CHAPTER 6

TEST SUITE REDUCTION USING
DATA MINING APPROACH

In the previous chapters attempts have been made in proposing test suite reduction algorithms in the area of evolutionary computing, greedy concept and graph based heuristic. Another popular approach among researchers for the optimization problem is the data mining techniques (Ali & Golnoosh 2011). Mining frequent itemsets is a fundamental and essential concept where most of the proposed itemset mining approaches are a variant of Apriori algorithm (Agrawal et al 1996, Karam & Mohammed 2005). However, most of the frequent itemset mining approaches lead to an exponential number of pattern generations thereby making computation infeasible. This has invariantly motivated researchers to develop new approaches to mine frequent itemsets. In this chapter an algorithm called Maximal Frequent Test Set (MFTS) has been proposed as an extension of mining frequent test sets.

6.1 INTRODUCTION TO DATA MINING

Data mining is an extension of traditional data analysis and statistical approaches that incorporates analytical techniques drawn from a range of disciplines. It can be employed as a data driven approach that encourages pattern detection algorithms to find useful trends, patterns and relationships.
Data mining systems employ and provide different techniques. The classification of data mining systems according to the data analysis approach are as follows:

- Machine Learning
- Neural Networks
- Genetic algorithms
- Statistics
- Visualization
- Database-oriented

Such classification can also take into account the degree of user interaction involved in the data mining process such as query-driven systems, interactive exploratory systems, or autonomous systems.

When items occur together it can be associated to each other using data mining techniques. Such items form a frequent itemset. In frequent itemset mining terminology the term support refers to the percentage of task-relevant data transactions for which the pattern is true. The maximal frequent items from a given itemset can be obtained through standard approaches like:

- Search Strategies
- Pruning Strategies
- Dynamic Reordering
- Data Representation for Fast Support Counting
- Frequency Determination
However such approaches lead to an exponential number of pattern generation which may become computationally infeasible. This has invariably motivated researchers to develop new approaches to mine frequent itemsets. Some of the popular Maximal Frequent Itemset (MFI) algorithms in literature are represented in Figure 6.1 and also briefed next.

Ansaf et al (2002) proposed a new approach based on the properties of boolean algebra using truth tables. This approach explores and mines frequent itemset using a Binary Decision Diagram (BDD). Another maximal frequent itemset approach known as MAXimal Frequent Itemsets Algorithm (MAFIA) uses the lexicographic subset tree (Douglas et al 2005). This approach uses depth-first traversal of the tree where the tail contains all items lexicographically larger than any element of the head. MAFIA also uses a vertical bitmap representation for support counting along with effective
pruning mechanisms for searching the itemsets. In another approach FP-Tree (Frequent Pattern) data structure was adapted as a new strategy called “NBN” by Shiguang & Chen (2008). However the major drawback of FP tree is the memory consumption and issues pertaining to its construction.

The proposed MFTS algorithm adapts data mining concept to determine the maximum frequent test sets in the universal test suite to solve the optimal representative set selection problem. The test metrics such as: size, requirement coverage, fault defection and execution time have been evaluated in this work.

6.2 PROPOSED MFTS ALGORITHM

Data Flow Testing: One common approach to structural testing of software programs is to design and select test cases according to control flows of software programs. Data flow testing is one such testing technique based on values associated with variables and its effect on program execution. Data flow testing not only explores program control flows but also pays attention to how a variable is defined and used at different places along control flows. The association between the definitions and uses of each data member yields different define-use paths. Such unique define-use path represents a test case. Each unique define-use variable pair in a path constitutes a requirement and its associated test cases.

Test Suite Reduction: The test suite optimization problem can be formulated as follows: Given the universal test suite, T for a program comprising of test sets T_i, find all frequent test sets. A frequent test set is one that occurs for at least a user-specified support value in the universal test suite, T. The optimal subset of the test suite that covers all the requirements is known as the representative set, T_r. The goal of test suite reduction for a given set of test requirements is to produce a representative set that is smaller than the original
test suite’s size thereby satisfying the original test suite’s test requirement (Harrold et al 1993, Ying et al 2012).

**Test Data Coverage:** Software testing also includes activities to determine the effectiveness and quality of test suites considered. The general intuition is to use coverage criterion to decide when a program is sufficiently tested and the testing activity should stop. Test suite reduction techniques must also ensure that the reduced test suite should achieve the same coverage as the original test suite.

