2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Better Algorithms to Minimize the Cost of Test Paths

Nan Li, Fei Li, and Jeff Offutt
George Mason University, Fairfax VA, USA
{nli1, fli4, offutt}@gmu.edu

***Abstract*—Model-based testing creates tests from abstract models of the software. These models are often described as graphs, and test requirements are defined as subpaths in the graphs. As a step toward creating concrete tests, complete (test) paths that include the subpaths through the graph are generated. Each test path is then transformed into a test. If we can generate fewer and shorter test paths, the cost of testing can be reduced. The minimum cost test paths problem is finding the test paths that satisfy all test requirements with the minimum cost.**

This paper presents new algorithms to solve the problem, and then presents data from an empirical comparison. The algorithms adapt approximation algorithms for the shortest superstring problem. The comparison is with an existing tool that uses a brute force approach to extend each subpath to a complete path. One new algorithm is based on the greedy set-covering algorithm and the other is based on finding a matching over a prefix graph. The comparison was performed on open software and showed that both new solutions generate fewer test paths than the brute force approach. The prefix-graph based solution takes much less time than the other two solutions when the number of test requirements is large.

I. Introduction

In model-based testing, models are abstracted from software artifacts and coverage criteria are used to generate test requirements. The models can be in various forms; this paper considers graphs and addresses a specific problem with satisfying criteria on graphs. Graphs can come from any functional requirements, design models, or even source (which is not normally considered in model-based testing), but this research considers graphs in the abstract without consideration of what software artifact was used to generate them. Graph criteria typically define test requirements (*TR*) as sequences of nodes in the graph (a *subpath*). These are then turned into *abstract tests* by creating complete paths. A complete path is a path from an initial node to a final node, called a *test path* (*TP*). Each test path represents a complete execution of the software being modeled. Test paths can be constructed by creating one test path per test requirement, or by satisfying multiple test requirements in the same path.

It turns out that how test requirements are turned into test paths has a major impact on the overall cost of testing, both initially and as the software goes through maintenance. Thus, this research addresses the following problem:

Definition 1 (Minimum Cost Test Paths Problem (MCTP)). *Consider a set of test requirements* $TR = \{r_1,\ r_2,\ \cdots,\ r_n\}$. *Each test requirement is presented as a subpath in a graph* $G = (V,\ E)$. *The problem MCTP is to find a set of test paths* $TP = \{t_1, t_2,\ \ldots,\ t_k\}$ *that cover all test requirements in the graph* G *such that the cost of using the test paths is minimum.*

It turns out that the cost of testing is a multi-faceted problem. In fact, the cost can be reduced in several ways. Following are four possible goals for reducing cost, some of which are complementary and some of which conflict. The MCTP definition above could be refined to explicitly satisfy any of these goals.

1) **Fewer total nodes**. Each TP contains a sequence of nodes in the graph and every execution of every node represents a cost. More nodes in the set of TPs means more values that have to be found, more time executing the tests, and more time evaluating the results of the tests.
2) **Fewer test paths**. Every TP is an abstract test that will be realized into a concrete test, and every test has an associated cost. This includes the initial cost of finding values for the test and a continuing cost of running the test and modifying the test as the software evolves.
3) **Fewer test requirements per test path** (Smaller TR to TP ratio). Some TRs are infeasible, but which is often not known until abstract tests are realized into concrete tests. If one TP satisfies several TRs, and one of those TRs is infeasible, the entire TP is lost.
4) **Shorter test paths**. Finding values for TPs becomes more difficult as the TPs get longer. Tests from long TPs are also long, which makes them harder to maintain as the software evolves.

Satisfying multiple goals is hard enough, but some of these conflict. The most obvious way to reduce the TR to TP ratio is to have exactly one TP for each TR. However, that conflicts with the **Fewer test paths** goal. The **Shorter test paths** goal and the **Smaller TR to TP ratio** goal are complementary because reducing the ratio will result in shorter test paths. This implies that the infeasible path

This material is based upon work supported by NSF under Grants CCF-0915681 and CCF-1146578. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.40

Problem	NP-completeness	Reduction / Solution
1	NP-complete	Bin-Packing
2	P	A modified algorithm of the one solving CP_1^1 [1]
3	NP-complete	Bin-Packing
4	NP-complete	Bin-Packing
5	NP-complete	Bin-Packing

Table I
VARIANTS OF THE MCTP PROBLEM AND THEIR NP-COMPLETENESS

problem that occurs when the ratio gets large can also occur when we get fewer but longer test paths. In practice, the TP length depends on the context; the source of the graph and the overall difficulty of finding values for the concrete test.

The **Fewer total nodes** goal has no tradeoffs with the other goals above, thus is always a valid goal. To balance the other goals, this research takes the following approach. We hope to find TPs to satisfy all TRs with the fewest overall number of nodes, with the fewest test paths, and with the fewest paths such that the maximum TR to TP ratio is not "too big." Of course, "too big" seems somewhat arbitrary but it depends on the context and the source of the graph, just like the TP length. Specially the goals to be optimized are:

1) The total number of nodes
2) The total number of test paths
3) The maximum ratio of TR to TP
4) The total number of test paths subject to a bounded ratio of TR to TP
5) The total number of nodes subject to a bounded ratio of TR to TP

Ideally, we could achieve all five goals with one algorithm. However, it is very hard to design an algorithm to satisfy more than two optimization goals. Each of these five goals can be specific instantiations of the MCTP problem. Table I shows whether each of them are NP-complete and the reduction approaches.

Variant (2), finding the fewest number of test paths, can be solved in polynomial time using a modified algorithm proposed by Aho and Lee [1]. However, other variants of the MCTP problem are provably NP-complete. Their reductions come from a well-known NP-complete problem called the Bin-Packing problem [2], [3]. Space does not permit us to include the proofs.

