A Comparative Study of Five Regression Testing Algorithms

Ghinwa Baradhi and Nashat Mansour

Computer Science
Lebanese American University
nmansour@lau.edu.lb

Abstract

We compare five regression testing algorithms that include: slicing, incremental, firewall, genetic, and simulated annealing algorithms. The comparison is based on the following ten quantitative and qualitative criteria: execution time, number of selected retests, precision, inclusiveness, user parameters, handling of global variables, type of maintenance, type of testing, level of testing, and type of approach. The experimental results show that the five algorithms are suitable for different requirements of regression testing. Nevertheless, the incremental algorithm shows more favorable properties than the others.

1. Introduction

Software maintenance involves modifying programs as a result of errors, or alterations in the user requirements [1]. During such modifications, new errors may be introduced, causing unintended adverse side effects in the software. Regression testing aims at providing confidence that the modifications are correct and have not adversely affected other parts of the program. Regression testing can be progressive or corrective [2]. The former involves retesting major changes to the program's specification. The latter is performed on a specification that essentially remains unchanged, so that only minor modifications, which do not affect the overall program structure, require retesting.

For regression testing, it would be costly to repeat the whole set of test cases used in the initial development of the program and unreliable to choose a random subset of these test cases. Therefore, it is important to select a suitable subset of test cases that accomplish the objectives of regression testing.

The selection of suitable test cases can be made in different ways, and a number of regression testing approaches and algorithms have been proposed. A strategy based on input partitioning and cause-effect graphing of the program specification has been described in [3]. Leung and White [2] have suggested classifying the initial test cases as reusable, retestable, obsolete, and adding new-structural and new-specification test cases and, then, selecting test cases from some of or all five classes. They have also proposed methods to limit integration regression testing to a small set of modules [4,5]. A slicing algorithm which decomposes a program and selects test cases to ensure that there is no undetected "linkage" between the modified and unmodified code has been presented in [6]. A strategy which combines data flow testing with incremental data flow analysis for unit and integration regression testing has been described in [7]. Harrold et al. [8] have also suggested a methodology for selecting a minimal number of retests that revalidates all requirements of a module. A safe algorithm which uses a module's dependence graph and selects retests that will cause the modified module to produce different output than the original module has been proposed in [9]. Path expressions are used in [10] to represent the program and then algebraic operations are used to determine the paths affected by the modifications. This facilitates the selection of the initial test cases needed for restesting. A similar approach based on semantic differencing has been proposed in [11]. An optimal retesting approach based on a 0-1 integer programming problem formulation has been proposed by Fischer [12] and extended in [13] and solved by natural optimization in [14]. Some software tools, which are based on some of the afore-mentioned techniques, have been constructed [15,16].

Evaluating and comparing regression testing algorithms has received little attention. A cost model has been suggested in [17] and a theoretical analysis framework has been recently presented in [18]. In this paper, we present an experimental comparative study of
five representative regression testing algorithms, which include: slicing, incremental, firewall, genetic, and simulated annealing algorithms. The comparison criteria used are: (i) algorithm’s execution time, (ii) the number of test cases selected for regression testing, (iii) precision, (iv) inclusiveness, (v) whether the algorithm includes parameters to be set by the user, (vi) handling of global variables, (vii) type of maintenance, (viii) type of testing, (ix) level of testing, and (x) type of approach. The experimental results show that the five algorithms have diverse properties for different criteria.

This paper is organized as follows. Section 2 describes the regression testing problem and the program models used. Section 3 gives a brief description of the five implemented algorithms. Section 4 explains the comparison criteria. Section 5 presents and discusses the experimental results. Section 6 contains conclusions.

2. Program modeling and regression testing problem

We assume that a program is modeled by a control flow graph with M nodes. Each node represents a program segment, which corresponds to a control statement or to a contiguous sequence of assignment statements. The graph edges represent control dependence. Also the data flow information is stored for relevant nodes, which includes the definitions and uses of the variables in the program statements within segments. Uses are classified as either computation uses (c-uses) or predicate uses (p-uses) according to whether a variable is used in a computation statement or to a contiguous sequence of assignment statements. A definition that reaches a use forms a definition-use pair.

