Received July 19, 2019, accepted August 25, 2019, date of publication September 13, 2019, date of current version September 30, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2941429

A Time and Space-Efficient Compositional Method for Prime and Test Paths Generation

EBRAHIM FAZLI AND MOHSEN AFSHARCHI
Department of Computer Engineering, University of Zanjan, Zanjan 45371-38891, Iran
Corresponding author: Ebrahim Fazli (efazli@znu.ac.ir)

ABSTRACT This paper investigates the problem of prime and test paths generation, which is an important problem in ensuring path coverage in software testing. Most existing methods for prime/test paths generation have little success in generating the set of all prime/test paths of structurally complex programs with high Npath complexity. This paper puts forward two novel methods for the generation of prime and test paths of highly complex programs, namely a vertex-based algorithm and a compositional method. The proposed vertex-based method enables a time-efficient approach for the generation of all prime paths, and the compositional method provides a highly time and space-efficient method for the generation of prime and test paths in cyclic control flow graphs with extremely large Npath complexity. We also implement the proposed algorithms as a software toolset for the generation of prime and test paths. Our experimental results on a set of complex programs indicate that the proposed approaches significantly outperform existing methods, especially for large and structurally complex programs.

INDEX TERMS Software testing, structural testing, prime path coverage.

I. INTRODUCTION

Prime Paths (PPs) generation is an important problem in software testing as PPs subsume other structural testing criteria such as edge and branch coverage as well as enabling test data generation through generating Test Paths (TPs). A *prime path* is a maximal simple path in a directed graph; i.e., a simple path that cannot be extended further without breaking its simplicity property. In PPs generation, researchers [3], [4], [9] consider the Control Flow Graph (CFG) of the Program Under Test (PUT), where each node/vertex captures a block of assignments often ending in a conditional statement, and each arc (b_1, b_2) represents the transfer of control from a block b_1 to another block b_2. This paper enhances the state-of-the-art in PPs and TPs generation for large and structurally complex programs through a compositional method.

Most existing methods for PPs/TPs generation have had little success in developing algorithms and tools that can generate PPs/TPs in a highly time/space efficient fashion for complex and large programs. For example, Ntafos and Hakimi [16] study the minimum path cover problem in a graph-theoretic setting where they show that finding the minimum number of test paths that cover all vertices and branches is NP-hard. Ammann and Offutt [3] define the PPs generation problem and propose a dynamic programming algorithm that can generate all PPs. Their approach extends each path as long as there is an edge that can be included while preserving the simplicity property of the path; i.e., an *edge-based* method. They also implement their approach as a web-based tool [4]. Aho and Lee [2] leverage minimum flow algorithms towards generating minimum number of test paths that ensure node coverage. Li *et al.* [12] investigate minimum TPs generation as a shortest superstring problem, which is an NP-complete problem, and present some polynomial approximation algorithms. Dwarakanath and Jankiti [9] exploit Max-Flow/Min-Cut algorithms [10] to generate minimum number of TPs. Bures and Ahmed [8] formulate TPs generation in a broader and more abstract setting where they take a model of PUT, a coverage criterion (e.g., prime path), the priority of each component in PUT, and the optimality criterion (e.g., minimum number of paths). Then, they generate a test set that meets the optimality criterion. Search-based methods exploit heuristic search techniques to generate PPs/TPs. For instance, Hoseini and Jalili [11] present a model by means of Genetic Algorithm (GA) to generate PPs/TPs of CFGs extracted from sequence diagrams. Sayyari and Emadi [17] utilize ant colony optimization to generate TPs. Srivastava *et al.* [18] take a Markov Chain model of PUT and generate optimized test sequences. Bidgoli *et al.* [6] exploit swarm intelligence algorithms along

The associate editor coordinating the review of this manuscript and approving it for publication was Hui Liu.

with a normalized fitness function to ensure prime paths coverage. Lin and Yeh [13] and Bueno and Jino [7] present GA-based methods for prime paths coverage.

What the aforementioned methods have in common is their limited power in generating PPs/TPs of structurally complex programs with extremely large Npath complexity (in the scale of a few hundreds of billions) [15], where Npath complexity captures the number of execution paths while limiting the loops to at most one iteration. There are several reasons behind this deficiency. First, the number of PPs could still be extremely large even in small programs with loops. Second, time and space efficiency may sometimes be conflicting goals. Third, space cost of existing methods is significant mostly due to inefficient data structures and memory management methods used in such methods. Fourth, most existing methods do little to reuse the information produced during PPs generation for TPs generation, thereby increasing costs.

This paper presents a compositional method for PPs/TPs generation in a highly time and space-efficient fashion that significantly outperforms existing methods. The proposed method first generates the component graph of the input CFG and processes each node of the component graph (i.e., each SCC of the CFG) in isolation.[1] We first present a vertex-based PPs generation algorithm that we run on each SCC. Our vertex-based algorithm associates a list of partial paths to each vertex and keeps extending those paths while preserving simplicity. Upon termination, the list associated to each vertex v contains the PPs ending in v. In each iteration of the algorithm, the list of v is expanded by extending the PPs in v's predecessors, and eliminating redundant paths in the list of v. The proposed vertex-based algorithm is more time-efficient than Ammann and Offutt's algorithm [3], nonetheless, it is less space efficient due to the cumulative propagation of partial paths in the lists associated to vertices. To manage this space inefficiency, we prune the vertex lists after each update, and in a modular fashion, apply our vertex-based algorithm on individual SCCs. For each SCC, we generate different types of PPs (e.g., PPs ending in exit vertices of an SCC), which we use in the merging phase of the proposed method. In order to generate the PPs of the input CFG, the proposed merging method combines partial PPs of individual SCCs on each prime path of the CFG's component graph. To enable a highly space-efficient merging method, we generate the resulting PPs on the secondary memory.

