed a new polynomial time greedy algorithm that produces same size or smaller size reduced suites as the classical greedy heuristic [6, 5]. Our algorithm generates the optimal size minimized suites for examples in Tables 1 and 2. We call our algorithm as *Delayed-Greedy* since it postpones the application of the greedy heuristic as opposed to the classical greedy algorithm which applies it at every step. We implemented our algorithm and conducted experiments with the programs from the Siemens suite [10, 11] and the *space* program [11]. In our experiments, our algorithm always produced same size or smaller size reduced suites than prior heuristics. The important contributions of the paper are:

- A new greedy heuristic algorithm called *Delayed-Greedy*, that is guaranteed to obtain same or more reduction in the test suite size as compared to the classical greedy [5, 6] heuristic.
- In our experiments, Delayed-Greedy always produced same size or smaller size reduced test suites than prior heuristics for test suite minimization.
- The time performance of Delayed-Greedy was found to be comparable in our experiments and it always produced same or more number of optimally reduced test suites than other heuristics.

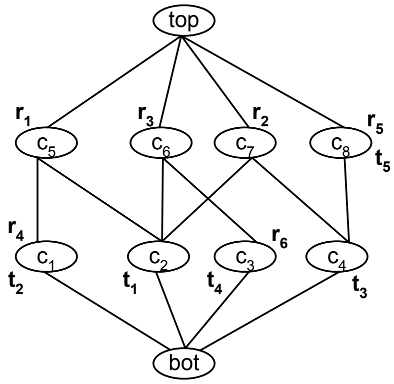
The organization of this paper is as follows. In section 2, we show how concept analysis framework guided us to develop a new algorithm for test suite minimization. In section 3, we describe our Delayed-Greedy algorithm for test suite minimization. We present our experimental results in section 4 and discuss the related work in section 5. Finally, we summarize the results in section 6.

2. CONCEPT ANALYSIS AND TEST SUITE MINIMIZATION

Concept Analysis is a hierarchical clustering technique [3] for objects with discrete attributes. The input to concept analysis is a set of *objects* O and a set of *attributes* A, and a binary relation $R \subseteq O \times A$ called the *context* which relates object to their attributes. *To analyze the test suite minimization problem using concept analysis, the test cases can be considered as the objects and the requirements as their attributes. The coverage information for each test case is the relation between the objects and the attributes.* We now have a context that can be analyzed using a concept analysis framework. Let us consider the test suite minimization for the context table (same as in Table 1) in the example shown in Figure 1.

Concept Analysis identifies maximal groupings of objects and attributes called *concepts*. A concept is an ordered pair (X, Y) where $X \subseteq O$ is a set of objects and $Y \subseteq A$ is a set of attributes satisfying the property that X is the maximal set of objects that are related to all the attributes in Y and Y is the maximal set of attributes that are related to all the objects in X. For example, the set $\{t_1, t_3\}$ is the maximal set of test cases that covers the requirement r_2. This is shown by concept c_7 in the table on the right in Figure 1. Similarly, $\{t_1\}$ is the maximal set of test cases that covers all of the requirements r_1, r_2, r_3. This is shown by the concept c_2 in the table on the right in Figure 1. The concepts form a partial order defined as follows. For concepts (X_1, Y_1) and (X_2, Y_2),

	r_1	r_2	r_3	r_4	r_5	r_6
t_1	X	X	X			
t_2	X			X		
t_3		X			X	
t_4			X			X
t_5					X	



Concept	(Objects,Attributes)
TOP	$(\{t_1, t_2, t_3, t_4, t_5\},\{\})$
c_1	$(\{t_2\},\{r_1, r_4\})$
c_2	$(\{t_1\},\{r_1, r_2, r_3\})$
c_3	$(\{t_4\},\{r_3, r_6\})$
c_4	$(\{t_3\},\{r_2, r_5\})$
c_5	$(\{t_1, t_2\},\{r_1\})$
c_6	$(\{t_1, t_4\},\{r_3\})$
c_7	$(\{t_1, t_3\},\{r_2\})$
c_8	$(\{t_3, t_5\},\{r_5\})$
BOT	$(\{\},\{r_1, r_2, r_3, r_4, r_5, r_6\})$

Figure 1: Context table, concept lattice and the table of concepts.

$(X_1, Y_1) \leq_R (X_2, Y_2)$ iff $X_1 \subseteq X_2$. For example, in Figure 1, $c_2 \leq_R c_7$ since $\{t_1\} \subseteq \{t_1, t_3\}$. This partial order induces a complete lattice on the concepts, called the *concept lattice*. The top (bottom) element of the lattice is the concept with all the objects (attributes) and none of the attributes (objects). If (X_1, Y_1) and (X_2, Y_2) are two concepts such that $(X_1, Y_1) \leq_R (X_2, Y_2)$, then $X_1 \subseteq X_2$ and $Y_1 \supseteq Y_2$. The concepts (maximal groupings) with their respective objects and attributes and the concept lattice for the example being considered are shown in the Figure 1.