**Fault Detection:** The process of software testing and fault detection capability of test cases continues to challenge the software community. A fault is a manifestation of an error. A fault, if encountered, may cause a failure (IEEE 1998). Generally, when faults are found in the early stages of software development it is easier to remove and also cheaper in cost. Both identification and removal of faults are crucial for the success of a software. According to the literature survey findings by Xia & Michael (2005) the effect of code coverage has a direct impact on the fault detection effectiveness.

**Execution Time:** The acceleration of test suite reduction approach is vital in software testing. Iterative or recursive test suite reduction processes may invariably slow down the test activity. The time complicacy of the test suite is calculated as $\log_{10}(mn)$ where $m$ refers to the number of requirements that the representative set should satisfy and $n$ the number of test cases in the test suite (Hao Zhong et al 2006).

**MFTS Algorithm**

Mining frequent itemsets is an active research area and has been extended to mine maximal frequent itemsets by Rajalakshmi (2012). This data mining approach for frequent item sets has been adapted and proposed as Maximal Frequent Test set (MFTS) algorithm. The proposed algorithm uses
terminologies in particular to determine test set size (cardinality), test case cardinality (Global test case array) and common test case pattern among test sets. Selecting frequent test sets based on support value to generate an optimal test suite is the focus of this proposed algorithm.

- MFTS Initialization
- MFTS Mining

**MFTS Initialization**

The test suite reduction using algorithm 6.1 begins with the initialization of the associated tables and vectors. Each test set \( T_i \) represents a requirement \( r_i \). The cardinality of each test set, \( T_i \) is determined as in Equation 6.1. Similarly the cardinality of each individual test case \( t_j \) in all the test sets \( T_i \) is placed in the array \( TC_i \) and computed as in equation 6.2. The test sets \( T_i \) is sorted in descending order of cardinality and placed in the MFTS_Source table. The cardinality of each individual test case \( t_j \) in all the test sets \( T_i \) are identified and the value of global test case array (GTC) is computed as in equation 6.3. Finally the min_support representing the minimum cardinality is computed (Equation 6.4).

\[
\text{Card}(T_i) = \sum_{j=1}^{m} t_j
\]  

(6.1)

**Algorithm 6.1: MFTS Initialization**

**Input:** \( T_i \)

**Output:** MFTS_Source table, Global vector GTC, Vector \( TC_i \)

**begin**

\{//Construct the source table\}

Create a table MFTS_Source and place all \( T_i \)'s
//Consider each test set
for each T_i do
//Determine the cardinality of each test set
compute \( \text{Card}(T_i) = \sum_{i=1}^{n} \text{Card}(t_i) \)

//Determine the count of the individual unique test cases for the test set considered
\[ TC_i = \sum_{i=1}^{n} \text{Card}(t_1), \sum_{i=1}^{n} \text{Card}(t_2), \ldots, \sum_{i=1}^{n} \text{Card}(t_m) \]

//Place the test set in the source table along with it's details in the descending order of cardinality
insert T_i along with cardinality and TC_i into MFTS_Source table in the descending order of cardinality

// Compute the count of each individual test case in all the test sets
\[ GTC = \sum_{j=1}^{m} \text{Card}(t_1), \sum_{j=1}^{m} \text{Card}(t_2), \ldots, \sum_{j=1}^{m} \text{Card}(t_m) \]
endfor

//Finally determine the minimum support as the minimum test set size
\[ \text{min} \_ \text{support} = \text{MIN}(\text{Card}(T_i)) \]
\}

end

\[ TC_i = \sum_{i=1}^{n} \text{Card}(t_1), \sum_{i=1}^{n} \text{Card}(t_2), \ldots, \sum_{i=1}^{n} \text{Card}(t_m) \quad (6.2) \]

\[ GTC = \sum_{j=1}^{m} \text{Card}(t_1), \sum_{j=1}^{m} \text{Card}(t_2), \ldots, \sum_{j=1}^{m} \text{Card}(t_m) \quad (6.3) \]

\[ \text{min} \_ \text{support} = \text{MIN}(\text{card}(T_i)) \quad (6.4) \]
**MFTS Mining**

After initialization of MFTS variables: tables and vectors, Algorithm 6.2 begins by retrieving each test set, \( T_i \) from MFTS_Source table and placing it in the MFTS table for frequent test set mining. Then the support(\( T_i \)) for each test set is set as ‘1’ when it is placed in MFTS table. This is followed by recomputing the value of GTC as in equation 6.5. Within a loop the test sets in MFTS table are checked if they are not disjoint. If any test set is not disjoint the corresponding support value is incremented by ‘1’. Then the support value is determined if it is greater than or equal to min_support. If any test set is maximal frequent, the corresponding test set \( T_i \) is placed in the table \( T_{rs} \). The occurrence of the test set \( T_i \) placed in the table \( T_{rs} \) is removed from the table MFTS. After a test set \( T_i \) is added to the MFTS table if any element(s) in, GTC becomes ‘0’ then the corresponding test case \( t_i \) is infrequent. The occurrence of that test case (s) is pruned as in equation 6.6 from all the test sets in MFTS table.