In order to validate the proposed algorithms, we have implemented them as a toolset and have compared their time and space efficiency with respect to the state-of-the-art in PPs generation. Moreover, we have conducted several experiments on two classes of programs. The first category includes five CFGs (adopted from [5]), which represent five programs in the Apache Commons library. The maximum Npath complexity amongst the first five CFGs is almost 33500. The second category includes three synthetic CFGs that we have manually generated by including cycles and conditionals in them. The Npath complexity of these programs is up to 612 billion. Our experimental results on these eight programs indicate that the proposed SCC-based approach significantly outperforms existing methods and our vertex-based method in terms of both time and space efficiency (see Section VII for details). Wherever existing methods fail to generate the prime paths of some programs (amongst these 8 programs), the proposed compositional method succeeds.

Organization. Section II introduces some basic concepts related to directed graphs, prime paths and test paths. Section III states the problem of prime paths generation. Section IV presents a vertex-based algorithm for PPs generation. Subsequently, Section V puts forward a highly time and space-efficient method for compositional generation of PPs. Section VI presents an efficient method for TPs generation. Section VII talks about our experimental results. Finally, Section VIII makes concluding remarks and discusses future work.

II. PRELIMINARIES

This section presents some graph-theoretic concepts that we rely on throughout this paper. A *directed graph* $G = (V, A)$ contains a set of vertices V and a set of arcs $(v, v') \in A$, where $v, v' \in V$. A *simple path* p in G is a sequence of vertices $v_1, \cdots, v_k$, where each arc (v_i, v_{i+1}) belongs to A for $1 \leq i < k$ and $k > 0$, and no vertex appears more than once in p unless $v_1 = v_k$. A vertex v' is *reachable* from another vertex v iff (if and only if) there is a simple path starting at v and ending in v'. A Strongly Connected Component (SCC) in G is a sub-graph $G_c = (V', A')$, where $V' \subseteq V$ and $A' \subseteq A$, and for any pair of vertices $v_s, v_d \in V'$, v_s and v_d are reachable from each other. Tarjan [19] presents a method for identifying the SCCs of graphs and constructing their *component graph*, which is a Directed Acyclic Graph (DAG) whose every vertex is an SCC. We now present some definitions related to the focus of this work; i.e., prime and test paths generation. The Control Flow Graph (CFG) of a program is a directed graph whose vertices are blocks of statements ending in conditionals and branches, and its arcs capture the transfer of control between two statement blocks. A CFG often has a *start vertex* that captures the block of statement starting with the first instruction of the program, and has some *end vertices* representing the blocks of statements that end in a halt/exit/return instruction. For example, Figure 1 illustrates the CFGs (adopted from [5]) of two methods of a class in the Apache Commons library.

Definition 1 (Prime Path): A *prime path* is a maximal simple path in a directed graph; i.e., a simple path that cannot be extended further without breaking its simplicity property (e.g., prime path $\langle 2, 4, 5, 7, 2\rangle$ in Figure 1(b)).

Definition 2 (Complete Prime Path): A prime path that emanates from the start vertex and terminates at some end vertex (e.g., prime path $\langle$Start, 1, 2, 4, 6, 8, End$\rangle$ in Figure 1(b)).

Definition 3 (Component Graph of CFGs): The component graph of a CFG $G = (V, A)$, called CCFG, is a DAG

[1] Throughout this paper the terms compositional and SCC-based are used interchangeably

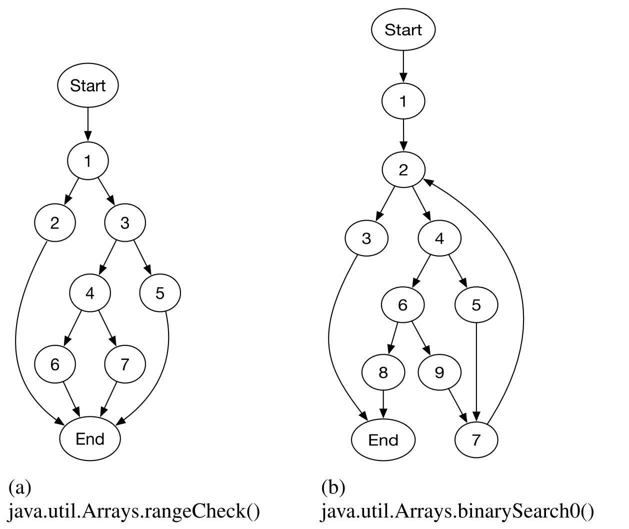
Start
1
2
3
4
5
6
7
End
(a)
java.util.Arrays.rangeCheck()
Start
1
2
3
4
5
6
7
8
9
End
(b)
java.util.Arrays.binarySearch0()