Since variant (2) is polynomial-time solvable, this paper focuses on approximation algorithms that can solve the other NP-complete variants. The new algorithms proposed in this paper form the solid algorithmic ground to solve variants (1), (3), (4), and (5) in Table I. To solve a specific variant, we can adapt our solutions slightly. For instance, if we want to get the fewest number of nodes in the test paths, after we get the test paths using our algorithms introduced in Section III, we can use dynamic programming to approximate the minimum number of nodes. If we want to ensure that the maximum TR to TP ratio is below a pre-defined threshold, then we can use a number to restrict the length of each test path generated in our algorithms. The algorithms developed in this paper provide a basis to solve the general problem MCTP.

Note this research does not emphasize time. Time depends on many factors, including the people, tools, test criteria, and software under test. Instead, this research focuses on stable, long term costs. For completeness, the experimental section does measure time explicitly.

The paper is organized as follows. Section II provides background and related work. Next, three solutions to the MCTP problem are defined and compared in section III. An existing brute force solution has been used in a partial test automation tool [4]. Two new algorithms are presented, a greedy set-covering solution, and a prefix-graph based solution. They are based on the approximation algorithms designed for the shortest superstring problem [2]. Experiments, described in section IV, compares these three algorithms' performance against our formulation of the MCTP. These algorithms are compared in terms of the number of the test paths found, the lengths of the test paths found, the maximum ratio of the number of TRs to the number of TPs, and their execution times. Section V concludes the paper and describes some ideas for the future.

II. BACKGROUND AND RELATED WORK

This section describes model-based testing and defines the prime path coverage criterion. A related testing problem is described, then the shortest superstring problem is defined. Existing algorithms for the shortest superstring problem are described, then the algorithms adapted to use in this research. Lastly, we discuss whether fewer but longer test paths that are generated by our solutions are good by comparison with many but shorter paths.

A. Model-Based Software Testing and Prime Path Coverage

In *model-based testing* (*MBT*), a model is a partial abstract description of a program, usually reflecting functional aspects. Tests are designed in terms of the model, for example, a path in a graph. They are called *abstract tests*.

The following definitions are taken from Ammann and Offutt's testing book [5]. A graph G is drawn as a collection of circles connected by arrows, where the circles represent nodes and the arrows represent edges. Formally, G is

- a set N of *nodes*, where $N \neq \emptyset$
- a set N_0 of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$
- a set N_f of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$
- a set E of *edges*, where E is a subset of $N \times N$

Thus, a test graph G **requires** at least one initial and one final node, but **allows** more.

Edges are considered to be directed arcs *from* one node n_i and *to* another n_j and written as (n_i, n_j). The initial node

n_i is sometimes called the *predecessor* and n_j is called the *successor*. A *path* is a sequence $[n_1,\ n_2,\ ...,\ n_M]$ of nodes, where each pair of adjacent nodes, $(n_i,\ n_{i+1})$, $1 \leq i < M$, is in the set E of edges. The *length* of a path is defined as the number of edges it contains. A *subpath* of a path p is a subsequence of p (including p itself). A *test path* represents the execution of a test case on a graph. Test paths must start at an initial node and end at a final node. A test path p *tours* a subpath q if q is a subpath of p. A path from node n_i to n_j is *simple* if no node appears more than once in the path, with the exception that the first and last nodes may be identical. That is, simple paths have no internal loops, although the entire path itself may be a loop. Even small graphs will have lots of simple paths, many of which are subpaths of other simple paths. Prime paths eliminate this redundancy without omitting any important coverage. A path from n_i to n_j is *prime* if it is simple and does not appear as a proper subpath of any other simple path.

Definition 2 (Prime Path Coverage). *Tour each reachable prime path in G.*

Test requirements are specific elements of software artifacts that must be satisfied or covered. An example test requirement for *statement coverage* is "Execute statement 7." A *test criterion* is a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests. That is, the criterion describes the test requirements for the artifact in a complete and unambiguous manner. For example, the test criterion "cover every node" results in one test requirement for each node in the graph. A test criterion requires that at least one test path is needed to cover all test requirements. Test requirements are *satisfied* by *visiting* specific nodes or edges or by *touring* specific paths or subpaths.

B. Test Suite Minimization Problem

The nearest related testing problem to the MCTP is the problem of reducing the number of tests in a test suite while still maintaining coverage. The *test suite minimization problem* is the subject of a comprehensive survey by Yoo and Harman [6]:

Definition 3 (Test Suite Minimization Problem). *Consider a set of test-case requirements $\mathcal{R} = \{r_1,\ \cdots,\ r_n\}$ that must be satisfied to provide the desired adequate testing of the program. There is a test suite $\mathcal{T}$. For any subset $T_i \subseteq \mathcal{T}$, $\forall i = 1,\ \ldots,\ n$, any test case $t_j \in T_i$ can be used to test a subset of $\mathcal{R}$. The problem TSMP is to find a representative set $\mathcal{T}' \subseteq \mathcal{T}$ such that all the requirements in $\mathcal{R}$ can be tested using the test cases in $\mathcal{T}'$.*

This test minimization problem is related to the set cover problem and several heuristics have been developed [7], [8], [9], [10].

This research uses a heuristic algorithm to reduce the number of test paths for each solution after the test paths are generated using the brute force, set-covering, and prefix-graph based algorithms. These approaches are similar to the GE and GRE heuristics proposed by Chen and Lau [8]. The minimization algorithm is a simplified version of the GRE heuristic, removing all *redundant* test paths, but not running the GE heuristic as Chen and Lau did. A test path is *redundant* if it satisfies a proper subset of the test requirements satisfied by another test path.

C. The Shortest Superstring Problem

This paper adapts the ideas used in solving the *shortest superstring problem*. The *shortest superstring problem* has been used in the fields of computational biology and data compression, but not previously in software testing. The first variant in table I, the minimum test paths problem, can be modeled as a shortest superstring problem, allowing us to adapt existing algorithms for solving the shortest superstring problem. *The shortest superstring problem* is defined as follows.