\[
GTC = GTC - TC_i
\]  

(6.5)

\[
T_i = \text{remove } t_j \text{ in } T_i, \forall \ T_j \in \text{MFTS}
\]  

(6.6)

**Algorithm 2: MFTS Mining**

**Input:** MFTS_Source table, min_support, Global vector GTC, Vector TC

**Output:** \( T_{rs} \) table

**begin**

\{
//Retrieve each test set placed in the source table
for each \( T_i \) in MFTS_Source do
//Initialize the support value of the test set as one
    support(\( T_i \)) := 1
//Place the test set in a temporary table
MFTS := MFTS \cup T_i

//Recompute the value of GTC after including the test set
compute \ GTC=GTC-TC_i

//Determine the common test cases in the temporary table
for each T_i in MFTS do
    if (T_i \cup T_i+1) are not disjoint
        // increment the support value of the subset
        support(T_i)=support(T_i)+1
    endif
endfor

for each T_i in MFTS do
    //Check for maximal frequent test set
    if (support(T_i)>\=min_support)
        //If a test set is maximal frequent then add the test set in the representative set
        T_rs:= T_rs \cup T_i
    endif
endfor

for each t_j in GTC do
    //Determine if the cardinality of any test case has become zero
    if (t_j==0)
        //Prune the occurrence of the test case t_j that is infrequent from all the test sets in MFTS table
        T_i := remove t_j in T_i
    endif
endfor
}
end
The above process iterates till all the elements of GTC become ‘0’ or there are no more maximal frequent test sets. The resulting $T_{rs}$ table contains the reduced test suite of maximal frequent test sets. As $m$ test sets are added to the MFTS source table in a linear manner the computational complexity becomes $O(m)$ for test suite reduction.

**Concept Illustration**

To illustrate the proposed MFTS algorithm and the state-of-the-art algorithms namely HGS (Appendix 2) and BOG (Appendix 2), a hypothetical program has been considered. The test cases $t_j$s are generated for the program considered based on all possible flow of data from the declaration to the assignment of variables and depicted in Table 6.1. In Table 6.1, $B_1$, $B_3$ are blocks consisting of definition/redefinition of variables and $B_2$, $B_5$, $B_6$ refer to blocks containing the use of the variables. From the possible flow of data

<table>
<thead>
<tr>
<th>TIDs</th>
<th>DU pair</th>
<th>Test Sets ($T_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$(B_1,B_2)$</td>
<td>${t_1,t_2,t_4,t_5}$</td>
</tr>
<tr>
<td>2</td>
<td>$(B_1,B_5)$</td>
<td>${t_2,t_3,t_5}$</td>
</tr>
<tr>
<td>3</td>
<td>$(B_1,B_6)$</td>
<td>${t_1,t_2,t_4,t_5}$</td>
</tr>
<tr>
<td>4</td>
<td>$(B_1,B_3)$</td>
<td>${t_1,t_2,t_3,t_5}$</td>
</tr>
<tr>
<td>5</td>
<td>$(B_3,B_5)$</td>
<td>${t_1,t_2,t_3,t_4,t_5}$</td>
</tr>
<tr>
<td>6</td>
<td>$(B_3,B_6)$</td>
<td>${t_2,t_3,t_4}$</td>
</tr>
</tbody>
</table>

Table 6.2 MFTS_Source table
paths, the pairs of ‘definition/redefinition’ and ‘use’ are identified as DU pairs. These DU pairs are used as coverage criterion to assess the adequacy of requirement coverage by the representative set, Trs.