FIGURE 1. **CFGs for two open source methods.**

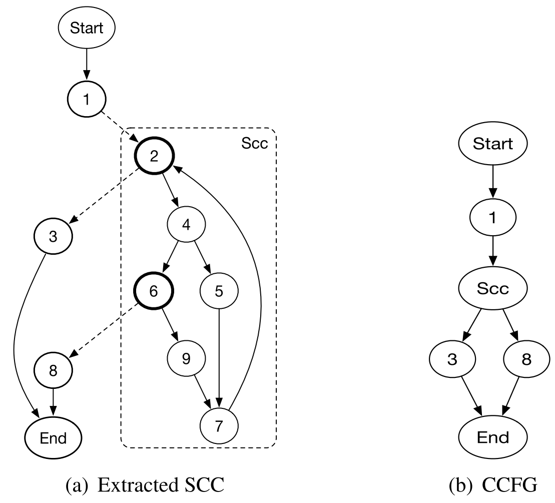

FIGURE 2. **SCC and CCFG extracted from CFG Fig.1(b).**

whose vertices are the SCCs of G, and any arc $(v_i, v_j) \in A$ starts in an SCC$_i$ and ends in SCC$_j$ (see Fig 2(b)).

In the following definitions, let $G = (V, A)$ be a CFG and $C = (V_c, A_c)$ be an SCC in the CCFG of G; i.e., C is a vertex in CCFG of G,

Definition 4 (SccEntryVertex): A vertex $v_e \in V_c$ is an SccEntryVertex of C *iff* $\exists v : v \in V \wedge v \notin V_c : (v, v_e) \in A$. (e.g., Vertex 2 in Fig 2(a)).

Definition 5 (SccExitVertex): A vertex $v_e \in V_c$ is an SccExitVertex of C *iff* $\exists v : v \in V \wedge v \notin V_c : (v_e, v) \in A$. (e.g., Vertices 2 and 6 in Fig 2(a)).

Definition 6 (SccEntryExitPath): An SccEntryExitPath is an acyclic simple path from $v_s \in V_c$ to $v_t \in V_c$, where v_s is an SccEntryVertex and v_t is an SccExitVertex of C. (e.g., path $\langle 2 \rangle$ and $\langle 2, 4, 6 \rangle$ in Fig 2(a)).

Definition 7 (SccExitPath): An *exit path* of C is a simple path that starts in $v_s \in V_c$ and ends in $v_t \in V_c$, where v_s is not an SccEntryVertex but v_t is an SccExitVertex of C. We call p an SccExitPath *iff* p is a maximal exit path; i.e., p is not a proper subpath of any other exit path in C. (e.g., path $\langle 4, 6, 9, 7, 2 \rangle$ in Fig 2(a))

Definition 8 (SccEntryPath): An *entry path* of C is a simple path that starts in $v_s \in V_c$ and ends in $v_t \in V_c$, where v_s is an SccEntryVertex but v_t is not an SccExitVertex of C. We call p an SccEntryPath *iff* p is a maximal entry path; i.e., p is not a proper subpath of any other entry path in C. (e.g., the path $\langle 2, 4, 5, 7 \rangle$ in Fig 2(a))

Definition 9 (SccInternalPrimePath): An SccInternal-PrimePath p of C is a prime path starting in $v_s \in V_c$ and ending at $v_t \in V_c$. Moreover, if p is acyclic, then it must not start at an SccEntryVertex in C or terminate in some SccExitVertex in C. (e.g., the prime path $\langle 4, 5, 7, 2, 4 \rangle$ in Fig 2(a))

Definition 10 (SccExitPrimePath): A prime path p from v_s to v_t is an SccExitPrimePath *iff* $v_s \in V_c$ and $v_t \in V$ is an End vertex of G. (e.g., the prime path $\langle 4, 6, 9, 7, 2, 3, End \rangle$ in Fig 2(a))

Definition 11 (SccEntryPrimePath): A prime path p from v_s to v_t is an SccEntryPrimePath *iff* v_s is the Start vertex of G and $v_t \in V_c$. (e.g., the prime path $\langle Start, 1, 2, 4, 5, 7 \rangle$ in Fig 2(a))

Definition 12 (CompletePrimePath): A prime path p from v_s to v_t is a CompletePrimePath *iff* v_s is the Start vertex of G and v_t is an End vertex in G. (e.g., the prime path $\langle Start, 1, 2, 4, 6, 8, End \rangle$ in Fig 2(a))

III. PROBLEM STATEMENT

The problem of generating PPs of a directed graph is an important problem as even small graphs with a few cycles may have a huge number of PPs. In addition to the algorithmic importance of generating the set of PPs of a graph, the practical implications are also significant as there is a need for software tools that can generate all PPs of programs with high Npath complexity in a time and space-efficient manner. We formulate the problem of generating PPs in a graph-theoretic setting as follows:

Problem 1 (Prime Paths Generation):

- **Input**: A graph $G = (V, A)$ that represents the CFG of a given program, and a start vertex $s \in V$.
- **Output**: The set of prime paths to each vertex $v \in V$.

Ammann and Offutt's state-of-the-art *edge-based* algorithm [3] for PPs generation starts from each vertex $v \in V$, and expands the candidate prime paths emanating from v by including outgoing arcs that preserve the simplicity of the paths. That is, for any vertex v and an arc (v, w), the vertex w is added to the path ending in v if the resulting path remains simple or forms a cycle. If w has no outgoing arcs, then there will not be further expansion. This process continues until no path can be extended further without breaking the simplicity property. Table 1 demonstrates an execution of Ammann and Offutt's algorithm on the CFG in Figure 1(a). Notice that while the number of paths per vertices is manageable, the number of iterations of this algorithm could be as large

TABLE 1. Running the edge-based algorithm [3] on CFG in Figure 1(a).