We also assume that the set, \( T = \{t_1, t_2, t_3, \ldots, t_N\} \), of \( N \) test cases used in the initial development of the program is saved and that a table of test case-segment coverage information can be determined. After a program is modified, regression testing requires that a subset of test cases, \( R \), be selected from \( T \) for rerunning on the modified program with the objective of providing confidence that no adverse effects have been caused by the modification. The selection of \( R \subseteq T \) is normally guided by criteria such as minimum-cardinality of the subset \( R \) or testing different data effects. A number of such criteria are given in Section 4.

3. Implemented algorithms

In this section, we briefly present the five algorithms that we have implemented for our comparative study.

3.1. Slicing algorithm

We have implemented Gupta, Harrold, and Soffa’s algorithm [19] which is data flow based. It uses slicing to explicitly detect definition-use pairs that are directly or indirectly affected by a program modification. A program slice consists of all statements/segments, including conditionals, that might affect the values of variables in the modified statement/segment. The advantage of this slicing approach is that no data flow history is needed, nor is the recomputation of data flow for the entire program required after its modification. The algorithm uses backward and forward walk procedures in order to determine the program slice associated with the modified statement/segment. The selected test cases will be all those that traverse the slice.

3.2. Incremental algorithm

We have implemented Agrawal, Horgan, and Krauser’s incremental algorithm [20] which selects test cases from \( T \) whose outputs may be affected by the modifications made to the program.

The incremental approach exploits the following observations: (i) Not all statements in the program are executed under all test cases, (ii) If a statement is not executed under a test case, it can not affect the program output for that test case, (iii) Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case, and (iv) Every statement does not necessarily affect every part of the program output. These observations can be used to determine execution, dynamic, and then relevant slices for the test cases. A relevant slice contains the statements/segments, including predicates, in an execution slice (traversed by) of a test case that have an influence on an output statement in the slice.

3.3. Adapted firewall algorithm

Leung and White [5] have proposed the concept of “firewall” to assist the tester in integration regression testing. For our study’s purpose, we have implemented the “firewall” concept at the segment (vs. module) level, and hence the name adapted firewall algorithm.

Leung and White suggest building “firewalls” to confine integration regression testing to a small set of modules rather than allowing it to spread to many other modules. A “firewall” involves the modules that are modified and their direct ascendants and direct descendants.

The adapted firewall is based on the control flow graph. A “firewall” is built around the modified
segments and other related segments. Only those segments within the "firewall" need to be regression tested by test cases selected from \( T \). The construction of the "firewall" is made at the boundaries of the segments tested by test cases selected from segments and other related segments. Only those segments within the "firewall" need to be regression tested by test cases selected from \( T \).

One of the following ways: (i) by being direct ascendants of definitions affecting the program modification, or (ii) by being direct descendants where uses are affected by the modification.

3.4. Genetic algorithm

We have implemented a genetic algorithm proposed in [14], which is based on a population of individuals. An individual in the population is encoded as an \( N \)-element vector \([X_1, X_2, \ldots, X_N]\) that corresponds to a candidate solution for the regression testing problem. An element (gene) \( X_i = 1 \) (or 0) indicates the inclusion (or exclusion) of test case \( t_i \in T \) in the selected subset of retests, \( R \). The fitness of an individual to be maximized by the genetic algorithm is given by \( 1/Z \), where

\[
Z = X_1 + X_2 + \ldots + X_N
\]

subject to the constraints: \( \sum_{j=1}^{N} a_{ij}X_j \geq b_i; \quad i = 1, \ldots, M; \)

where \( a_{ij} \) is an element of the test-segment coverage table; \( b_i \) indicates whether segment \( i \) is reachable from the modified segment and, thus, needs to be covered by the subset of retests.