**MFTS Algorithm:** The cardinality of each test set is first determined. The test sets T_i’s are then sorted in descending order of cardinality and placed in MFTS_Source table (Table 6.2). The count of each test case t_j in the currently placed test set T_i is determined and placed in the local array TC_i. This is followed by determining the min_support which is ‘3’ for the sample program. Similarly the values of GTC array are also computed as in Table 6.3. Then test set (TID:5) is retrieved from the MFTS_Source table and placed in the temporary table called MFTS. Its corresponding support value, support(T_i) is set as ‘1’. This is followed by recomputing the value of GTC. As the test sets in MFTS table are disjoint, the next test set (TID:1) is placed in this table from MFTS_Source table. The support for this test set is incremented by ‘1’. Again GTC values are recomputed. Test sets in MFTS are checked if they are not disjoint. As test set (TID:1) is a subset of the test set

<table>
<thead>
<tr>
<th>TIDs</th>
<th>Test Sets</th>
<th>Cardinality</th>
<th>TC_i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>t_1</td>
<td>t_2</td>
</tr>
<tr>
<td>5</td>
<td>{t_1,t_2,t_3,t_4,t_5}</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>{t_1,t_2,t_4,t_5}</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>{t_1,t_2,t_4,t_5}</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>{t_1,t_2,t_3,t_5}</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>{t_2,t_3,t_5}</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>{t_2,t_3,t_4}</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

**min_support=3**
Table 6.3 Computing GTC for the sample test sets

<table>
<thead>
<tr>
<th>Test case ( t_j )</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>4</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>6</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>4</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>4</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>5</td>
</tr>
</tbody>
</table>

\( \text{GTC} = \{t_1;4,t_2;6,t_3;4,t_4;4,t_5;5\} \)

(TID:5), it’s support is incremented. Test set (TID:3) is added to the temporary table (MFTS), it’s support value is assigned as ‘1’ and GTC value is recomputed. The subsets in MFTS table is identified as TID:1 and hence it’s support value is incremented to ‘3’. As the support value is equal to the minimum support value and TID:1 is maximal frequent test set, it is placed in the representative set \( T_{rs} \). Test set TID: 4 is placed in MFTS table followed by assigning the support value as ‘1’ and recomputing GTC values. As TID: 4 is a subset of TID 5 the support value of TID:4 is again incremented by ‘1’. Similarly the first element \( (t_1) \) in GTC contains a value ‘0’ indicating that \( t_1 \) is infrequent. Hence ‘\( t_1 \)’ is removed from all test sets in MFTS. The above process iterates till all the elements of GTC becomes ‘0’ (Table 6.4). The resulting \( T_{rs} \) table contains maximal test sets. For the sample program considered the representative set obtained through the proposed MFTS and state-of-the-art algorithms are depicted in Table 6.7.

**HGS Algorithm:** Table 6.5 depicts the test suite reduction process using HGS algorithm along with the intermediate values of some essential parameters. This algorithm begins with the computation of the test set cardinality. Then test sets with cardinality ‘1’ (singleton test cases) are
Table 6.4 Using MFTS algorithm for the sample test sets

<table>
<thead>
<tr>
<th>TID</th>
<th>MFTS</th>
<th>GTC</th>
<th>$T_{rs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>${t_1,t_2,t_3,t_4,t_5:1}$</td>
<td>${t_1:3,t_2:5,t_3:3,t_4:3,t_5:4}$</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>${t_1,t_2,t_3,t_4,t_5:1$</td>
<td>$t_1,t_2,t_4,t_5:2}$</td>
<td>${t_1:2,t_2:4,t_3:3,t_4:2,t_5:3}$</td>
</tr>
<tr>
<td>3</td>
<td>${t_1,t_2,t_3,t_4,t_5:1}$</td>
<td>$t_1:1,t_2:3,t_3:3,t_4:1,t_5:2$</td>
<td>${t_1,t_2,t_4,t_5:3}$</td>
</tr>
<tr>
<td>4</td>
<td>${t_1,t_2,t_3,t_4,t_5:1$</td>
<td>$t_1,t_2,t_3,t_5:2}$</td>
<td>${t_1:0,t_2:2,t_3:2,t_4:1,t_5:1}$</td>
</tr>
<tr>
<td>2</td>
<td>${t_2,t_3,t_4,t_5:1}$</td>
<td>$t_1:-,t_2:1,t_3:1,t_4:1,t_5:0$</td>
<td>${t_1,t_2,t_4,t_5:3$</td>
</tr>
<tr>
<td>6</td>
<td>${t_2,t_3,t_4}$</td>
<td>$t_1:-,t_2:0,t_3:0,t_4:0,t_5:-$</td>
<td>${t_1,t_2,t_4,t_5:3$</td>
</tr>
</tbody>
</table>