Definition 4 (Superstring Problem [2]). *Given a finite alphabet Σ, and a set of n strings, $S = \{s_1,\ \ldots,\ s_n\} \subseteq \Sigma^+$, the superstring problem is to find a shortest string s that contains each s_i as its substring. Without loss of generality, we assume that no string s_i is a substring of another string s_j, $\forall j \neq i$.*

In model-based software testing with graphs, a string is a test requirement as a sequence of nodes, specifically, a prime path in this paper. For example, for the two prime paths $[1,\ 2,\ 3,\ 1]$ and $[2,\ 3,\ 1,\ 2]$, two possible superstring test paths that cover them are $s_1 = [1,\ 2,\ 3,\ 1,\ 2]$ and $s_2 = [2,\ 3,\ 1,\ 2,\ 3,\ 1]$. The length of the string s is $|s|$. Given two different strings s and t, the overlap between s and t, $|over(s,\ t)|$, is the length of the longest string x such that $s = mx$ and $t = xn$ with non-empty strings m and n. m, denoted as $|pref(s,\ t)|$ is the prefix of s with respect to t. $|pref(s,\ t)|$ is also referred as the *prefix distance* from s to t. If we use the example above, for s_1, $[2,\ 3,\ 1]$ is the overlap and $[1]$ is the prefix of $[1,\ 2,\ 3,\ 1,\ 2]$; for s_2, $[1,\ 2]$ is the overlap and $[2,\ 3]$ is the prefix of $[2,\ 3,\ 1,\ 2,\ 3,\ 1]$.

The shortest superstring problem was proved to be *NP-complete* by Gallant, Maier and Strorer [11]. Tarhio and Ukkonen [12] proposed a greedy algorithm whose *approximation ratio* (the ratio between the polynomial-time approximation algorithm and the hypothetical optimal solution) is $2 \ln n$. Blum et al. [13] presented another greedy algorithm that finds a superstring that is no longer than four times optimal and a modified one that was no more than three times optimal. The best known approximation ratio is 2, given by Weinard and Schnitger [14].

Romero, Brizuela and Tchernykh [15] conducted an experimental comparison of the shortest superstring problem.

Their paper compared a 3-approximation algorithm and a 4-approximation algorithm. The results showed the superstrings returned by both approximation algorithms are almost the same. Both solutions were very close to the optimum; at most 1.4% longer than the optimum.

Romero et al. used DNA sequences as examples in their experiment to find out a shortest superstring; this experiment used test requirements as examples. This paper presents results on test requirements, precisely prime paths, and different algorithms. Instead of generating only one superstring (super-test requirement), it is necessary to split the results of the superstring into separate test paths. Details are described in Section III.

D. The Greedy Set-Covering Algorithm

It is necessary to study the existing approximation algorithms for the shortest superstring problem. The first is the set-covering algorithm using the greedy idea. Consider the superstring problem. The input to an algorithm is a set of strings, S, of size $|S| = n$. The greedy algorithm works as below:

1) Initialize T to be S.
2) At each step, select two strings from T that have the maximum overlap and replace them with the string obtained by overlapping them as much as possible. If the remaining strings have no overlap, they are concatenated.
3) After $n - 1$ steps, the single string in T will be the superstring returned.

For $s_i, s_j \in S$ and $k > 0$, if the last k symbols of s_i are the same as the first k symbols of s_j, let σ_{ijk} be the string obtained by overlapping these k positions of s_i and s_j. Figure 1 shows the superstring covering s_i and s_j.

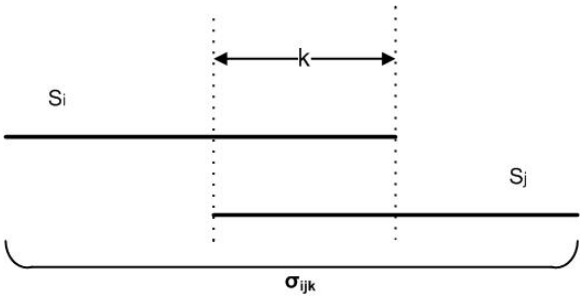

Figure 1. A superstring for two arbitrary strings

Let M be the set that consists of the string σ_{ijk} for all valid choices of i, j, k. The final set is $P \cup M$, where P is the set of the original strings.

E. The Matching-based Prefix Graph Algorithm

This approximation algorithm for the shortest superstring problem finds matchings based on a prefix graph and the definition of prefix graph is shown as follows.

Definition 5 (Prefix Graph). *Given a set of strings S, a* prefix graph $G = (V,\ E)$ *is constructed as follows: a vertex $v_i \in V$ represents a string $s_i \in S$. For each edge $e_{ij} \in E$, the cost $c_{e_{ij}}$ of the edge is $|pref(s_i,\ s_j)|$ if s_i and s_j overlap or $+\infty$ otherwise.*

The matching-based approximation algorithm for the shortest superstring problem is to find a matching over a prefix graph. We observe that the minimum weight of a traveling salesman tour of the prefix graph gives a lower bound on the length of the resulting optimal superstring (denoted by OPT). However the minimum traveling salesman tour OPT cannot be efficiently computed since this problem is NP-complete. Thus, the key idea is to find the minimum weighted paths, using cycle covers, over the prefix graph G. A *cycle cover* is a collection of disjoint cycles that cover all vertices. If $s_1,\ s_2,\ \ldots,\ s_n,\ s_1$ is the minimum-weight cycle cover, then

$$\begin{aligned} OPT &\geq \sum_i |prefix(s_i,\ s_{i+1}| + |overlap(s_1,\ s_n)| \\ &= |prefix(s_1,\ s_2)| + |prefix(s_2,\ s_3)| + \\ &\quad \ldots + |prefix(s_n,\ s_1)| + |overlap(s_n,\ s_1)|. \end{aligned}$$

The prefix graph is then transformed into a bipartite graph $G = (U \cup V,\ E)$. $U = \{u_1, ..., u_n\}$ and $V = \{v_1, ..., v_n\}$ are the vertex sets of the two sides of the graph. For each vertex u_i on the left side and each vertex v_j on the right side, add an edge of weight $|prefix(u_i, v_j)|$ if u_i and v_j overlap. Given a graph $G = (U \cup V,\ E)$, a *matching* M in G is a set of edges such that no two edges share a common vertex. A *perfect matching* ensures that every vertex of the graph is incident to exactly one edge. Therefore, each cycle cover of the prefix graph corresponds to a perfect matching of the same weight in G. Thus, finding a minimum weight cycle is equivalent to finding a minimum weight perfect matching in G.