For every object $o \in O$, there is a unique smallest (with respect to $\leq_R$) concept in which it appears. This concept c, the smallest concept for o, is *labeled* with o. For example, in Figure 1, the concept c_2 is labeled with t_1 and t_1 also appears in c_7 since $c_2 \leq_R c_7$. Analogously, for every attribute $a \in A$, there is a unique largest concept in which it appears. This concept c', the largest concept for a, is labeled with a. For example, in Figure 1, the concept c_5 is labeled with the attribute r_1 and r_1 also appears in c_1 since $c_1 \leq_R c_5$. Given such a labeling of the lattice, we define the following two implications.

Object Implication: Given two objects $o_1, o_2 \in O$, $o_1 \Rightarrow o_2$ *iff* $\forall a \in A,\ (o_2\ R\ a) \Rightarrow (o_1\ R\ a)$. Such an implication can be detected from the lattice as follows. The implication $o_1 \Rightarrow o_2$ is true if the concept *labeled* with o_1 occurs *lower* in the lattice than the concept labeled with o_2 and the two concepts are ordered. In this case, all the attributes covered by o_2 are also covered by o_1. In other words, for two objects o_1 and o_2, if $o_1 \Rightarrow o_2$ then the row corresponding to the object o_2 can be safely removed from the context table without affecting the size of the minimal subset of test suite that covers all the requirements. We refer to this as the *object reduction rule.*

In Figure 1, the concept c_4 is labeled with the test case t_3 and the concept c_8 is labeled with test case t_5. Since c_4 occurs lower in the lattice than c_8, therefore there is an object implication $t_3 \Rightarrow t_5$. This means selecting t_3 makes t_5 redundant with respect to coverage. Indeed this is the case as can be seen from the context table since the set of requirements covered by t_5 is a subset of the requirements covered by t_3. Therefore, the row corresponding to t_5 can be safely removed from the context table. The context table in Figure 2 shows the row for t_5 removed from the table by applying the object reduction rule.

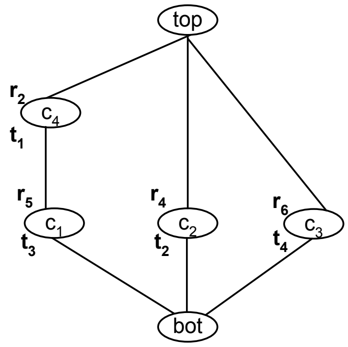
Attribute Implication: Given two attributes $a_1, a_2 \in A$, $a_1 \Rightarrow a_2$ *iff* $\forall\ o \in O,\ (o\ R\ a_1) \Rightarrow (o\ R\ a_2)$. An attribute implication can be detected from the lattice as follows. The implication $a_1 \Rightarrow a_2$ is true if the concept *labeled* with a_1 occurs *lower* in the lattice than the concept labeled with a_2 and the two concepts are ordered. In this case, the column corresponding to the attribute a_2 can be removed from the context table since coverage of a_1 would also imply coverage of a_2. In other words, the requirement a_2 can be safely removed from the context table without affecting the optimality of the solution. We refer to this as *attribute reduction rule.* In the lattice in Figure 1, the concept c_1 is labeled with the attribute r_4, and the concept c_5 is labeled with the attribute r_1. Since c_1 occurs lower in the lattice than c_5, there is an attribute implication $r_4 \Rightarrow r_1$. This means if the requirement r_4 is covered by some test case and this test case is selected, then the coverage requirement for attribute r_1 becomes redundant since it is also covered by this test case. From the context table we can see that it is indeed true that to cover r_4 we need to select the test case t_2, and that t_2 also covers r_1. Therefore, the column for r_1 can be safely removed from the context table. Similarly, the column for r_3 can be removed from the context table since coverage of r_6 implies the coverage of r_3.

The context table in Figure 2 shows the reduced table after removing the columns corresponding to r_1 and r_3 by applying the attribute reductions and after removing the row for t_5 by applying the object reduction. The corresponding concept lattice and the table of concepts for the reduced context table are also shown in Figure 2. Note that as a result of the above attribute reductions, a new object implication $t_3 \Rightarrow t_1$ is exposed in the reduced context table in the Figure 2. Applying this reduction removes the row for t_1 from the context table shown in Figure 2 and the resulting context table is shown in Figure 3.

In the example in section 1, the classical greedy [5, 6] heuristic removed t_5 from the un-minimized suite but it was not able to identify that coverage of r_4 and r_6 implied the coverage of r_1 and r_3 respectively. In contrast, the techniques in [1, 2, 13] will be able to exploit the above attribute implications, however they do not exploit the implications among the test cases. *Thus, we noted that the classical greedy heuristic exploits only the object implications during minimization and it misses the additional opportunities for minimization enabled by the attribute implications. Similarly, the techniques in [1, 2, 13] exploit only the attribute implications and miss the opportunities for reduction enabled by the object implications.* Note that we were able to make the above critical observation only because the concept analysis framework exposed these different types of implications simultaneously in a single framework namely the concept lattice.