Vertex	Iteration1	Iteration2	Iteration3	Iteration4	Iteration5
{0}	{0,1}	{0,1,2} {0,1,3}	{0,1,2,8} {0,1,3,4} {0,1,3,5}	{0,1,2,8} {0,1,3,4,6} {0,1,3,4,7} {0,1,3,5,8}	{0,1,2,8}{0,1,3,4,6,8} {0,1,3,4,7,8} {0,1,3,5,8}
{1}	{1,2} {1,3}	{1,2,8} {1,3,4}{1,3,5}	{1,2,8} {1,3,4,6} {1,3,4,7}{1,3,5,8}	{1,2,8} {1,3,4,6,8} {1,3,4,7,8}{1,3,5,8}	{1,2,8}{1,3,4,6,8} {1,3,4,7,8}{1,3,5,8}
{2}	{2,8}	{2,8}	{2,8}	{2,8}	{2,8}
{3}	{3,4}{3,5}	{3,4,6}{3,4,7}{3,5,8}	{3,4,6,8}{3,4,7,8} {3,5,8}	{3,4,6,8}{3,4,7,8} {3,5,8}	{3,4,6,8}{3,4,7,8} {3,5,8}
{4}	{4,6} {4,7}	{4,6,8}{4,7,8}	{4,6,8} {4,7,8}	{4,6,8}{4,7,8}	{4,6,8} {4,7,8}
{5}	{5,8}	{5,8}	{5,8}	{5,8}	{5,8}
{6}	{6,8}	{6,8}	{6,8}	{6,8}	{6,8}
{7}	{7,8}	{7,8}	{7,8}	{7,8}	{7,8}
{8}	{8}	{8}	{8}	{8}	{8}

as $|V|$, which negatively impacts the time efficiency of their algorithm. Thus, it is desirable to devise algorithms that can tackle this deficiency while preserving (and preferably improving) space efficiency.

IV. VERTEX-BASED ALGORITHM

This section presents a time-efficient algorithm (i.e., a solution for Problem 1) that computes the set of all prime paths reaching a vertex in a given CFG. Algorithm 1 takes a digraph $G(V, A)$ (representing the CFG of a program) and a start vertex $s \in V$, and then generates the set of all prime paths reaching each vertex $v_i \in V$ in a list associated to v, denoted $v_i.list$. Initially, the prime paths list of each vertex contains only the vertex itself, and the start vertex s and its immediate successors are inserted in a queue Q. Algorithm 1 performs two kinds of processing on $v_i.list$: (1) extending the paths in the lists of all predecessor vertices of v_i (Lines 6 to 16), and (2) pruning the redundant paths (Lines 17 to 20) in $v_i.list$. A *redundant path* in $v_i.list$ is an acyclic path that (i) is read by both successors of v_i, and (ii) is either extended by at least one successor or covered by another path in $v_i.list$. Each vertex v_i undergoes these processing steps after extraction from Q (Line 2). Algorithm 1 then propagates the wave of updates to the successors of v_i by inserting them in Q (Lines 21 to 23). After exiting the while loop, Algorithm 1 will prune the list of paths in each end vertex of the input CFG (Lines 24 to 27).

Table 2 illustrates the $v_i.list$ of each vertex v_i and the contents of the queue Q upon executing Algorithm 1 on the CFG in Figure 1(a). The 'Queue' column illustrates the contents of Q as vertices are inserted in it starting from Vertex 0 (i.e., the Start vertex). Each column contains the paths that are inserted in the list of each vertex v_i throughout the execution of Algorithm 1. Each list $v_i.list$ (e.g., list of 7) is initialized by $\{v_i\}$ (e.g., {7}). Then, as the wave of reading reaches v_i, the list of v_i is updated by extending the paths in its predecessor (e.g., the predecessor 4 in arc (4, 7)). For example, when the paths in the list of Vertex 4 are read by Vertex 7, each path is extended by the new vertex 7, creating a longer path that ends in 7. The number of paths in each list demonstrates the memory requirements of Algorithm 1. The list of paths belonging to Vertex 8 becomes the largest list as it reads and extends the paths in the lists of vertices 2, 5, 6 and 7. The last entry in Column 8 represens the pruned list of Vertex 8 where no path is subsumed by another.

Lemma 1: Let (v_j, v_i) be an arc in A and q be an acyclic path in $v_j.list$. The condition of the if statement on Line 8 evaluates to true for q and v_i no more than once. That is, each acyclic path $q \in v_j.list$ is read by v_i at most once.

proof 1: Suppose Algorithm 1 has already entered the if statement on Line 8 for some q and v_i. This means that q has been labeled as 'read by v_i'. Thus, next time Algorithm 1 gets to check if q is read by v_i, the condition on Line 8 evaluates to false.

Lemma 2: For any vertex v_j that has an outgoing arc (v_j, v_i), each acyclic path q in $v_j.list$ will eventually be labeled 'read by v_i'.

proof 2: Initially, each vertex v_j includes $\{v_j\}$ as the only path, and this path is not read yet. The for-loop in Line 6 iterates through all incoming arcs of each vertex $v_i \in V$ and will eventually get to (v_j, v_i). As a result, any path $q \in v_j.list$ will be labeled on Line 9 because initially all paths are not read and are acyclic. If new paths are imported in $v_j.list$ from its predecessors in subsequent iterations of the algorithm, then such paths will be labeled as 'read by v_i' unless they form a cycle.