The bipartite graph $G = (U \cup V,\ E)$ is used to find as many perfect matchings as possible. A maximum matching is found from those. The *Edmonds-Karp* algorithm [16] can find a maximum matching in the bipartite graph. Although the running time of the *Edmonds-Karp* algorithm is $O(VE^2)$, it runs slowly in practice, thus was not used.

III. Algorithms

The current brute force, greedy set-covering, and prefix-graph *approaches* (*solutions*) include several *algorithmic components* (*algorithms*). The brute force solution is composed of algorithms 1, 2, 3, and the minimization algorithm. The minimization algorithm from section II-B is quite simple and not shown in this paper. The greedy set-covering solution consists of algorithm 4 (the set-covering algorithm), algorithm 6 (the splitting algorithm), and the minimization

algorithm. The prefix-graph solution is comprised of algorithm 5 (the matching-based prefix graph algorithm), plus the splitting and minimization algorithms.

A. The Current Brute Force Solution

Algorithm 1 The heuristic algorithm to cover all nodes

Input: A directed graph $G = (V,\ E)$ and a set of initial nodes $I \subseteq V$
Output: A set of paths P that cover all the nodes in V.

```
1: for each initial vertex v_i ∈ I do
2:    run the Breath-First-Search (BFS) [16] algorithm to cover the vertices in V;
3:    build a BFS-tree with the root v_i;
4:    add the path from v_i to a vertex v ∈ V \ {v_i} into P;
5: end for
6: return P
```

Algorithm 2 The heuristic algorithm for generating a small set of test paths

Input: A set of paths P and a set of final nodes F
Output: A set of test paths TP

```
1: for each path p_i ∈ P do
2:    if the last node of p_i ∈ F then
3:       add p_i to TP and remove p_i from P;
4:    else
5:       for each node f_j ∈ F do
6:          if p_i includes f_j then
7:             chop the sub-path p_ij from the beginning to
               the position where f_j is in p_i
8:             add p to TP;
9:          end if
10:      end for
11:   else
12:      if the last node of p_i ∉ F then
13:         extend p_i to reach a final node;
14:      end if
15:   end if
16: end for
17: remove redundant paths in TP;
18: return TP
```

The current web application [4] takes a directed graph $G = (V,\ E)$ and set of test requirements represented by some subpaths in G. G has a set of initial nodes I and final nodes F. The idea of the brute force solution is to list a set of paths P that cover all the nodes. This step is implemented in algorithm 1. In algorithm 2, three steps are needed to make P a set of test paths. First, we keep those paths in P with initial nodes in I and final nodes in F. For a path in P with its initial node in I but final node not in F, as long as it contains a final node in this path, we chop to get a subpath with its initial node in I and final node in F. Last, we extend all the remaining paths (without final nodes in F) by making their final nodes fall in F. In algorithm 3, we extend all the test requirements that are not covered by the set of test paths generated by algorithm 2 by making their initial nodes and final nodes fall in I and F. The output of algorithm 1, a set of paths P, is an input to algorithm 2. Then the paths in the set P are extended to reach a final node using algorithm 2. This results in a set of complete test paths TP, which is an input to algorithm 3. The last step removes redundant paths.

Algorithm 3 The brute force algorithm for generating test paths to cover all requirements

Input: A set of test requirements TR and a small set of test paths TP
Output: A complete set of test paths CTR that covers all test requirements

```
1: for each test requirement tr_i ∈ TR that is not covered by TP do
2:    if the first node of tr_i is not an initial node, nor is the last node a final node then
3:       repeat
4:          for each path p_j ∈ TP do
5:             if p_j includes the first node or last node of r_i then
6:                extend tr_i using p_j such that r_i reaches an initial node or final node;
7:             end if
8:          end for
9:       until tr_i is a complete path
10:      add tr_i to CTR;
11:   end if
12: end for
13: return CTR
```

B. A Set-Covering Based Solution

This solution uses the *set-covering algorithm*, the *splitting algorithm*, and the *minimization algorithm*. Given a set of test requirements, the set-covering algorithms return only one super-test requirement, which is concatenated by multiple paths. This super-test requirement covers all the test requirements. Then, the splitting algorithm cuts the super-test requirement into separate complete paths. Finally, we apply the minimization algorithm to remove any *redundant* test paths.

The set-covering algorithm is described in Algorithm 4. We use the greedy approach to find out the set-covering for the graph. The splitting algorithm is described in Algorithm 6, which is also used by the Matching-Based Prefix Graph solution (Algorithm 5). We describe the splitting algorithm in Section III-D.

The set-covering algorithm described in Algorithm 4 has two parts: constructing the sets and picking the set cover using the greedy algorithm. The running-time complexity of constructing the sets is $O(n^2)$ while that of picking the sets is $O(n \lg n)$. Let Π denote the super-test requirement that we are looking for. Consider a test path g_i. The *cost* of g_i is $|g_i|$ (the length of g_i) and the *effectiveness* of g_i is the *number* of test requirements that are not covered by the test paths already in Π but will be covered by g_i when we add g_i into Π. The line 8 of Algorithm 4 shows that the cost-effectiveness of a path g_i is calculated when picking set covers greedily during each iteration.