Owner Reductions: We define the *strongest* concepts of the lattice as the ones that are immediately above the bottom concept of the lattice. In the lattice shown in Figure 1, c_1, c_2, c_3 and c_4 are

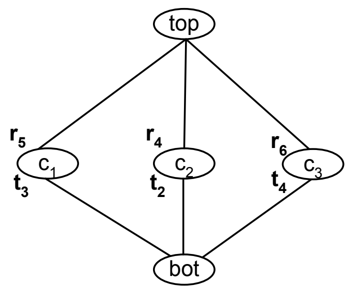
	r_2	r_4	r_5	r_6
t_1	X			
t_2		X		
t_3	X		X	
t_4				X



Concept	(Objects,Attributes)
TOP	$(\{t_1, t_2, t_3, t_4\},\{\})$
c_1	$(\{t_3\},\{r_2, r_5\})$
c_2	$(\{t_2\},\{r_4\})$
c_3	$(\{t_4\},\{r_6\})$
c_4	$(\{t_1, t_3\},\{r_2\})$
BOT	$(\{\},\{r_2, r_4, r_5, r_6\})$

Figure 2: Reduced context table, concept lattice and table of concepts after applying object reduction $t_3 \Rightarrow t_5$ and attribute reductions $r_6 \Rightarrow r_3$ and $r_4 \Rightarrow r_1$ to context table in Figure 1.

	r_2	r_4	r_5	r_6
t_2		X		
t_3	X		X	
t_4				X



Concept	(Objects,Attributes)
TOP	$(\{t_2, t_3, t_4\},\{\})$
c_1	$(\{t_3\},\{r_2, r_5\})$
c_2	$(\{t_2\},\{r_4\})$
c_3	$(\{t_4\},\{r_6\})$
BOT	$(\{\},\{r_2, r_4, r_5, r_6\})$

Figure 3: Reduced context table, concept lattice and table of concepts after applying object reduction $t_3 \Rightarrow t_1$ to context Table in Figure 2.

the strongest concepts. If any strongest concept s of the lattice is *labeled* with an attribute a then it implies that a test case in the concept s *must* be chosen in order to cover that attribute. Since s is a strongest concept and is *labeled* with attribute a, only the test cases contained in the object set of s cover a. So, a test case in s has to be selected to cover attribute a. We refer to this as the **owner reduction rule**. For instance, by applying the owner reductions to the table in Figure 3, we get an empty table and we are done. Thus, for this example our algorithm generates the optimal size minimized test suite $\{t_2, t_3, t_4\}$.

In [15], Sampath et. al presented a concept analysis based algorithm that constructs the concept lattice for the given context table and conservatively selects one test case from *each* of the strongest (they call them next-to-bottom) concepts to generate reduced test suites. In other words, their algorithm does not consider whether the strongest concept is *labeled* with an attribute or not. This can result in selecting test cases that are redundant with respect to coverage of additional requirements beyond those already covered by the previously selected test cases. For example, for the context table and the corresponding concept lattice in Figure 1, their algorithm will select one test case each from c_1, c_2, c_3 and c_4 and this will result in the reduced test suite $\{t_2, t_1, t_4, t_3\}$. The test case t_1 is redundant since the attributes covered by test cases in c_2 are covered by the test cases in the other strongest concepts.

It should be noted at this point, that *the attribute, object and owner reductions always preserve the optimality of the solution* for the context table. However, this may not be the case in general since the next step that uses greedy heuristic would be applied if the table is not empty and none of the above reductions are present in the lattice, namely an interfering lattice.

Interference is said to be present in a concept lattice if for any concept c, there are at least two concepts c_i and c_j such that $c \leq_R c_i$ and $c \leq_R c_j$ and c_i and c_j are the neighbors of c in the lattice. In a lattice with interference, there are no object or attribute implications and also, no strongest concepts are labeled with attributes. Now we apply the greedy heuristic to select the test case that covers maximum requirements in the table and add it to the minimized suite. We remove the row corresponding to this test case from the context table and remove the requirements covered by this test case from the table. As a result of these modifications to the context table, some new object reductions may be enabled.

Since we delay applying the greedy heuristic to the point when no object, attribute or owner reductions can be applied, we refer to our algorithm derived from the above discussion as *Delayed-Greedy*. In contrast, the classical greedy algorithm [5, 6] applies the greedy heuristic at every step. Note that our algorithm is guaranteed to achieve *at least* as much reduction of the test suite size as possible using the classical greedy algorithm [5, 6]. Next we present the steps of Delayed-Greedy algorithm in detail.

3. OUR DELAYED GREEDY ALGORITHM

The concept lattice helped us develop the steps of our algorithm in terms of different reduction rules. Now that we know the types of implications we need to look for in the lattice or the context table, we can realize this algorithm by looking for these implications directly in the context table rather than constructing the concept lattice. Thus, we derive a new test suite minimization algorithm with worst case polynomial time complexity in terms of the size of the context table. The outline of our Delayed-Greedy algorithm is given in Figure 4.

Step 1: Reducing the size of context table by applying object reductions. An object implication exists in the context table if there are two test cases (rows) t_i and t_j such that the set of requirements covered by t_i is *superset* of the set of requirements covered by t_j. This can be found directly from the context table by comparing the sets of requirements covered by every pair of the rows in the table. In this case row corresponding to t_j is removed from the table. This is the application of object reduction rule to the context table. Reducing the context table by exploiting object implications can result in exposing new owner and attribute reductions.