Table 6.5 Test set description

| Test sets $T_i$ | Requirement $r_i$ | Test set cardinality $|T_i|$ | Singleton Test set (Yes/No) |
|-----------------|-------------------|----------------|-----------------------------|
| $T_1$           | $r_1$             | 4              | No                          |
| $T_2$           | $r_2$             | 3              | No                          |
| $T_3$           | $r_3$             | 4              | No                          |
| $T_4$           | $r_4$             | 4              | No                          |
| $T_5$           | $r_5$             | 5              | No                          |
| $T_6$           | $r_6$             | 3              | No                          |

identified by scanning through the test sets. As there are no singleton test sets, the temporary set $T_s$ is a null set. Before proceeding to the next higher cardinality the contents of the temporary set is checked, as it is null the test suite reduction operation terminates without performing any test suite
**BOG Algorithm:** BOG algorithm begins by multiplying the test case requirement matrix with its transposed matrix. The resultant matrix is placed in a matrix called the multiplied matrix as in (Figure 6.2a). Using this matrix the number of requirements covered by each test case is placed in a vector called sumColumns as shown in Figure 6.2b. This is followed by determining the maximum diagonal value(s) and placing those test cases into the maxList. Similarly the minimum diagonal value is determined and those test case(s) are placed into the minList. Then, test cases of maxList and minList are intersected. Table 6.6a shows that these list values are disjoint. The function, selectOptimumTestCase() is invoked to select near optimal test case(s). Then the values computed are tabulated as in table 6.6b. After the first call to the function SelectOptimumTestCase(), the test case $t_2$ is added to the representative set $T_{rs}$. Then the sumColumn vector is updated with respect to the selected test case and cumulative coverage of the requirements. Similarly the multiplied matrix is also updated for unselected test cases. After the update process, the values of the diagonal elements become “0” and hence the reduction process terminates and the representative set constructed consists of \{t_1,t_3,t_4\}.

\[
\text{Multiplied} = \begin{pmatrix}
4 & 2 & 4 & 4 & 4 \\
2 & 3 & 2 & 2 & 3 \\
4 & 2 & 4 & 4 & 4 \\
4 & 2 & 4 & 4 & 4 \\
4 & 3 & 4 & 4 & 5 \\
2 & 3 & 2 & 2 & 3
\end{pmatrix} \quad \Rightarrow \quad \text{sumColumns} = \begin{pmatrix}
18 \\
18 \\
20
\end{pmatrix}
\]

(a) Multiplied matrix    (b) sumColumns matrix

**Figure 6.2 Matrix values of BOG algorithm**
Table 6.6 Reduction process using BOG algorithm

(a) Initial

<table>
<thead>
<tr>
<th>Iteration No.</th>
<th>Diagonal Elements</th>
<th>Element Intersection of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4,3,5,5,3</td>
<td>t₅, t₂</td>
</tr>
</tbody>
</table>

(b) Iterative call

<table>
<thead>
<tr>
<th>Call</th>
<th>MinMax</th>
<th>Maxmax</th>
<th>Maxmin</th>
<th>Minmin</th>
<th>Minlist</th>
<th>Maxlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

The observations of the test suite reduction (Table 6.7) are summarized as follows:

- HGS algorithm cannot reduce without singleton test sets (test set with cardinality 1).
- BOG offers better test suite reduction and coverage but poor fault detection capability.
- MFTS identifies the maximal frequent test sets which is the representative set T_{rs}.

Table 6.7 MFTS and state-of-the-art algorithms

<table>
<thead>
<tr>
<th>Test Suite Reduction Algorithms</th>
<th>Representative Set T_{rs}</th>
</tr>
</thead>
<tbody>
<tr>
<td>HGS</td>
<td>Null</td>
</tr>
<tr>
<td>BOG</td>
<td>{t₂}</td>
</tr>
<tr>
<td>MFTS</td>
<td>{t₁,t₂,t₄,t₅} / {t₂,t₃,t₅}</td>
</tr>
</tbody>
</table>
In the next section the observations of the experimentation and results of the proposed MFTS and state-of-the-art algorithms are discussed.

6.3 EXPERIMENTAL PROCEDURE

The goal of the proposed MFTS algorithm is to achieve optimal test suite size reduction with maximum coverage while losing little to none of the fault detection effectiveness. The experimentation compares the relative performance and effectiveness of the proposed MFTS algorithm with the state-of-the-art HGS and BOG algorithms.

6.3.1 Subject Programs for Testing

The performance evaluation of the proposed MFTS algorithm and state-of-the-art algorithms has been performed on a set of ten subject programs (Table 6.8). These programs in Java perform simple operations using control flow statements with ‘definition’/‘redefinition’ and ‘use’ of three to five variables. The DU pairs generated for the programs in Table 6.8 range from “4” to “7”. The DU pairs generated serve as test coverage criteria to assess the level of test suite optimization.