The initial population of individuals is randomly generated and then evolves over a number of generations, until convergence. A new generation is created by allocating reproduction trials to the individuals according to their fitness, and then randomly selecting pairs of surviving individuals and applying genetic operators to each pair to create their offspring. The algorithm is hybridized by feasibilizing and hill-climbing procedures. At the end of the evolution, an optimal subset of retests would correspond to the maximum fitness of feasible individuals.

3.5. Simulated annealing algorithm

We have implemented a simulated annealing algorithm suggested in [14], where a candidate solution is represented by the configuration \([X_1, X_2, \ldots, X_N]\) and the energy to be minimized is given by the cost function \( Z \) in Equation 1. The algorithm starts with a random initial configuration at a high (artificial) temperature and reduces the temperature gradually to a freezing point. At each temperature, regions in the solution space are searched by the Metropolis algorithm.

An iteration of the Metropolis algorithm starts with proposing a random perturbation to the configuration and evaluating the resultant change in the energy of the system. If the change is negative (downhill move) or zero, the perturbation is accepted and the new lower-energy configuration becomes the next starting point. If the energy change is positive (uphill move), the proposed perturbation may be accepted with a temperature-dependent probability to prevent the system from being trapped in a bad local minimum-energy state. The algorithm incorporates a feasibilization technique for downhill moves to focus the search in the feasible regions of the solution space. At the freezing point, the configuration yields a feasible optimal or near-optimal subset of retests.

4. Comparison Criteria

In this section we present ten quantitative and qualitative criteria used for evaluating and comparing the five regression testing algorithms.

The quantitative criteria are:

(i) Execution time of a regression testing algorithm, denoted as \( T_{exec} \).

(ii) Number of test cases in \( R \) selected by an algorithm, from \( T \), to be rerun for regression testing, which is denoted as \#R.

(iii) Precision, which is defined in terms of modification-revealing and modification-traversing tests. Modification-revealing tests are those that produce different outputs for the modified program from those for the original version. modification-traversing tests are those that execute modified code. Obviously, not all modification-traversing tests are modification-revealing.

Precision measures the ability of an algorithm to omit non-modification-revealing tests that will not cause the modified program to produce different output [18]. If the initial test suite \( T \) contains \( n_1 \) non-modification-revealing tests, and a regression testing algorithm selects \( (n_r,m_r) \) of these tests, then the precision of the algorithm is given by the ratio \( (m_r/n_r) \).

(iv) Inclusiveness, which measures the extent to which a regression testing algorithm chooses modification-revealing tests that will cause the modified program to produce different output [18]. If the test suite \( T \) contains \( n_2 \) modification-revealing tests, and a regression testing algorithm selects \( m_2 \) of these tests, then the inclusiveness of the algorithm is given by the ratio \( (m_2/n_2) \).

The qualitative criteria are:

(v) User's parameter setting: this refers to whether a regression testing algorithm requires user's intervention to set some of its parameters.
(vi) **Testing of global variables:** this refers to the ability of an algorithm to test global variables. Global variables are considered since they create data flow dependences between modules that may not directly callable.

(vii) **Type of maintenance:** software maintenance can be [2]: (i) corrective, due to error reports, (ii) perfective, for enhancing the software’s functionality, (iii) adaptive, due to changes in the external environment, or (iv) preventive, for facilitating future maintenance. In this work, we classify the regression testing algorithms according to their use for corrective and/or perfective type of maintenance.

(viii) **Type of testing:** this refers to whether a regression testing algorithm assumes that the initial testing done during the program development was functional or structural [21].

(ix) **Level of testing:** this refers to whether an algorithm is suitable for integration regression testing or only at the module level.

(x) **Type of approach:** regression testing approaches can be classified as coverage, minimization, or safe. Algorithms that select only modification-traversing tests are coverage algorithms. Algorithms that aim for minimizing the number of selected retests and, thus, the regression testing time are minimization algorithms. Algorithms that always select all modificationrevealing tests are safe algorithms.