Lemma 3: For each vertex $v_i \in V$, at some finite point in time, $v_i.updateFlag$ will become false and will remain false.

proof 3: Lemmas 1 and 2 imply that all acyclic paths in the predecessors of v_i will eventually be labeled as 'read

Algorithm 1 Vertex-Based Prime Paths Generation

Input: $G(V, A)$ with an outdegree 2; Start vertex $s \in V$

Output: The set of prime paths ending in each vertex $v \in V$.

Initialize: $\forall v_i \in V$, $v_i.list = \{v_i\}$, $v_i.updateFlag = false$, and queue Q with s.

```
1:  while ($Q$ is non-empty) do
2:      Extract $v_i$ from $Q$;
3:      $v_i.updateFlag = false$;
4:      if $v_i = s$ then
5:          Insert the immediate successors of $v_i$ in $Q$;
6:      for each $v_j$ where $(v_j, v_i) \in A$ do
7:          for each acyclic path $q \in v_j.list$ do
8:              if $q$ is not read by $v_i$ then
9:                  Label $q$ as read by $v_i$;
10:                 if $v_i$ does not appear in $q$ or $v_i$ is the first vertex of $q$ then
11:                     $r = q + v_i$;
12:                     Label $q$ as an extended path;
13:                     if $v_i$ is the first vertex of $q$ then
14:                         Label $r$ as a cycle.
15:                     Add $r$ to $v_i.list$;
16:                     $v_i.updateFlag = true$;
17:         if All successors of $v_j$ have read $v_j.list$ then
18:             for each acyclic path $p \in v_j.list$ do
19:                 if ($p$ is an extended path) or ($p$ is covered by some path $p' \in v_j.list$) then
20:                     remove $p$;
21:     if $v_i.updateFlag = true$ then
22:         for each $v_k$ where $(v_i, v_k) \in A$ do
23:             Insert $v_k$ in $Q$.
24: for each end vertex $v_i \in V$ do
25:     for each acyclic path $p \in v_i.list$ do
26:         if ($p$ is an extended path) or ($p$ is covered by some path $p' \in v_i.list$) then
27:             remove $p$;
28: for each $v_i \in V$ do
29:     return $v_i.list$;
```

by v_i', and will keep their status of being 'read'. As such, the condition on Line 8 will never become true again when processing arc (v_j, v_i). Therefore, Algorithm 1 will no longer get to Line 16; i.e., $v_i.updateFlag$ will become false (on Line 3) and will never become true again.

Theorem 1: Algorithm 1 will eventually terminate.

proof 4: Lemma 3 implies that at some finite point in time, the condition in Line 21 will become false for each $v_i \in V$ and will remain false. Thus, Algorithm 1 will eventually stop inserting vertices in Q. Moreover, the remaining vertices in Q are extracted on Line 2 in subsequent iterations of the while-loop. Thus, Q will eventually become empty; i.e., Algorithm 1 exits the while loop.

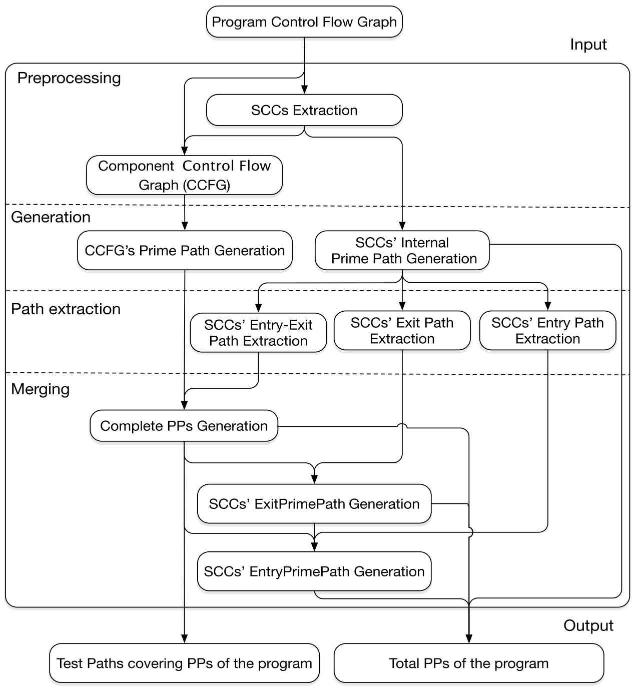
Program Control Flow Graph
Input
Preprocessing
SCCs Extraction
Component Control Flow Graph (CCFG)
Generation
CCFG's Prime Path Generation
SCCs' Internal Prime Path Generation
Path extraction
SCCs' Entry-Exit Path Extraction
SCCs' Exit Path Extraction
SCCs' Entry Path Extraction
Merging
Complete PPs Generation
SCCs' ExitPrimePath Generation
SCCs' EntryPrimePath Generation
Output
Test Paths covering PPs of the program
Total PPs of the program

FIGURE 3. Overview of the SCC-based approach.