In contrast to the existing works, the proposed algorithm iteratively exploits the implications among the coverage requirements and test suites using the minimum support value to derive the representative set T_{rs}. For each program the test suite was hand-instrumented for dataflow adequacy criterion (Rapps & Weyuker 1985). The proposed algorithm also uses hand seeded faults for the subject programs listed in Table 8 as suggested by Andrews et. al (2006). In this work, all faults are treated as equally severe. The representative set (reduced test suite) obtained for the ten subject programs
Table 6.8 Programs used for testing

<table>
<thead>
<tr>
<th>Program No.</th>
<th>Program Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pgm 1</td>
<td>Leap year</td>
</tr>
<tr>
<td>Pgm 2</td>
<td>Num_Digits</td>
</tr>
<tr>
<td>Pgm 3</td>
<td>Calc_bill</td>
</tr>
<tr>
<td>Pgm 4</td>
<td>Compute_1</td>
</tr>
<tr>
<td>Pgm 5</td>
<td>Valid Pin</td>
</tr>
<tr>
<td>Pgm 6</td>
<td>Sum_Digits</td>
</tr>
<tr>
<td>Pgm 7</td>
<td>Max_Val</td>
</tr>
<tr>
<td>Pgm 8</td>
<td>Prime_Num</td>
</tr>
<tr>
<td>Pgm 9</td>
<td>Prod_Discount</td>
</tr>
<tr>
<td>Pgm 10</td>
<td>Triangle_type</td>
</tr>
</tbody>
</table>

are analysed for test suite size, requirement coverage, fault detection effectiveness and execution time.

6.3.2 Test Effectiveness Measures

Software test metrics aid in evaluating the test cases and also focus more on the testing activity. In the experimental design variables are considered. The presented work uses two classifications of variables namely independent variables and dependent variables.

The experiment manipulated four independent variables are: MFTS algorithm, state-of-the-art HGS and BOG algorithms, along with the DU pairs as requirements. Similarly for the data points, Test suite T and requirements R, four dependent variables were computed:
The following test metrics are used to evaluate the performance of the proposed MFTS and state-of-the-art algorithms:

- The first test metric is the percentage reduction in Test Suite Size ($T_{\text{red}}$) (Scott & Atif 2007) and computed as in Equation (6.7).

\[
T_{\text{red}} = 100 \times \left( 1 - \frac{T_{\text{rs}}}{T} \right)
\]  

(6.7)

$T$ denotes the number of test cases in the universal test suite and $T_{\text{rs}}$ the number of test cases in the representative set. Higher $T_{\text{red}}$ indicates better test suite reduction.

- The second test metric is the percentage of Requirement Coverage ($R_{\text{cov}}$) and is computed as in Equation (6.8).

\[
R_{\text{cov}} = 100 \times \frac{R_{\text{rs}}}{R}
\]  

(6.8)

where $R$ denotes the total number of test requirements under consideration during test suite reduction and $R_{\text{rs}}$ is the number of requirements satisfied by the test cases in the representative set. The higher the percentage of $R_{\text{cov}}$, the better is requirement coverage by the representative set $T_{\text{rs}}$. 

- Test suite size
- Requirement coverage
- Fault Detection Density (FDD)
- Execution time
The third test metric includes the percentage of *Fault Detection Density* (FDD), formulated by Sampath et al (2008) and it is represented in Equation (6.9).

\[
FDD = \frac{\sum_{i=1}^{n} f_i}{|T| - |F|} \times 100
\]  

where \( i=1 \) to \( n \) and \( j=1 \) to \( m \)

where \(| F|\) denotes the total number of known faults, \(|T|\) size of the universal test suite and \( f_i \) is the fault(s) identified by each test case in the representative set. Higher FDD denotes better fault detection capability by the representative set \( T_{rs} \).

The last test metric used to evaluate the proposed MFTS and state-of-the-art algorithms is the *execution time* (\( \text{Time}_{\text{exec}} \)) as in Equation (6.10).

\[
\text{Time}_{\text{exec}} = \text{EndTime}_{\text{exec}} - \text{StartTime}_{\text{exec}}
\]  

The execution time of the proposed MFTS and state-of-art algorithms were determined using the built in Java functions.