Step 2: Reducing the size of context table by applying attribute reductions. An attribute implication exists in the table if there are two requirements r_i and r_j such that set of objects that cover r_i is a *subset* of the set of objects that cover r_j. In this case any test case that covers r_i will also cover r_j. Therefore, the requirement r_j is removed from the context table. This is the application of attribute reduction rule to the context table. The attribute implications are also found directly from the table by comparing the object-sets of each of pairs of requirements (columns). Notice that after a context table is reduced by applying attribute reductions, additional object implications that enable further reduction of the context table may be exposed.

Input: Context_table for given test suite T.
Output: Set of test cases in minimized suite T_{min}.
procedure Delayed-Greedy(Context_table)
  T_{min}=empty;
  while (Context_table $\neq$ empty) **do**
    fInter=false; detectInter=0;
    while not(fInter) **and** (Context_table $\neq$ empty) **do**
      fInter=true;
Step 1: **For** each *object implication* $o_i \Rightarrow o_j$ **do**
        Remove row for test case o_j from Context_table;
        fInter=false;
      endfor
Step 2: **For** each *attribute implication* $r_i \Rightarrow r_j$ **do**
        Remove column for requirement r_j from Context_table;
        fInter=false;
      endfor
Step 3: **For** each attribute r_k resulting in an *owner reduction* **do**
        Remove row for test case t that covers requirement r_k;
        Remove columns for attributes covered by test case t;
        T_{min} = $T_{min} \cup$ {t};
        fInter=false;
      endfor
    endwhile
Step 4:
    if (Context_table $\neq$ empty) **then**
      Let t be test case picked using greedy coverage heuristic.
      Remove row for test case t from the Context_table;
      Remove columns for attributes covered by test case t;
      T_{min} = $T_{min} \cup$ {t}; detectInter=1;
    endif
  endwhile
  if (detectInter=0) **then**
    report minimized test suite T_{min} is of optimal size.
  else
    report interference encountered.
  endif
  return(T_{min})
endprocedure

Figure 4: Delayed-Greedy algorithm

Note that we need to compare entries in every pair of rows(columns) to find object(attribute) implications only for the first time in the beginning of the algorithm. Every time an object(attribute) reduction is applied, the context table is updated by removing the corresponding row(column) from the table. Note that after the table is updated by removing a column(row), only the rows(columns) effected by the removed column(row) need to be checked for object(attribute) implication. Thus, after each reduction, we do not have to compare all the rows(columns) with each other to find a possible object(attribute) implication. This contributes significantly to the efficiency of our algorithm.

Step 3: Reducing the size of context table and selecting a test case using owner reduction. If there exists an attribute a_i that is possessed only by one object o_j, we add o_j to the solution set and remove o_j and all attributes covered by o_j from the table. Owner reductions may expose new object reductions which can further reduce the size of the context table and thus delay the need to apply the greedy heuristic. Note that *the attribute and object reductions merely reduce the size of the context table by removing redundant attributes and objects from further consideration, whereas the owner reduction also chooses a test case to be in the minimized suite*.

In each iteration of the algorithm, the owner reductions select those test cases that would eventually have to be included in the selected suite since they cover attributes not covered by other test cases in the context table. Therefore, the requirements covered by these test cases are removed from further consideration. However, in the classical greedy algorithm, selection of some test cases corresponding to owner reductions may be postponed to a later stage if they cover a small number of requirements. This may result in selection of some test cases early on by the classical greedy algorithm that may become redundant due to test cases selected later.

Step 4: Removing the interference by selecting a test case using the greedy heuristic. We choose the object that possesses the most number of attributes and add it to the selected set. We break the ties by as follows. For each attribute covered by each object, we compute the number of other objects that cover this attribute. We select the object covering an attribute that is least covered by all other objects. The reason for this strategy to remove interference is that if a test case covering maximum attributes is selected, the solution would be at least as good as that obtained by the classical greedy [5, 6] heuristic. The row corresponding to the selected test case and the columns corresponding to all the attributes covered by it are removed from the context table. This could give rise to further possible reductions of the context table by exposing new object implications. The variable $detectInter$ is set to indicate the greedy heuristic was used in the reduction. It is due to this heuristic the final solution may not be optimal. Note that lattices without interference give optimal solutions to the test suite minimization problem. The algorithm terminates when the context table is empty.

4. EXPERIMENTS

We implemented our Delayed-Greedy heuristic (DelGreedy), the classical Greedy heuristic, the HGS [8] algorithm and the SMSP [15] algorithm as C language programs. SMSP algorithm computes the reduced suite by selecting one test case each from the strongest concepts. For implementation of the SMSP algorithm, we computed the strongest concepts directly from the context table. We conducted experiments with the programs in the Siemens test suite [10, 11] and the *space* program [11] to measure the extent of test suite size reduction obtained by the above four heuristics.