V. A COMPOSITIONAL METHOD FOR PRIME PATHS GENERATION

In order to scale up PPs generation, this section presents a compositional method that provides better time and space efficiency in comparison with existing methods for PPs generation. The basic idea behind our compositional method is to (1) compute the component graph of a given CFG, denoted CCFG; (2) calculate the set of prime paths of CCFG and the set of prime paths of each individual SCC in CCFG (*generation* phase); (3) generate different types of prime paths (*path extraction*), and (4) merge the prime paths of SCCs towards generating all prime paths of the original CFG. Figure 3 illustrates the steps of our SCC-based compositional method as well as different types of prime paths we generate. Our proposed method distinguishes four types of prime paths. The first type, called *CompletePrimePaths* (Definition 12), are those paths that emanate from the start vertex and terminate in an end vertex. The second type includes those PPs whose start and end vertices lie in the same SCC, called *SccInternalPrimePaths* (Definition 9). The third type contains *SccExitPrimePaths* (Definition 10) that start from an SCC and end in an end vertex. The fourth type contains those PPs that end in an SCC but start in the start vertex, called *SccEntryPrimePaths* (Definition 11). Subsection V-A discusses the activities of the preprocessing phase in Figure 3. Subsection V-B focuses on the generation of InternalPrimePaths in each SCC and the PPs of CCFG. Subsection V-C presents algorithms for the extraction of incomplete PPs, namely SccEntryExitPaths, SccExitPaths and SccEntryPaths. Finally, Subsection V-D puts forwards a novel and highly efficient method for compositional generation of complete PPs from incomplete PPs.

TABLE 2. The status of the queue *Q* and the list of paths of each vertex when running algorithm 1 on the CFG in Figure 1(a). Vertices 0 and 8 respectively denote the Start and the End vertices.

	Vertices' lists								
Queue	0	1	2	3	4	5	6	7	8
0	{0}	{1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
1		{1} {0,1}	{2}	{3}	{4}	{5}	{6}	{7}	{8}
2		{1} {0,1}	{2}{1,2} {0,1,2}	{3}	{4}	{5}	{6}	{7}	{8}
3			{2}{1,2} {0,1,2}	{3}{1,3} {0,1,3}	{4}	{5}	{6}	{7}	{8}
8				{3}{1,3} {0,1,3}	{4}	{5}	{6}	{7}	{2,8}{1,2,8} {8}{0,1,2,8}
4				{3}{1,3} {0,1,3}	{3,4} {1,3,4} {4}{0,1,3,4}	{5}	{6}	{7}	{2,8}{1,2,8} {8}{0,1,2,8}
5					{3,4} {1,3,4} {4}{0,1,3,4}	{3,5} {1,3,5} {5}{0,1,3,5}	{6}	{7}	{2,8}{1,2,8} {8}{0,1,2,8}
6					{3,4} {1,3,4} {4}{0,1,3,4}	{3,5} {1,3,5} {5}{0,1,3,5}	{6}{4,6}{1,3,4,6} {3,4,6} {0,1,3,4,6}	{7}	{2,8}{1,2,8} {8}{0,1,2,8}
7						{3,5} {1,3,5} {5}{0,1,3,5}	{6}{4,6}{1,3,4,6} {3,4,6} {0,1,3,4,6}	{7}{4,7}{1,3,4,7} {3,4,7}{0,1,3,4,7}	{2,8}{1,2,8} {8}{0,1,2,8}
8									{0,1,2,8}{0,1,3,5,8} {0,1,3,4,7,8} {0,1,3,4,6,8}

A. PREPROCESSING

This phase identifies the set of all SCCs of an input CFG $G = (V, A)$ and constructs the Component Control Flow Graph (CCFG) based on the extracted SCCs and the arcs between them. There are several candidate algorithms such as Kosaraju's algorithm [1] and Tarjan's algorithm [19] that can be used to find all SCCs of a given digraph. We use Tarjan's algorithm [19] that takes a digraph (in this context, a CFG) as an input and produces its component graph whose vertices are SCCs. Each vertex of the input CFG belongs to exactly one SCC. Moreover, any vertex which does not lie on any cycle forms an independent SCC by itself. Figures 2(a) and 2(b) respectively illustrate the result of applying Tarjan's algorithm [19] on the CFG of Figure 1(b).

B. COMPOSITIONAL PRIME PATHS GENERATION

In this phase, we use the vertex-based algorithm of Section IV to separately generate the prime paths of all extracted SCCs and the constructed CCFG. We consider the SCCs with more than one vertex. Running the vertex-based algorithm on the constructed CCFG in Figure 2(b) yields two prime paths demonstrated in the first column of Table 3. Moreover, the second column of Table 3 illustrates the prime paths we generate by applying the vertex-based algorithm on the SCC of Figure 2(a). According to the definition of SccInternalPrimePaths, some generated prime paths of the SCCs may not necessarily belong to the main CFG's prime paths. We eliminate these paths. The second column of Table 4 illustrates the remaining paths.

C. PATH EXTRACTION

This phase involves three tasks including the extraction of SccEntryExitPaths (Definition 6), SccExitPaths (Definition 7) and SccEntryPaths (Definition 8) using Algorithms 2, 3, and 4. We run Algorithm 2 on each SCC of the CFG. Algorithm 2 takes all internal prime paths (computed by Algorithm 1) of a given SCC along with a specified entry vertex v_{en} and an exit vertex v_{ex} (Lines 4 to 9). Algorithm 2 then produces a set of paths starting from v_{en} and ending at v_{ex}. If an entry vertex coincides with the exit one, this vertex itself is reported as an entry-exit path. If there is a path that is a subpath of another path, then Algorithm 2 removes it (Lines 10 to 13). The SCC-based approach applies Algorithm 2 on all pairs of the entry and the exit vertices of the given SCC. The third column of Table 3 illustrates the output of this algorithm when it takes the SCC in Figure 2(a).