### 6.4 RESULT ANALYSIS AND DISCUSSION

**Test Suite Reduction:** In the proposed MFTS algorithm, maximal frequent test sets are determined through simple operations. The basic operations include computing the test suite size, count of each test case \( t_j \) in test set \( T_i \) and identifying frequent test sets through the support values. Finally, the representative set \( T_{rs} \) is generated by applying the MFTS algorithm. The state-of-the-art HGS algorithm also uses test set cardinality to construct the representative set \( T_{rs} \). However the bottleneck of the HGS algorithm includes random selection of test cases into the representative set when there is a tie
between test cases. Another issue is the necessity for singleton test sets (test set with size one). Next, the state-of-the-art BOG algorithm considered for experimentation uses extensive matrix operation to reduce the test cases for the given test suite. However, in BOG algorithm the representative set \( T_{rs} \) constructed by test cases invariably depends on the order of test sets assigned for test suite reduction. The reduction in test suite size is illustrated for the proposed MFTS and state-of-the-art algorithms in Figure 6.3. The y-axis in the graph represents the percentage reduction in test suite size (Tred) against the ten subject programs. The maximum, average and minimum test suite size

![Figure 6.3 Reduction in Test Suite Size](image)

**Table 6.9 Reduction in Test Suite Size**

<table>
<thead>
<tr>
<th></th>
<th>HGS</th>
<th>BOG</th>
<th>MFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (%)</td>
<td>79.50</td>
<td>82.9</td>
<td>76.87</td>
</tr>
<tr>
<td>Maximum (%)</td>
<td>94.12</td>
<td>94.12</td>
<td>92.31</td>
</tr>
<tr>
<td>Minimum (%)</td>
<td>46.67</td>
<td>62.50</td>
<td>50.00</td>
</tr>
</tbody>
</table>
reduction obtained for the proposed MFTS algorithm with respect to the state-of-the-art algorithms is depicted in Table 6.9. The observations of the results tabulated show that the maximum reduction in test suite size for the proposed MFTS algorithm is 92.31\% while the average was 76.87\%. These results reveal that the reduction in test suite size for the proposed MFTS algorithm is competitive with respect to the state-of-the-art algorithms.

**Requirement Coverage:** One way to access the quality of testing is through the requirements test covered by the representative set. The coverage of requirements is a fundamental need throughout the software life cycle. Requirement coverage also involves mapping test cases with requirements to check whether all the requirements are covered by the test cases in the representative set. Requirement coverage defines the tracing of a program by test cases with respect to the DU pairs. It also determines the areas of a program not exercised by a set of test cases during testing. Figure 6.4 shows the results of the requirements covered for the ten subject programs. The requirement coverage in percentage has been represented in the y-axis against all the ten programs considered in the x-axis. Table 6.10 shows the requirement coverage after experimentation for the proposed MFTS algorithm.

![Figure 6.4 Requirements Coverage Obtained](image-url)
Table 6.10 Requirement Coverage Summary

<table>
<thead>
<tr>
<th></th>
<th>HGS</th>
<th>BOG</th>
<th>MFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (%)</td>
<td>79.50</td>
<td>82.90</td>
<td>94.10</td>
</tr>
<tr>
<td>Maximum (%)</td>
<td>94.12</td>
<td>94.12</td>
<td>100</td>
</tr>
<tr>
<td>Minimum (%)</td>
<td>46.67</td>
<td>62.50</td>
<td>75.00</td>
</tr>
</tbody>
</table>

and state-of-the-art algorithms. The observations made after experimentation show that MFTS algorithm provided an average of 94.10% requirement coverage against the state-of-the-art HGS (79.5%) and BOG (82.9%) algorithms. The maximum requirement coverage attained while using MFTS algorithm includes all the DU pairs considered as test requirements. However for the state-of-the-art algorithms the maximum coverage was 94.12% only. Thus, the results reveal that the proposed MFTS algorithm consistently outperformed the state-of-the-art HGS and BOG algorithms in requirement coverage.

**Fault Detection Effectiveness:** The objective of testing is to fulfill the requirements and also identify maximum number of faults injected. Fault detection plays a vital role in the software industry, as inadequate testing may conversely lead to other software related problems. The faults are injected in each test set through a fault matrix. The fault detection effectiveness of the test cases in the representative set was evaluated using Equation 6.9. From the experimentation done it could be inferred that requirement coverage directly influenced the fault detection capability of the representative set. Figure 6.5 illustrates FDD for the proposed MFTS algorithm and state-of-the-art algorithms. The FDD values in y-axis are plotted against the sample programs considered in x-axis. From the experiments conducted the average
FDD when using MFTS algorithm was 1.0 against the state-of-the-art algorithms which was 0.43 and 0.48 respectively. In the presented work FDD values close to 1.0 was achieved for most of the subject programs. This indicates that maximum number of known faults injected during test suite reduction was detected by the test cases in the representative set as stated in the work of Sampath et al (2008). The observations of the experimentation tabulated in Table 6.11 suggest that MFTS algorithm consistently outperformed HGS and BOG algorithms in determining the known faults using the test cases in the representative set.