We obtained these programs and their associated test pools from the the Subject Infrastructure Repository website [11]. We generated the instrumented versions of these programs using the LLVM infrastructure [17] to record the branch/def-use coverage informa-

Table 3: Experiment Subjects

Prog.	loc.	*Avg. size of un-minimized suite*		*Total No. of requirements*	
		Branch Cov.	Def-use Cov.	Branches	Def-use pairs
space	6218	533	539	1356	5179
tcas	138	20	21	41	51
print tokens	402	64	66	127	275
print tokens2	483	77	79	154	235
schedule	299	46	46	84	148
replace	516	83	108	155	759
totinfo	346	53	53	83	287

tion for each test case. The instrumented version generated by the LLVM infrastructure for the *schedule2.c* program resulted in segmentation faults when executed with test cases (whereas the un-instrumented version executed fine for the same test cases). Therefore, we conducted experiments with the *space* program and the remaining six programs in the Siemens suite. From the test pool for each program, we created 100 branch (def-use pair) coverage adequate test suites as follows. For each program, for each test suite, 10%-20% (5%-10% for the Space program since it is large) of the line of code test cases were randomly selected from the test pool, together with additional test cases as necessary to achieve 100% coverage of branches (def-use pairs). The number of lines of code, test pool size, average size of the un-minimized test suites for branch/def-use coverage and the total number of branches/def-use pairs in each program are shown in Table 3. We minimized each of these test suites using each of the above test suite minimization algorithms and recorded the size of the minimized test suite and time taken by each algorithm to minimize the test suite.

4.1 Results and Discussion

The average sizes of reduced suites produced by DelGreedy for each of the programs are shown in Table 5. In our experiments, for each test suite for each program, the size of minimized suite generated by DelGreedy was of the same size or of smaller size than that generated by the other algorithms. Therefore the numbers in Table 4 show for each program, for *how many* test suites (out of total 100), the difference between the size of reduced suite produced by other algorithm (Greedy, HGS or SMSP) and the size of reduced suite produced by DelGreedy was equal to 0, 1, 2, 3, etc. In other words, it shows the *frequency* with which the reduced suites for other algorithms were same size, larger by 1 test case, larger by 2 test cases, , , , larger by 9 test cases, etc. when compared with the size of reduced suites produced by DelGreedy. For example, for the minimization of branch coverage suites for the *tcas* program, the number 41 in the column labeled 1 and in the row corresponding to the Greedy algorithm shows that there were 41 test suites (out of total 100) for which the minimized suite by Greedy algorithm contained 1 more test case than the corresponding minimized suite generated by the DelGreedy algorithm. The Table 4 shows that DelGreedy, Greedy and HGS achieved more suite size reduction than SMSP and that DelGreedy can go even further than Greedy and HGS algorithms in producing smaller size suites.

Table 5: Average size of minimized suite by DelGreedy

Program	Branch Coverage	Def-Use Coverage
space	123	143
tcas	4	4
print-tokens	6	7
print-tokens2	4	8
schedule	2	2
replace	9	26
totinfo	2	5

Also note from the Table 5 that the average sizes of reduced test suites produced by DelGreedy are quite small and therefore the differences of sizes 1, 2, 3, , , 9 etc. in the reduced test suites produced by the other algorithms and DelGreedy are quite significant. In our experiments, on an average, for branch coverage adequate suites, DelGreedy produced smaller size suites than Greedy, HGS and SMSP in 35%, 64% and 86% of the cases respectively. On an average, for def-use coverage adequate suites, DelGreedy produced smaller size suites than Greedy, HGS and SMSP in 39%, 46% and 91% of the cases respectively.

Table 6: Number of *Optimal size* (#Opt) and *Non-Optimal size (#Non-Opt)* test suites produced by each algorithm and time performance

Prog.	Algo.	*Branch Coverage Suites*				*Def-Use Coverage Suites*			
		#Non-Opt.	#Opt.	#Un-Dec.	Time (sec)	#Non-Opt.	#Opt.	#Un-Dec.	Time (sec)
space	DelGreedy	-	92	8	.737	-	99	1	1.912
	Greedy	100	0	0	.444	100	0	0	1.932
	HGS	96	4	0	.307	93	7	0	.666
	SMSP	100	0	0	-	100	0	0	-
tcas	DelGreedy	-	68	32	.006	-	96	4	.006
	Greedy	43	37	20	.004	32	65	3	.004
	HGS	42	39	19	.002	2	94	4	.001
	SMSP	100	0	0	-	100	0	0	-
print tokens	DelGreedy	-	71	29	.006	-	92	8	.011
	Greedy	18	62	20	.005	22	70	8	.009
	HGS	57	35	8	.006	30	66	4	.008
	SMSP	100	0	0	-	100	0	0	-
print tokens2	DelGreedy	-	84	16	.010	-	80	20	.011
	Greedy	14	70	16	.007	38	50	12	.010
	HGS	51	29	20	.007	52	40	8	.008
	SMSP	100	0	0	-	100	0	0	-
schedule	DelGreedy	-	99	1	.003	-	91	9	.006
	Greedy	0	99	1	.003	0	91	9	.004
	HGS	63	36	1	.006	38	56	6	.008
	SMSP	1	99	0	-	34	66	0	-
replace	DelGreedy	-	53	47	.011	-	94	6	.027
	Greedy	51	25	24	.006	78	11	11	.021
	HGS	67	17	16	.006	71	28	1	.020
	SMSP	100	0	0	-	100	0	0	-
totinfo	DelGreedy	-	46	54	.004	-	88	12	.010
	Greedy	16	32	52	.004	2	87	11	.009
	HGS	73	11	16	.004	34	58	8	.008
	SMSP	99	1	0	-	100	0	0	-