5. Experimental results and discussion

In this section, we experimentally and qualitatively compare the properties of the five algorithms.

5.1. Quantitative criteria results

The experiments were done on a PC with an INTEL CPU486 DX4 100 MHz. Twenty different program modules (m1-m20) were used, for which the codes were written and the graphs were manually generated along with the test cases. These modules are of small sizes (m1-m14) and of medium sizes (m15-m20). With each module are associated: the number of segments, \( M \), the number of the modified segment, \( S_{mod} \), and a table of \( N \) test cases and their segment coverage information.

5.1.1. Execution time and number of selected retests.

Table 1 gives the execution times, \( t_{exec} \), in seconds, and the number of selected retests, \( \#R \), for the five algorithms on the 20 modules. A typical sample of these results are illustrated in Figures 1 and 2. These results show that for small-size modules, the slicing and adapted firewall algorithms exhibit similar behavior. For some modules, they select a number of retests close to the retest-all strategy. The incremental algorithm yields a better number of retests for similar execution times. The genetic and simulated annealing algorithms select the least number of retests, although they are slower.

For medium-size modules, the adapted firewall algorithm takes the greatest time with a number of retests close to that of the slicing algorithm. The incremental algorithm gives better number of retests, at an execution time close to that of the slicing algorithm. The best number of retests, is consistently offered by the genetic and simulated annealing algorithms, although with a slow execution. We also note that the number of retests selected by the incremental algorithm is close to that of the genetic and annealing algorithms.

5.1.2. Precision and inclusiveness. Figures 3 and 4 present the results obtained for precision and inclusiveness of the five implemented algorithms on 8 modules. Clearly, the incremental, genetic, and simulated annealing algorithms give 100% precision, i.e. tests that will not cause the modified program to produce different output are entirely avoided. The slicing and the adapted firewall algorithms have lower precision. This is due to the fact that these two algorithms define all the def-use pairs that may be affected by the modification, which results in selecting additional test cases that are non-modification-revealing.

Inclusiveness is high for the slicing and adapted firewall algorithms. These algorithms select only modification-traversing tests by selecting tests that exercise definition-use pairs. However, they may fail to identify tests that execute modified output statements that contain no variable uses, although these statements may cause the program to produce different outputs. Also, these algorithms cannot detect tests that include deleted statements of a modified program. Thus, depending on the modification, their inclusiveness varies for different modules.

The inclusiveness of the incremental algorithm varies for different modules, but is generally lower than that for the slicing and adapted firewall algorithms. The incremental algorithm aims for selecting the test cases on which the new and the old programs may produce different outputs. However, not all of these test cases are included; if several tests exercise a particular affected statement, only some of them are selected depending on the control dependence of the statements.