Algorithm 2 SccEntryExitPath Extraction

```
Input: $SccInternalPrimePaths$, $SccEntryVertex\ v_{en}$, $SccExitVertex\ v_{ex}$
Output: $SccEntryExitPath(v_{en}, v_{ex})$
Initialize: $L = \emptyset$;
1: if $v_{en}$ is equal to $v_{ex}$ then
2:     return $v_{en}$;
3: else
4:     for each path $p \in SccInternalPrimePaths$ do
5:         if $p$ includes $v_{en}$ and $v_{ex}$ then
6:             $indexS$ = position of $v_{en}$ in $p$;
7:             $indexE$ = position of $v_{ex}$ in $p$;
8:             $q$ = subpath $p$ from $indexS$ to $indexE$;
9:             append $q$ to list $L$;
10: for each path $p \in L$ do
11:     for each path $k \in L$ do
12:         if $k$ is a subpath of $p$ then
13:             remove $k$;
14: return $L$;
```

We devise Algorithm 3 to list the set of prime paths that terminate at an exit vertex in some SCC. This algorithm takes all the internal prime paths of the input SCC as well as a specified exit vertex v_{ex}. Algorithm 3 first creates a list L of longest acyclic simple paths that (1) end at v_{ex}, and (2) are extracted from each internal prime path that includes v_{ex} (Lines 1 to 5). Then, it removes from L any path covered by another path in L (Lines 6 to 9). The fourth column of Table 3 demonstrates the result of applying this algorithm on the SCC of Figure 2(a).

Similar to the SccExitPath extraction, the SCC-based approach employs Algorithm 4 to generate all prime paths in a given SCC starting at an entry vertex v_{en}. The inputs of this algorithm include the set of all internal prime paths of the given SCC and v_{en}. Algorithm 4 first creates a list L of longest acyclic simple paths that start at v_{en} and are extracted from some internal prime path of the SCC (Lines 1 to 5). Then, it removes from L any path that is a subpath of another path (Lines 6 to 9). We use Algorithm 4 for all entry vertices of each SCC to generate all SccEntryPaths. The fifth column of Table 3 illustrates the results of applying Algorithm 4 on the SCC in Figure 2(a).

D. MERGING

The purpose of this phase is to generate all prime paths of the input CFG without processing the CFG as a whole; i.e., *compositional* PPs generation. The merging phase takes all generated prime paths of the CCFG as well as selected paths in the previous step. Then, it yields all CompletePrimePaths, SccExitPrimePaths and SccEntryPrimePaths of the CFG using Algorithms 5, 6, and 7 respectively. Algorithm 5 generates all CompletePrimePaths of the main CFG. This algorithm takes all prime paths of the CCFG as well as SccEntryExit paths of all SCCs and produces all complete

Algorithm 3 SccExitPath Extraction

```
Input: $SccInternalPrimePaths$, $SccExitVertex\ v_{ex}$
Output: $SccExitPaths$ to $v_{ex}$
Initialize: $L = \emptyset$;
1: for each path $p \in SccInternalPrimePaths$ do
2:     if $p$ includes $v_{ex}$ and $p$ is not a cycle started with $v_{ex}$ then
3:         $index$ = position of $v_{ex}$ in $p$;
4:         $q$ = subpath $p$ from the beginning to $index$;
5:         append $q$ to list $L$;
6: for each path $p \in L$ do
7:     for each path $k \in L$ do
8:         if $k$ is a subpath of $p$ then
9:             remove $k$;
10: return $L$;
```

Algorithm 4 SccEntryPath Extraction

```
Input: $SccInternalPrimePaths$, $SccEntryVertex\ v_{en}$
Output: $SccEntryPaths$ from $v_{en}$
Initialize: $L = \emptyset$;
1: for each path $p \in SccInternalPrimePaths$ do
2:     if $p$ includes $v_{en}$ and $p$ is not a cycle started by $v_{en}$ then
3:         $index$ = position of $v_{en}$ in $p$;
4:         $q$ = subpath $p$ from $index$ to the end of $p$;
5:         append $q$ to list $L$;
6: for each path $p \in L$ do
7:     for each path $k \in L$ do
8:         if $k$ is a subpath of $p$ then
9:             remove $k$;
10: return $L$;
```

prime paths of the CFG that run through these SCCs. For each prime path p of CCFG, Algorithm 5 replaces any unexplored *SCC* with all *SccEntryExitPaths* that can be toured with p (Lines 1 to 6). Each replacement generates a new complete prime path of the main CFG. The second column of Table 4 presents the result of applying Algorithm 5 on the first and the third column of Table 3.

Algorithm 6 generates all prime paths that exit SCCs of the CFG. The inputs of this algorithm include the SccExitPaths of all SCCs and the CompletePrimePaths of the CFG. The resulting output contains all prime paths that exit SCCs and finish at an end vertex of the CFG. When a complete prime path p enters into an SCC_i and leaves it through an exit vertex v_{ex}, Algorithm 6 extracts a subpath starting from v_{ex} to the end vertex of p (Lines 1 to 7). Then, it merges all exit paths of SCC_i that terminate at v_{ex} with the aforementioned subpath of p (Line 8 and 9). Each merging results in a new SccExitPrimePath (Line 9). At the end, all redundant paths are removed (Line 10). Using Algorithm 6, we generate all SccExitPrimePaths for the CFG of Figure 2(b), illustrated in the second column of Table 4 and the fourth column of

TABLE 3. All paths generated in multiple phases of the SCC-based approach using the CCFG in Figure 2(b). The start and end vertices are respectively represented by 0 and 10.