Recall that unlike HGS, Greedy and SMSP algorithms, our DelGreedy algorithm can identify that a reduced suite is of optimal size if it was produced by using only the object, attribute and owner reductions. For each row labeled with DelGreedy in the Table 6, the column labeled #Opt shows the number (out of total 100) of reduced suites identified as of optimal size by DelGreedy. The rows corresponding to other algorithms for this column show, for how many of those suites identified as of optimal size by DelGreedy, did the algorithm compute same size suite as DelGreedy. The column labeled with #Non-Opt shows for how many test suites the other algorithms produced larger size test suites than those produced by DelGreedy. In these instances, it was clear that other algorithms generated non-optimal size suites. The data for this column can be computed by subtracting from 100, the number in the corresponding row under the column labeled with 0 in Table 4. The column labeled with 0 in Table 4 shows the number of test suites for which the respective algorithm computed the same size suites as those produced by the DelGreedy. Therefore, subtracting this number from 100 gives the number of test suites for which the algorithm definitely computed non-optimal size suite. The column labeled with #Un-Dec. in Table 6 shows, for each respective row, the number of test suites for which it could not be determined if the reduced suite was of optimal size or not. This can be computed for each row by subtracting from 100, the number test of suites that were definitely reduced to optimal size (#Opt.) and the number of test suites that were definitely reduced to non-optimal size (#Non-Opt.).

Note that for each row corresponding to algorithms Greedy, HGS and SMSP in Table 6, the sum of the entries under the columns labeled #Opt. and #Un-Dec. is equal to the corresponding entry under the column labeled 0 in Table 4. For example, for reduc-

Table 4: *Frequency of (size of* Tmin by Algo. - *size of* Tmin by DelGreedy *)* for Branch coverage and Def-Use coverage test-suites

Prog.	Algo.	*frequency of (size of* **Tmin by Algo.** - *size of* **Tmin by DelGreedy** *)*											*frequency of (size of* **Tmin by Algo.** - *size of* **Tmin by DelGreedy** *)*										
		Branch Coverage Suites											Def-Use Coverage Suites										
		0	1	2	3	4	5	6	7	8	9	> 9	0	1	2	3	4	5	6	7	8	9	>9
space	Greedy	0	4	10	20	22	22	12	8	2	-	-	0	1	3	5	19	17	24	16	8	6	1
	HGS	4	19	22	18	18	10	5	3	1	-	-	7	20	24	16	14	9	4	2	4	-	-
	SMSP	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	100
tcas	Greedy	57	41	2	-	-	-	-	-	-	-	-	68	30	2	-	-	-	-	-	-	-	-
	HGS	58	36	6	-	-	-	-	-	-	-	-	98	2	0	-	-	-	-	-	-	-	-
	SMSP	0	0	0	0	5	6	11	10	13	18	37	0	1	8	26	28	15	14	7	1	-	-
print tokens	Greedy	82	17	1	-	-	-	-	-	-	-	-	78	19	3	-	-	-	-	-	-	-	-
	HGS	43	36	15	4	2	-	-	-	-	-	-	70	22	6	2	-	-	-	-	-	-	-
	SMSP	0	0	0	0	0	0	2	3	2	2	91	0	0	0	0	0	0	1	2	5	7	85
print tokens2	Greedy	86	14	-	-	-	-	-	-	-	-	-	62	34	3	1	-	-	-	-	-	-	-
	HGS	49	44	17	0	-	-	-	-	-	-	-	48	34	15	3	-	-	-	-	-	-	-
	SMSP	0	0	0	2	1	0	0	1	1	1	94	0	0	0	0	0	1	1	2	0	0	96
schedule	Greedy	100	0	-	-	-	-	-	-	-	-	-	100	-	-	-	-	-	-	-	-	-	-
	HGS	37	48	14	1	-	-	-	-	-	-	-	62	32	6	-	-	-	-	-	-	-	-
	SMSP	99	0	1	-	-	-	-	-	-	-	-	66	19	12	2	1	-	-	-	-	-	-
replace	Greedy	49	45	4	2	-	-	-	-	-	-	-	22	36	38	10	3	1	-	-	-	-	-
	HGS	33	37	27	3	-	-	-	-	-	-	-	29	41	24	5	1	-	-	-	-	-	-
	SMSP	0	0	0	0	0	0	0	0	0	1	99	0	0	0	0	0	0	0	0	0	0	100
totinfo	Greedy	84	16	-	-	-	-	-	-	-	-	-	98	2	-	-	-	-	-	-	-	-	-
	HGS	27	47	20	5	1	-	-	-	-	-	-	66	29	4	1	-	-	-	-	-	-	-
	SMSP	1	2	8	12	20	12	17	12	9	4	3	0	0	0	0	1	2	2	7	11	12	65