C. A Matching-Based Prefix Graph Solution

Algorithm 4 The set-covering algorithm

Input: A set of test requirements TR
Output: A concatenation of paths that covers all test requirements Π

1: **for** each test requirement $tr_i \in TR$ and test requirement $tr_j \in TR$, $i \neq j$ **do**
2: **if** tr_i has overlap with tr_j **then**
3: put the super-test requirement s_k of tr_i and tr_j into a set S {refer to figure 1}
4: **end if**
5: **end for**
6: **repeat**
7: **for** each element $g_i \in TR \bigcup S$ **do**
8: compute the cost-effectiveness of g_i;
9: find g_i with the *minimum* cost-effectiveness and extend the super-test requirement Π with g_i;
10: remove the test requirements covered by g_i;
11: **end for**
12: **until** $TR = \emptyset$
13: **return** the super-test requirement Π

This Matching-Based Prefix Graph Solution is composed of the prefix-graph based algorithm, the splitting algorithm, and the minimization algorithm. Given a set of test requirements, the prefix-graph based algorithm returns a set of super-test requirements that cover all the test requirements. In algorithm 5, we first construct a prefix graph and a bipartite graph $G = (V \bigcup U,\ E)$ in the same way as is described in Section II-E. A perfect matching M over this bipartite graph G is a set of edges that connect left side nodes to right side nodes in the bipartite graph (each node represents a test requirement in the bipartite graph) such that no node is incident to two or more edges. An edge in the perfect matching M represents a super-test requirement of the two nodes to which the edge is connected. Then, with the calculated M, we construct a cycle cover by merging super-test requirements represented by the edges in the perfect matching M. Here, the greedy algorithm is used to select a cycle cover or a test requirement that has the minimum cost-effectiveness each time until all requirements are covered. The cost of a cycle cover is the length of the cycle cover; the effectiveness of the cycle is the number of test requirements that are not covered by the test paths already in Π but can be covered by the cycle cover when we add this cycle into Π. Algorithm 6 then splits the super-test requirement into separate complete test paths. The minimization algorithm is used to remove all *redundant* test paths.

In the bipartite graph, the number of the vertices on the left side may be greater than, less than, or equal to the number of vertices on the right. Therefore, a "perfect matching" cannot guarantee that every vertex is incident to one edge, which means some vertices are not reached by the edges in the "perfect matching."

Algorithm 5 The matching-based prefix graph algorithm

Input: a set of test requirements TR
Output: a concatenation of paths that covers all the test requirements Π

1: construct a bipartite graph G from the prefix graph;
2: find a perfect matching M of G;
3: **repeat**
4: construct a set of cycle covers C from the elements in M;
5: **until** no overlap exist between the cycle covers in C and the remaining matchings in M
6: add the remaining matchings in C;
7: run the greedy algorithm over $C \cup TR$ to select one element $c \in C$ that has the minimum cost-effectiveness until $TR = \emptyset$; during each iteration, extend Π with the selected c and remove c from TR;
8: **return** the super-test requirement Π.

D. Splitting the Super-Test Requirement into Test Paths

Both the set-covering and prefix-graph based algorithm returns a super-test requirement that covers all test requirements. This "super-test requirement" must then be split into individual complete test paths, as shown in algorithm 6. A small set of test paths TP generated by algorithm 2 is one of the inputs of this splitting algorithm. The "super-test requirement" is partitioned into several intermediate paths and they are not connected to each other; in other words, the last node of path n_i is not connected to the first node of path n_{n+1}. If the resulting shorter paths are not test paths, these intermediate paths are extended to reach an initial node and a final node using TP (see Algorithm 3).

IV. Experiment

The algorithms in this paper apply to arbitrary graphs, without consideration of what software artifact they came from. Model-based testing designs tests from functional requirements or design models, and normally excludes source.

Algorithm 6 The algorithm to split a super-test requirement into test paths

Input: A super-test requirement Π, a small set of test paths TP

Output: A set of test paths CTP

```
1: for each node n_i in Π do
2:    build an empty path p;
3:    repeat
4:       extend p with n_i;
5:    until n_i is not connected to n_{i+1} on the graph
6:    extend this path p such that p becomes a test path
      using TP and add p to CTP;
7: end for
8: return CTP
```

Since this research considers graphs in the abstract and the origin of the graphs is irrelevant, the experimental verification chooses control flow graphs from source for convenience. The results apply to any test design strategy that uses graphs.

The three algorithms developed in section III were studied in two stages. In the first stage, 18 methods were taken from several open source projects, including *Joda-Time*[1], *javassist*[2], *graph coverage web application*[3], *Apache Ant*[4], and *jdom*[5]. All methods were written in Java. Methods that had relatively complicated control structures (nested loops) were chosen, because simple methods have fewer prime paths and thus the algorithm used to create test paths is less important. Control flow graphs were generated by hand for each method and the graph coverage web application (support software provided as part of Ammann and Offutt's testing book [4]) was used to generate prime paths.

We hypothesized that the better algorithms would show more benefit for graphs that have more complex structures especially nested loops. So in the second stage, another 19 methods from the open source projects were modified structurally to yield control flow graphs that had more complex structures. Modifications included adding a nested loop, adding a *continue* statement deleting a statement, and adding additional decision structures.

The three procedures described in section III were implemented (in Java). Then the prime paths for all 37 methods were input to each of the three procedures, resulting in test paths that tour all prime paths.

A. Experiment Environment and Process

The experiment was conducted on a Dell mini tower, with an Intel Core i5-750 (2.66GHz 8MB L3 cache) CPU. The primary measure of size is the number of prime paths, which was determined by the graph coverage web application. The program was executed without interactions with the Internet or interruption from other programs.

The three procedures were compared based on the number of test paths, the total number of nodes needed in the test paths, the maximum ratio of TR over TP, and the execution time. For each test path, TP is counted as 1. Thus, the ratio of TR to TP is the number of test requirements covered by this test path. The maximum ratio of TR over TP finds the most number of test requirements covered by any test path for each program. Execution time was measured with the Java builtin method *System.nanoTime()*.

To handle variance in measuring execution time, each subject was measured five times for the three solutions and the average (mean) of the measured time was computed.