The genetic and simulated annealing algorithms exhibit lower inclusiveness than the other algorithms; if several tests exercise a particular affected statement, they aim for selecting only one such test.
Table 1. The execution times (sec) and the number of selected retests for the five algorithms.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>0.3</td>
<td>5</td>
</tr>
<tr>
<td>m2</td>
<td>8</td>
<td>32</td>
<td>8</td>
<td>0.8</td>
<td>26</td>
</tr>
<tr>
<td>m3</td>
<td>10</td>
<td>15</td>
<td>8</td>
<td>0.3</td>
<td>5</td>
</tr>
<tr>
<td>m4</td>
<td>10</td>
<td>27</td>
<td>10</td>
<td>0.5</td>
<td>8</td>
</tr>
<tr>
<td>m5</td>
<td>14</td>
<td>24</td>
<td>13</td>
<td>1.4</td>
<td>6</td>
</tr>
<tr>
<td>m6</td>
<td>14</td>
<td>32</td>
<td>14</td>
<td>1.5</td>
<td>32</td>
</tr>
<tr>
<td>m7</td>
<td>21</td>
<td>9</td>
<td>12</td>
<td>1.3</td>
<td>5</td>
</tr>
<tr>
<td>m8</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>2.0</td>
<td>11</td>
</tr>
<tr>
<td>m9</td>
<td>24</td>
<td>18</td>
<td>9</td>
<td>0.9</td>
<td>5</td>
</tr>
<tr>
<td>m10</td>
<td>24</td>
<td>23</td>
<td>12</td>
<td>1.3</td>
<td>5</td>
</tr>
<tr>
<td>m11</td>
<td>34</td>
<td>58</td>
<td>29</td>
<td>5.6</td>
<td>12</td>
</tr>
<tr>
<td>m12</td>
<td>34</td>
<td>126</td>
<td>32</td>
<td>7.0</td>
<td>29</td>
</tr>
<tr>
<td>m13</td>
<td>40</td>
<td>20</td>
<td>21</td>
<td>2.3</td>
<td>7</td>
</tr>
<tr>
<td>m14</td>
<td>40</td>
<td>28</td>
<td>39</td>
<td>4.6</td>
<td>9</td>
</tr>
<tr>
<td>m15</td>
<td>45</td>
<td>32</td>
<td>27</td>
<td>2.9</td>
<td>4</td>
</tr>
<tr>
<td>m16</td>
<td>45</td>
<td>96</td>
<td>43</td>
<td>7.0</td>
<td>16</td>
</tr>
<tr>
<td>m17</td>
<td>56</td>
<td>34</td>
<td>41</td>
<td>5.5</td>
<td>8</td>
</tr>
<tr>
<td>m18</td>
<td>56</td>
<td>108</td>
<td>50</td>
<td>9.4</td>
<td>22</td>
</tr>
<tr>
<td>m19</td>
<td>60</td>
<td>90</td>
<td>30</td>
<td>4.9</td>
<td>15</td>
</tr>
<tr>
<td>m20</td>
<td>60</td>
<td>145</td>
<td>46</td>
<td>9.2</td>
<td>22</td>
</tr>
</tbody>
</table>

Figure 1. The execution times for the five algorithms for modules m16-m20.
Figure 2. The number of selected retests for the five algorithms for modules m16-m20.

Figure 3. The precision of the five algorithms for selected modules.
5.2. Qualitative criteria results

5.2.1. User’s parameter setting. The genetic algorithm requires parameters to be set by the user, namely the population size, the probability range for reproduction, convergence threshold, crossover rate, and mutation rate. The annealing algorithm requires the parameters: convergence threshold, and initial probability for accepting uphill moves, the cooling rate, and the number of the Metropolis iterations.

5.2.2. Testing global variables. Leung and White [22] have shown that global variables can be treated as extra parameters and that they can be regression tested by the firewall algorithm. However, the five algorithms considered in this work can use an additional variable’s set/use matrix to only identify global variables within modules.

5.2.3. Type of maintenance. All five algorithms evaluated in this work are corrective regression testing algorithms. However, the firewall concept at the integration level supports both, corrective and perfective regression testing. Also, the incremental algorithm is both corrective and perfective; actions such as deletion and addition of program segments to enhance functionality can be handled. The other algorithms require further adaptation to make them suitable for perfective regression testing.

5.2.4. Type of testing. The program models used in the slicing, adapted firewall, genetic, and simulated annealing algorithms are based on the internal structure and logic of the program. The incremental and the integration-firewall algorithms can be both structure- and function-based.

5.2.5. Level of testing. All five algorithms have been applied at the module level, except for the firewall algorithm which is initially an integration level algorithm. Integration testing cannot be applied using slicing since it is based on identifying the definition-use pairs at the module level. In incremental regression testing, computation of the execution slices can be done at the module or integration level. At the integration level, a test case will be selected only if it invokes a modified function or module. For the genetic and simulated annealing algorithms, segments can be replaced by modules which enables the algorithms to work at the program level.

5.2.6. Type of approach. The slicing and firewall algorithms, are coverage algorithms, since they select tests that cover affected def-use pairs; at the integration
level the firewall algorithm selects tests that cover affected modules. The incremental algorithm is a safe regression testing algorithm, since it aims at selecting tests that will cause the modified program to produce different output than the original program. The genetic and simulated algorithms are minimization algorithms; if several tests exercise a particular modified segment, only one such test is selected.