ing the branch coverage suites for the *replace* program, in the row corresponding to the Greedy algorithm, the sum of entries under the columns labeled #Opt. and #Un-Dec. is 25+24=49 which is same as value in the corresponding row under the column labeled with 0 in the Table 4. Therefore, for 24 test suites in which the DelGreedy and Greedy computed same size suites, we could not determine if the size of the reduced suite is optimal since in these cases DelGreedy needed to use the greedy heuristic. However, if in each of these cases, if the Greedy algorithm had actually computed an optimal size suite, then DelGreedy would have also computed optimal size suite since both computed the same size reduced suits for these 24 test suites. Therefore, if it was the case that all the test suites counted in the undecided column (#Un-Dec.) for the row corresponding to Greedy algorithm for *replace* program indeed were reduced to the optimal size by Greedy algorithm, then still the difference in the number of optimal suites computed by DelGreedy and Greedy would be unchanged (53+24) - (25+24) = 53-25=28.

Therefore, the difference between the value of #Opt shown in the Table 4 for an algorithm (Greedy, HGS or SMSP) and the corresponding value of #Opt shown for DelGreedy gives the *minimum difference between the total number of optimal size suites generated by DelGreedy and the respective other algorithm*. The Table 6 clearly shows that DelGreedy can find significantly more number of optimal size reduced suites than the other algorithms. Note that we were able to do the above analysis without having to compute the optimal size suites for each of the test suites by using techniques like enumeration of suites which can run in exponential time. This analysis was made possible because of the property of DelGreedy to identify optimal size reduced suites in the cases when the reduced test suite could be generated without the need to apply the greedy heuristic.

Finally the column labeled with Time in 6 shows the average time taken by each algorithm in *seconds* to reduce test suites for each program. The time of SMSP is not shown as the implementation in [15] used a concept analysis tool to build the concept lattice to find the strongest concepts. However, we found the strongest concepts directly from the table without building the lattice. Since we used a different implementation to compute the same size reduced test suite as computed by SMSP, we have not given the time performance for SMSP. The Table 6 shows that the running time of DelGreedy is comparable to other algorithms.

5. RELATED WORK

Finding the minimal cardinality subset of a given test suite that covers the same set of requirements as covered by the original test suite is NP complete. This can be shown by a polynomial time reduction from the *minimum set-cover* [7] problem. Therefore, several heuristics have been developed to compute a solution that is of the size as close as possible to the optimal size solution. A classical approximation algorithm for the minimum set-cover problem by Chvatal [5, 6] uses a simple greedy heuristic. This heuristic picks the set that covers the most points, throws out all the points covered by the selected set, and repeats the process until all the points are covered. When there is a tie among the sets, one set among those tied is picked arbitrarily. This algorithm has been the most commonly cited solution to the minimum set-cover problem and an upper bound on how far it can be from the optimal size solution in the worst case has been analyzed in [6]. This heuristic exploits only the *implications among test cases* to determine which test cases become redundant while reducing a test suite. Another greedy heuristic, based on the number of test cases covering a requirement, was developed by Harrold et al.[8] to select a minimal subset of test cases that covers the same set of requirements as the un-minimized suite.

Agrawal used the notion of dominators, superblocks and megablocks [1, 2], to derive coverage implications among the basic blocks to reduce test suites such that the coverage of statements and branches in the reduced suite implies the coverage of the rest. Similarly, Marre and Bertolino [13] use a notion of entities subsumption to determine a reduce set of coverage entities such that coverage of the reduced set implies the coverage of un-reduced set. These works [1, 2, 13] exploit only the *implications among coverage requirements* to generate a reduced set of coverage requirements.

In contrast to the above work, our approach iteratively exploits the implications among the coverage requirements (attribute reductions) *and* the implications among the test cases (object reductions), in addition to the owner reductions, to derive a reduced suite and applies the greedy heuristic only when needed. This is in contrast to the classical greedy algorithm which applies the greedy heuristic at every step. Thus, our Delayed-Greedy algorithm is guaranteed to generate reduced suites that are of the same size or smaller size than those generated by the classical greedy algorithm. In essence, *while exploring a solution to the test suite minimization problem,* *we have discovered a new algorithm for the minimum set-cover problem*. Although, it does seem surprising that this algorithm has not been discovered before in various contexts in which the minimum set-cover problem may arise, it is easy to see that we were able to exploit different types of implications present in the context table because we started to analyze this problem with the help of concept analysis which exposed these different types of implications simultaneously in a single framework namely the concept lattice. Since the concept lattice is derived from the context table, all the desired implications can be derived directly from the context table, and this led to the development of our Delayed-Greedy algorithm.