B. Experiment Result

Table II shows results in the first stage of the experiment on natural open source methods. The table shows the number of prime paths (*PPs*), then results in terms of the number of test paths (*Paths*), and the total number of nodes in all test paths (*Nodes*). The *BF* column shows results from the brute force algorithm, *PG* denotes the prefix-graph algorithm, and *SC* represents the set-covering algorithm. These methods had from 9 to 1844 prime paths.

The next six columns show the length ratio of paths and nodes for each solution over the brute force solution. The bottom row shows the total cost of the paths and nodes, and the average ratios. The average ratios are weighted, that is, they are computed from the totals row. The last three columns display the maximum ratio of TR to TP for three solutions.

The prefix-graph and the set-covering based solutions generated no more test paths or nodes than the brute force solution, and for most methods, far fewer. The prefix-graph based solution generated fewer test paths than the set covering solution for most methods, but the set covering algorithm generated fewer for the three largest methods. The largest number of test requirements covered by one test path is 273 in the last subject, using the prefix-graph based solution. For the third subject, the set-covering solution covered 17 of the 27 (63%) test requirements with just one test path.

Table III presents the measured execution time of the three algorithms for each subject in the open source programs. The meanings of *PPs*, *BF*, *PG*, *SC*, *Total*, and *Average* are the same as in table II. The last three columns show the ratios of the running time of each solution over the brute force solution. The prefix-graph based solution takes the least time to generate the test paths for each subject. For the first 15 subjects, the set-covering based solution runs faster than the brute force solution; however, the brute force solution was faster for the last three subjects, which have

[1]http://joda-time.sourceforge.net/
[2]http://www.jboss.org/javassist/
[3]http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage
[4]http://ant.apache.org/
[5]http://www.jdom.org/

PPs	Paths			Nodes			Paths Ratio			Nodes Ratio			Max Ratio of TR/TP		
	BF	PG	SC	BF	PG	SC	BF	PG	SC	BF	PG	SC	BF	PG	SC
9	7	5	5	46	33	33	1.00	0.71	0.71	1.00	0.72	0.72	2	3	3
11	9	7	8	54	42	48	1.00	0.78	0.89	1.00	0.78	0.89	2	3	3
27	14	12	12	210	184	177	1.00	0.86	0.86	1.00	0.88	0.84	9	16	17
27	17	15	17	160	141	166	1.00	0.88	1.00	1.00	0.88	1.04	3	9	8
35	25	25	22	229	225	200	1.00	0.96	0.88	1.00	0.98	0.87	6	7	8
38	18	13	17	196	155	181	1.00	0.72	0.94	1.00	0.79	0.92	8	17	12
46	30	24	27	366	304	339	1.00	0.80	0.90	1.00	0.83	0.93	6	10	10
63	36	22	26	576	415	441	1.00	0.61	0.72	1.00	0.72	0.77	9	16	14
69	49	42	44	536	465	485	1.00	0.86	0.90	1.00	0.87	0.90	7	11	11
71	37	24	25	699	492	484	1.00	0.65	0.68	1.00	0.70	0.69	12	19	16
84	70	50	52	792	618	621	1.00	0.71	0.74	1.00	0.78	0.78	7	13	13
98	65	57	56	1086	965	947	1.00	0.87	0.86	1.00	0.89	0.87	8	12	12
101	66	50	52	984	788	771	1.00	0.76	0.79	1.00	0.80	0.78	11	18	11
122	81	70	63	1685	1521	1393	1.00	0.86	0.78	1.00	0.90	0.83	13	14	14
170	105	91	93	2176	1921	1927	1.00	0.87	0.89	1.00	0.88	0.89	9	26	13
1024	913	776	623	26,285	24,005	21,398	1.00	0.85	0.68	1.00	0.91	0.81	15	26	19
1141	982	913	761	39,370	37,267	32,934	1.00	0.93	0.78	1.00	0.95	0.84	16	29	19
1844	1265	395	376	28,278	12,875	11,607	1.00	0.31	0.30	1.00	0.46	0.41	14	273	77
Total	**3789**	**2590**	**2279**	**103,728**	**82,416**	**74,152**									
Weighted Average							**1.00**	**0.68**	**0.60**	**1.00**	**0.79**	**0.71**			

Table II
PATHS AND NODES FOR OPEN SOURCE METHODS

more than 1000 prime paths. The unweighted and weighted averages are quite different so this table shows both. The unweighted averages are straight averages of the numbers in the column, and the weighted averages are computed from the totals. With execution speed in particular, the weighted averages are heavily influenced by the last subject.

PPs	Time (seconds)			Time Ratio		
	BF	PG	SC	BF	PG	SC
9	0.002	0.0007	0.0004	1.00	0.35	0.20
11	0.004	0.0007	0.0004	1.00	0.18	0.10
27	0.048	0.0170	0.0160	1.00	0.35	0.33
27	0.023	0.0130	0.0030	1.00	0.57	0.13
35	0.035	0.0160	0.0230	1.00	0.46	0.66
38	0.045	0.0160	0.0190	1.00	0.36	0.42
46	0.064	0.0230	0.0370	1.00	0.36	0.58
63	0.140	0.0360	0.1200	1.00	0.26	0.86
69	0.200	0.0390	0.1300	1.00	0.20	0.65
71	0.230	0.0500	0.1800	1.00	0.22	0.78
84	0.330	0.0660	0.2900	1.00	0.20	0.88
98	0.670	0.0870	0.3600	1.00	0.13	0.54
101	0.330	0.0660	0.2900	1.00	0.20	0.88
122	1.070	0.1600	0.9600	1.00	0.15	0.90
170	3.610	0.3500	2.4400	1.00	0.10	0.68
1024	1340.000	142.0000	4040.0000	1.00	0.11	3.01
1141	3490.000	184.0000	5680.0000	1.00	0.05	1.63
1844	1680.000	235.0000	35,500.0000	1.00	0.14	21.13
Total	**6516.801**	**561.9404**	**45,224.8400**			
			Unweighted Average	**1.00**	**0.24**	**0.56**
			Weighted Average	**1.00**	**0.09**	**6.94**

Table III
EXECUTION TIME FOR OPEN SOURCE METHODS

Table IV shows the results of the test paths in the second stage of the experiment on the modified methods. The results are consistent with those in the first stage.