5.3. Summary of results

Table 2 summarizes the comparison results for the quantitative and qualitative criteria. We note that our assessment is based on the following considerations:
(i) Medium size modules are more important for assessment, since they are more realistic.
(ii) Since the test cases were manually developed, it was not possible to run experiments that were statistically highly-sound, especially for the execution time.
(iii) Execution time assessment (slow/fast) is based on comparing the algorithms with each other.

6. Conclusions

An experimental comparative study, based on ten quantitative and qualitative criteria, have been presented for five regression testing algorithms using 20 program modules of different sizes.

The choice among the five algorithms is dependent on the regression tester’s requirements. For example, to test all affected definition-use pairs, despite spending more regression testing time, the algorithms to choose would be slicing and adapted firewall. To choose a minimum number of test cases and, hence, to perform fast regression testing, the selection would be the genetic or simulated annealing algorithms, although they themselves are slow. The incremental algorithm would be the best choice for selecting a number of test cases whose outputs may be affected, while being the fastest among the five algorithms. However, an overall assessment, based on the ten criteria and the examples used in this work, tends to indicate that the incremental algorithm has more favorable properties than the other four algorithms.

Acknowledgement. The first author would like to thank the Arban Investment Computing Company, Arab Bank PLC, for supporting her work in part.

References
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Slicing</th>
<th>Incremental</th>
<th>Adapted firewall</th>
<th>Firewall at Integration</th>
<th>Genetic</th>
<th>Simulated Annealing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time</td>
<td>fast</td>
<td>fast</td>
<td>fast</td>
<td>acceptable</td>
<td></td>
<td>acceptable</td>
</tr>
<tr>
<td>small-size modules</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution time</td>
<td>acceptable</td>
<td>fast</td>
<td>slow</td>
<td>acceptable</td>
<td></td>
<td>slow</td>
</tr>
<tr>
<td>medium-size modules</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of</td>
<td>acceptable</td>
<td>good</td>
<td>acceptable</td>
<td>very good</td>
<td></td>
<td>very good</td>
</tr>
<tr>
<td>selected retests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>high</td>
<td>very high</td>
<td>high</td>
<td>very high</td>
<td></td>
<td>very high</td>
</tr>
<tr>
<td>Inclusiveness</td>
<td>high</td>
<td>medium</td>
<td>high</td>
<td>low</td>
<td></td>
<td>low</td>
</tr>
<tr>
<td>User’s parameter</td>
<td>not required</td>
<td>not</td>
<td>not required</td>
<td>not required</td>
<td>required</td>
<td>required</td>
</tr>
<tr>
<td>setting</td>
<td></td>
<td>required</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global variables</td>
<td>can be identified</td>
<td>can be identified</td>
<td>can be tested</td>
<td>can be identified</td>
<td></td>
<td>can be identified</td>
</tr>
<tr>
<td>Type of maintenance</td>
<td>corrective</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>perfective</td>
<td>with modification</td>
<td>yes</td>
<td>with addition</td>
<td>yes</td>
<td>with addition</td>
<td>with addition</td>
</tr>
<tr>
<td>Type of testing</td>
<td>structural</td>
<td>structural</td>
<td>functional</td>
<td>structural</td>
<td>structural</td>
<td>structural</td>
</tr>
<tr>
<td>Level of testing</td>
<td>Module</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Testing</td>
<td>Integration</td>
<td>no with modification</td>
<td>yes</td>
<td>with modification</td>
<td>with modification</td>
<td></td>
</tr>
<tr>
<td>Type of approach</td>
<td>coverage</td>
<td>safe</td>
<td>coverage</td>
<td>coverage</td>
<td>minimization</td>
<td>minimization</td>
</tr>
</tbody>
</table>

Table 2. Summary of the properties of the five regression testing algorithms.