![Figure 6.5 Fault Detection Effectiveness](image)

**Table 6.11 FDD values**

<table>
<thead>
<tr>
<th></th>
<th>HGS</th>
<th>BOG</th>
<th>MFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (%)</td>
<td>0.43</td>
<td>0.48</td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum (%)</td>
<td>0.67</td>
<td>0.67</td>
<td>1.0</td>
</tr>
<tr>
<td>Minimum (%)</td>
<td>0.13</td>
<td>0.27</td>
<td>0.86</td>
</tr>
</tbody>
</table>
**Execution Time:** Accelerating the process of test suite reduction is also an important parameter in software testing as testers are often subjected to time bound activities. The experimental studies done on the test sets have a direct correlation with the execution time. The time taken for the test suite optimization by the proposed MFTS and state-of-the-art algorithms has been illustrated in Figure 6.6. The execution time by the test suite reduction algorithms in seconds is plotted in y-axis against the programs considered in x-axis. Table 6.12 provides a summary of the execution time of the proposed MFTS and state-of-the-art algorithms. From this table, it can be observed that the proposed MFTS algorithm uses a simple approach for test suite reduction with an average execution time of 18.6 seconds. It further reveals that when HGS algorithm was used, singleton test cases were first determined and placed in the representative set followed by the actual iterative test suite reduction process. This invariably leads to substantial time overhead in the optimization process. Further, BOG algorithm took a longer time to obtain the

![Figure 6.6 Execution Time for Test Suite Reduction](image)
Table 6.12 Execution Time

<table>
<thead>
<tr>
<th></th>
<th>HGS</th>
<th>BOG</th>
<th>MFTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average (sec)</td>
<td>38.8</td>
<td>40.5</td>
<td>18.6</td>
</tr>
<tr>
<td>Maximum (sec)</td>
<td>66.0</td>
<td>55.0</td>
<td>39.0</td>
</tr>
<tr>
<td>Minimum (sec)</td>
<td>30.0</td>
<td>31.0</td>
<td>11.0</td>
</tr>
</tbody>
</table>

representative set as it involved cumbersome computational activities. From the results tabulated it can concluded that MFTS algorithm accelerated the process of test suite reduction, while the state-of-the-art algorithms used complex/lengthy iterative approaches.

**Experimentation Summary:** The experimentation details of MFTS algorithm is categorized against the test paradigms and summarized in Table 6.13. This table gives a clear outlook of the experimentation process.

Table 6.13 MFTS paradigms for experimentation

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Description of Experimentation</th>
<th>MFTS Paradigms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Basic Concept used for test suite reduction</td>
<td>Maximal frequent test sets based on support value</td>
</tr>
<tr>
<td>2.</td>
<td>Representation of requirements</td>
<td>Test ID</td>
</tr>
<tr>
<td>3.</td>
<td>Representation of test sets</td>
<td>Local array TC\textsubscript{i}</td>
</tr>
<tr>
<td>4.</td>
<td>Number of scans to generate the representative set</td>
<td>One</td>
</tr>
<tr>
<td>5.</td>
<td>Test Metrics addressed</td>
<td>Test suite size, requirement coverage, fault detection and execution time</td>
</tr>
<tr>
<td>6.</td>
<td>Computational Complexity</td>
<td>$\mathcal{O}(m)$</td>
</tr>
</tbody>
</table>
The next section presents the summary of the proposed MFTS algorithm and state-of-the-art algorithms.

**6.5 SUMMARY**

In the proposed work the concept of maximal frequent itemset mining has been adapted for test suite reduction. An algorithm called MFTS has been proposed to select maximal frequent test sets. The proposed MFTS algorithm focuses on:

(i) Test set size  
(ii) Test case cardinality  
(iii) Maximal frequent patterns among test sets

The performance studies of the subject programs reveal that the proposed MFTS algorithm provided an average of 76.87% reduction in test suite size, 94.1% requirement coverage and maximum FDD. The test suite reduction using MFTS algorithm was also close to the size reduction by the state-of-the-art HGS algorithm. The requirement coverage and FDD ratio of the MFTS generated representative set was better than that of the state-of-the-art: HGS and BOG algorithms. Further the test suite reduction by MFTS algorithm was faster than the state-of-the-art: HGS and BOG algorithms.

In the next and final chapter, the conclusions of the presented work and plan for future work have been discussed.