Table V shows the measured execution time for the modified methods. The results are similar to these in table III. The prefix-graph based algorithm spent the least time to generate the test paths for 17 subjects. However, the set-covering algorithm took more time than the brute force approach for 8 out of last 9 subjects when the number of prime paths gets bigger.

PPs	Time (second)			Time Ratio		
	BF	PG	SC	BF	PG	SC
26	0.016	0.0110	0.0060	1.00	0.69	0.38
35	0.033	0.0180	0.0260	1.00	0.55	0.79
62	0.130	0.0300	0.0800	1.00	0.23	0.62
63	0.004	0.0007	0.0004	1.00	0.18	0.10
68	0.150	0.0400	0.1700	1.00	0.27	1.13
81	0.170	0.0500	0.2300	1.00	0.29	1.35
82	0.300	0.0600	0.2000	1.00	0.20	0.67
88	0.220	0.0560	0.3900	1.00	0.25	1.77
91	4.480	0.4800	3.3100	1.00	0.11	0.74
97	0.710	0.0800	0.5600	1.00	0.11	0.79
105	0.490	0.1100	0.7600	1.00	0.22	1.55
115	0.620	0.1300	0.9100	1.00	0.21	1.47
121	0.700	0.1100	1.1300	1.00	0.16	1.61
157	2.520	0.3200	4.6700	1.00	0.13	1.85
161	2.470	0.3200	4.1300	1.00	0.13	1.67
183	1.790	0.3400	6.0900	1.00	0.19	3.40
189	3.610	0.3500	2.4400	1.00	0.10	0.68
293	10.400	1.1800	33.9000	1.00	0.11	3.26
369	26.200	2.7900	63.5000	1.00	0.11	2.42
Total	**55.013**	**6.4757**	**122.5024**			
			Unweighted Averages	**1.00**	**0.22**	**1.06**
			Weighted Averages	**1.00**	**0.12**	**2.23**

Table V
EXECUTION TIME FOR MODIFIED METHODS

PPs	Paths			Nodes			Paths Ratio			Nodes Ratio			Max Ratio of TR/TP		
	BF	**PG**	**SC**	**BF**	**PG**	**SC**	**BF**	**PG**	**SC**	**BF**	**PG**	**SC**	**BF**	**PG**	**SC**
26	10	5	8	150	117	125	1.00	0.50	0.80	1.00	0.78	0.83	10	15	11
35	15	10	11	198	155	153	1.00	0.67	0.73	1.00	0.78	0.77	9	13	14
62	35	21	26	476	369	385	1.00	0.60	0.74	1.00	0.78	0.81	9	13	11
63	41	24	29	625	456	511	1.00	0.59	0.70	1.00	0.73	0.82	7	18	9
68	37	20	25	559	443	419	1.00	0.54	0.67	1.00	0.79	0.75	11	36	13
81	55	19	30	674	335	407	1.00	0.35	0.55	1.00	0.50	0.60	7	19	16
82	58	42	42	1001	829	803	1.00	0.72	0.72	1.00	0.83	0.80	7	15	12
88	44	23	25	788	574	496	1.00	0.52	0.57	1.00	0.73	0.63	12	34	17
91	40	21	33	826	589	728	1.00	0.53	0.83	1.00	0.71	0.88	11	28	26
97	50	30	33	873	591	557	1.00	0.60	0.66	1.00	0.68	0.64	13	33	16
105	60	29	32	859	623	558	1.00	0.48	0.53	1.00	0.73	0.65	10	44	16
115	72	53	46	1179	1037	876	1.00	0.74	0.64	1.00	0.88	0.74	10	22	15
121	85	49	54	1335	1029	1031	1.00	0.58	0.63	1.00	0.77	0.77	9	28	18
157	94	53	54	2118	1425	1332	1.00	0.56	0.57	1.00	0.67	0.63	14	45	25
161	98	62	66	1816	1433	1342	1.00	0.63	0.67	1.00	0.79	0.74	14	36	15
183	103	70	61	2060	1647	1351	1.00	0.68	0.59	1.00	0.80	0.66	12	57	20
189	113	79	77	2515	2150	1973	1.00	0.70	0.68	1.00	0.85	0.78	7	63	21
293	150	78	77	2929	2043	1906	1.00	0.52	0.51	1.00	0.70	0.65	13	132	25
369	244	181	158	4941	4030	3604	1.00	0.74	0.65	1.00	0.81	0.73	13	60	21
Total	**1404**	**869**	**887**	**25,922**	**19,875**	**18,557**									
Weighted Average							**1.00**	**0.62**	**0.63**	**1.00**	**0.77**	**0.72**			

Table IV
PATHS AND NODES FOR MODIFIED METHODS

C. Analysis

The data in tables II and IV show that the set-covering based and prefix-graph based algorithms generated fewer test paths and nodes than the current brute force algorithm. On average, the two new algorithms saved 30%-40% of the test paths and 20%-30% of the nodes for the open source methods and the modified methods. The methods with more prime paths had higher savings. For example, the last method in table II had a saving of 70%.

Based on our analysis, it seems that the prefix-graph and set-covering based algorithms yield the greatest savings for methods that have "complex" nested loops. Although we were not able to quantify this, it seems that nested loops led to more overlaps among the prime paths.

The data in tables II and IV also do not definitely tell us which algorithm is better. Theoretically, the approximation ratio of the greedy set cover algorithm is $2 \ln n$ while the prefix graph algorithm is only 3 [2]. Thus, the prefix-graph algorithm is expected to yield fewer test paths. However, the experimental results do not always comply with the theoretical worst-case bounds. The prefix graph algorithm gave better results for 23 subjects, the set-cover algorithm for 11 subjects, and they were the same for 3 subjects.