Sampath et. al [15] have presented a concept analysis based algorithm (SMSP) for reducing a test suite for web applications. They consider the URLs used in a web session as the attributes and each web session as a test case. In this work, one test case from each of the strongest concept in the concept lattice is selected to generate a reduced test suite to cover all the URLs covered by the unreduced suite. As shown in our experiments and in their recent report [16], the reduced suites produced by their approach are in general larger than those produced by applying the classical greedy algorithm and the HGS algorithm to reduce a set of web user sessions. In our experiments, Delayed-Greedy algorithm always produced equal or smaller test suites than classical greedy algorithm [6, 5], the HGS algorithm [8] and the SMSP [15] algorithm.

The works in [14, 18] study the effects of test suite minimization on the fault detection capabilities of the reduced test suites. In [14], the HGS [8] algorithm is used for minimization of test suites selected from the Siemens suite [10] test pools. The test pools for the Siemens suite were generated to cover a wide range of requirements derived from black box testing techniques, white box techniques, and skills and experience of the researcher generating the test cases. Thus, the quality of the test suites selected from these test pools is high as they contain test cases to cover a wide range of requirements. Therefore, in the experimental studies reported in [14], a significant loss in the fault detection capability of the minimized suites was observed. In contrast, the experimental studies in [18] used ATAC [9] system to compute optimally minimized test suites from the randomly generated test suites. They conclude that minimization techniques can reduce the test suite size to a great extent without significantly reducing the fault detection capabilities of test suites. Although, these two studies seem to be contradictory, we believe that the fundamental reason for the different conclusions obtained in these two studies is the quality of the initial test suites used in detecting the faults experimented with.

Jones and Harrold have recently presented [12] some heuristics to minimize test suites specifically tailored for the MC/DC coverage criterion. However, our work presented in this paper is for reducing a test suite with respect to set of requirements which could be derived from any coverage criterion or a combination of different criteria. The only input to our algorithm is the context table which contains the information about the set of requirements covered by each test case in the test suite.

6. CONCLUSIONS

In this paper we presented a new greedy algorithm (Delayed-Greedy) to select a minimal cardinality subset of a test suite that covers all the requirements covered by the test suite. Our technique improves upon the prior heuristics by iteratively exploiting the implications and among the test cases *and* the implications among the coverage requirements, leveraged only independently from each other in the previous work. In our experiments, our technique consistently produced same size or smaller size test suites than prior heuristics.

7. REFERENCES

[1] H. Agrawal, “Dominators, super blocks, and program coverage,” *21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Portland, Oregon, 1994.

[2] H. Agrawal, “Efficient Coverage Testing Using Global Dominator Graphs,” *1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France, 1999.

[3] G. Birkhoff *Lattice Theory*, volume 5, American Mathematical Soc. Colloquium Publications, 1940.

[4] J. Black, E. Melachrinoudis and D. Kaeli, “Bi-Criteria Models for All-Uses Test Suite Reduction,” *26th International Conference on Software Engineering*, Edinburgh, Scotland, UK, 2004

[5] V. Chvatal. “A Greedy Heuristic for the Set-Covering Problem.” *Mathematics of Operations Research*. 4(3), August 1979.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein “Introduction to Algorithms”, *MIT Press*, Second Edition, September 2001.

[7] M.R. Garey and D.S. Johnson, “Computers and Intractability-A Guide to the Theory of NP-Completeness,” V Klee, Ed. Freeman, New York, 1979.

[8] M.J. Harrold, R. Gupta and M.L. Soffa, “A Methodology for Controlling the Size of a Test Suite,” *ACM Transactions on Software Engineering and Methodology*, 2(3):270-285, July 1993.

[9] J.R. Horgan and S.A. London, “ATAC: A data flow coverage testing tool for C,” in *Proceedings of Symposium on Assessment of Quality Software Development Tools*, pages 2-10, May 1992.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria,” *16th International Conference on Software Engineering*, May 1994.

[11] http://www.cse.unl.edu/∼galileo/sir

[12] J. A. Jones and M. J. Harrold, “Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage,” *IEEE Transactions on Software Engineering* , 29(3):195-209, March 2003.

[13] M. Marre and A. Bertolino, “Using Spanning Sets for Coverage Testing,” *IEEE Transactions on Software Engineering* , 29(11):974-984, Nov. 2003.

[14] G. Rothermel, M.J Harrold, J. Ostrin, and C. Hong, “An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites,” *International Conference on Software Maintenance*, November 1998.

[15] S. Sampath, V. Mihaylov, A. Souter and L. Pollock ”A Scalable Approach to User-Session based Testing of Web Applications through Concept Analysis,” in proceedings of *Automated Software Engineering, 19th International Conference on (ASE'04)* Linz, Austria, September 2004,

[16] S. Sprenkle, S. Sampath, E. Gibson, A. Souter, L. Pollock, ”An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications,” *Technical Report 2005-009*, Computer and Information Sciences, University of Delaware, November 2004

[17] ”The LLVM Compiler Infrastructure Project,” http://llvm.cs.uiuc.edu/

[18] W. E. Wong, J.R. Horgan, S. London, and A. P. Mathur. “Effect of Test Set Minimization on Fault Detection Effectiveness.” *Software Practice and Experience*. 28(4):347-369, April 1998.