Tables III and V show the prefix-graph based algorithm was faster than the brute force algorithm for all 37 subjects, and faster than the set-covering based algorithm for 31 of 37 subjects. The set-covering based algorithm was faster than the brute force algorithm for 24 subjects; however, for 14 subjects, the set-covering based algorithm was slower than the brute force algorithm.

We observed that when the graph has few prime paths, the set-covering based algorithm was faster than the other algorithms. When graphs have more prime paths, the set-covering based algorithm becomes slower. For example, consider the last subject in table III. The prefix-graph based algorithm finished in less than 4 minutes; while the brute force algorithm took about 25 minutes and the set-covering based algorithm took almost 10 hours! We conclude that the set-covering algorithm may be better for small graphs, but may simply be too slow with large graphs.

For some subjects, the maximum ratio of TR to TP is big and the ratio of test requirements that are covered by one test path over all test requirements is high as well. We do need to apply other algorithms to control the maximum ratio of TR to TP. Dynamic programming techniques may solve this problem, as mentioned in the future work section.

D. Recommendations

These data strongly indicate that both the prefix-graph and set-covering algorithms generate fewer test paths and nodes than the brute force algorithm, particularly when methods have nested loops and otherwise complex structure. It seems harder to quantify exactly which and when the algorithms should be used. There is probably no controversy that getting fewer test paths or total nodes is always better, however, the long execution times for the subject with 1844 prime paths is problematic. No tester wants to wait 10 hours for the set-cover algorithm to complete, particularly when it only saves 19 test paths over the prefix-based algorithm!

We might wish for a preliminary analysis of the graphs to decide which algorithm would be best. In the absence of such an ability, however, we plan to incorporate the prefix-

based algorithm in our tool. It does not always generate as good solutions as the set-covering algorithm, but also does not appear to ever take unacceptably long. In our future work, we will adapt our solutions to other objectives (see Table I).

E. Threats to Validity

As usual with most software engineering studies, there is no way to show that the subjects selected are representative. Thus an external threat to validity is that these results may not hold on other programs. It is important to note that the results do not depend on the behavior of the software, only on the structure of the graph. The key elements that affect the results are the number of prime paths and the amount of overlap among them. An obvious internal threat is that the implementations of these algorithms may be imperfect. The results could also be affected by the use of a different splitting algorithm or a different test minimization algorithm.

V. Conclusions and Future Work

This research presented and compared three solutions for the minimum test paths problem: the brute force, the prefix-graph based, and the set-covering based solutions. They were compared on 37 methods, 18 from open source programs and 19 constructed by hand. Graphs were created by hand, and prime paths were generated automatically. Then all three solutions were used to generate test paths for the same set of prime paths. Both the prefix-graph based and set-covering based solutions generated fewer test paths than the brute force solution, however neither the prefix-graph nor the set-covering solution was always better than the other and we could not quantify which solution is preferable or when. The set-covering based solution took an extremely long time with the largest sets of prime paths, so the prefix-graph based solution seems preferable.

In the future it would be helpful to try other algorithms from the shortest superstring literature. It would also be helpful to quantify properties of methods so that we could predetermine which algorithm would work better. The number of prime paths is an easy measure, but does not seem to be accurate. Checking overlaps among the prime paths may work better. Further experimentation could also measure other factors, such as the number of nodes and the structural complexity. It is also possible that different test minimization algorithms could yield different solutions. Furthermore, we plan to apply other algorithms such as dynamic programming to our set-covering and prefix-graph based solutions to solve the fewest total nodes, the minimizing maximum ratio of TR to TP, the fewest test paths subject to a bounded ratio of TR to TP, and the fewest total nodes subject to a bounded TR to TP ratio problems.

References

[1] A. V. Aho and D. Lee, "Efficient algorithms for constructing testing sets, covering paths, and minimizing flows," *AT&T Bell Laboratories Tech. Memo*, vol. 159, 1987.

[2] V. V. Vazirani, *Approximation Algorithms*. Springer, 2000.

[3] D. Hochbaum, *Approximation Algorithms for NP-Hard Problems*, 1st ed. Course Technology, July 1996.

[4] P. Ammann, J. Offutt, W. Xu, and N. Li, "Coverage computation web applications," Online, 2008, http://cs.gmu.edu:8080/offutt/coverage/, last access September 2011.

[5] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[6] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, 2010.

[7] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering Methodology*, vol. 2, pp. 270–285, July 1993.

[8] T. Y. Chen and M. F. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.

[9] J. R. Horgan and S. London, "A data flow coverage testing tool for C," in *Proceedings of the Symposium of Quality Software Development Tools*, 1992, pp. 2–10.

[10] A. J. Offutt, J. Pan, and J. M. Voas, "Procedures for reducing the size of coverage-based test sets," in *Proceedings of the Twelfth International Conference on Testing Computer Software*, 1995, pp. 111–123.

[11] J. Gallant, D. Maier, and J. Storer, "On finding minimal length superstrings," *Journal of Computer and System Sciences*, vol. 20, pp. 50–58, 1980.

[12] J. Tarhio and E. Ukkonen, "A greedy approximation algorithm for constructing shortest common superstrings," *Theoretical Computer Science-International Symposium on Mathematical Foundations of Computer Science*, vol. 57, no. 1, pp. 486–491, 1988.

[13] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, "Linear approximation of shortest superstrings," *Journal of the ACM*, vol. 41, no. 4, pp. 630–647, 1994.

[14] M. Weinard and G. Schnitger, "On the greedy superstring conjecture." *SIAM Journal on Discrete Mathematics*, vol. 20, no. 2, pp. 502–522, 2006.

[15] H. J. Romero, C. A. Brizuela, and A. Tchernykh, "An experimental comparison of approximation algorithms for the shortest common superstring problem," in *Proceedings of the Fifth Mexican International Conference in Computer Science*, 2004, pp. 27–34.

[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2009.