CCFG PPs	SccInernalPPs	SccEntryExitPath	SccExitPath	SccEntryPath
{0,1,SCC,3,10} {0,1,SCC,8,10}	{5,7,2,4,6,9}{4,5,7,2,4}{4,6,9,7,2,4}{2,4,5,7,2} {2,4,6,9,7,2}{5,7,2,4,5} {6,9,7,2,4,6} {7,2,4,5,7} {9,7,2,4,6,9}{7,2,4,6,9,7}{6,9,7,2,4,5}	{2}{2,4,6}	{4,6,9,7,2} {5,7,2,4,6} {9,7,2,4,6} {4,5,7,2}	{2,4,6,9,7} {2,4,5,7}

TABLE 4. All prime paths of the CFG in Figure 1(b). The start and end vertices are respectively represented by 0 and 10.

SccInternalPPs	CompletePPs	SccExitPPs	SccEntryPPs
{5,7,2,4,6,9} {4,6,9,7,2,4} {2,4,6,9,7,2} {6,9,7,2,4,6} {9,7,2,4,6,9}{7,2,4,6,9,7}{6,9,7,2,4,5}{4,5,7,2,4} {2,4,5,7,2}{5,7,2,4,5} {7,2,4,5,7}	{0,1,2,3,10} {0,1,2,4,6,8,10}	{4,6,9,7,2,3,10}{5,7,2,4,6,8,10} {9,7,2,4,6,8,10}{4,5,7,2,3,10}	{1,2,4,6,9,7} {1,2,4,5,7}

Algorithm 5 CFG CompletePrimePath Generation

```
Input: $SccEntryExitPaths$, $PrimePaths$ of the $CCFG$
Output: $CompletePrimePaths$ of the $CFG$
Initialize: $L = PrimePaths$ of the $CCFG$;
1: for each path $p \in L$ do
2:     for each unexplored $SCC \in p$ do
3:         for each path $q \in SccEntryExitPaths$ do
4:             if $p$ can tour $q$ then
5:                 r = replace $SCC$ in p with $q$;
6:                 append $r$ to $L$;
7:     remove $p$ from $L$;
8: return $L$;
```

Algorithm 6 SccExitPrimePath Generation

```
Input: $SccExitPaths$, $CompletePrimePaths$
Output: $SccExitPrimePaths$ of the $CFG$
Initialize: $L = \emptyset$;
1: for each path $p \in CompletePrimePaths$ do
2:     for each $SCC$ crossed by $p$ do
3:         for each path $q \in SccExitPath$ do
4:             $v_{ex}$ = last vertex of $q$;
5:             if $p$ leaves $SCC$ with $v_{ex}$ then
6:                 $index$ = position of $v_{ex}$ in $p$;
7:                 $r$ = subpath $p$ from $index$ to the end of $p$;
8:                 $r$ = merge $q$ and $r$;
9:                 append $r$ to $L$;
10: remove redundant paths in $L$;
11: return $L$;
```

Table 3. The result of this process is provided on the third column of Table 4.

The prime paths that enter SCCs are the final type of prime paths we generate. Algorithm 7 takes in the SccEntryPaths, the CompletePrimePaths, and the SccExitPrimePaths obtained in previous steps. If a CompletePrimePath or SccExitPrimePath p enters an SCC_i with entry vertex v_{en},

Algorithm 7 SccEntryPrimePath Generation

```
Input: $SccEntryPaths$, $CompletePrimePaths$, $SccExitPrimePaths$
Output: $SccEntryPrimePaths$ of the $CFG$
Initialize: $L = \emptyset$;
1: for each path $p \in$ ($CompletePrimePaths$ or $SccExitPrimePath$) do
2:     for each $SCC$ crossed by $p$ do
3:         for each path $q \in SccEntryPath$ do
4:             $v_{en}$ = first vertex of $q$;
5:             if $p$ entered $SCC$ through $v_{en}$ then
6:                 $index$ = position of $v_{en}$ in $p$;
7:                 $r$ = subpath p from beginning to $index$;
8:                 $r$ = merge $r$ and $q$;
9:                 append $r$ to $L$;
10: remove redundant paths in $L$;
11: return $L$;
```

Algorithm 7 extracts a subpath of p from the beginning to v_{en} (Lines 1 to 7). Then, it merges all $Scc_i EntryPaths$ starting from v_{en} with the aforementioned subpath of p (Line 8 and 9). At the end, we remove all redundant paths (Line 10). Algorithm 7 generates SccEntryPrimePaths of all SCCs for the CFG of Figure 2(b). In this case, the input of Algorithm 7 includes the second and third columns of Table 4, and the fifth column of the Table 3. The forth column of Table 4 illustrates the resulting output.

VI. TEST PATHS GENERATION

Generating the set of test paths that cover all prime paths of a given CFG requires each prime path to be a subpath of at least one complete test path. Unlike the approach proposed by [3], [4] where all incomplete prime path are extended to reach the start and the end vertices, we devise a new method based on merging, where we obtain a set of complete test paths using all incomplete prime paths. In each SCC, the SccIntenalPrimePaths, SccExitPrimePaths,

TABLE 5. Test paths generated using the SCC-based approach.

EntryExitPaths	TestPaths
{2,4,5,7,2,4,6,9,7,2} {2,4,6,9,7,2,4,6} {2,4,5,7,2,4,6}{2,4,5,7,2,4,5,7,2} {2,4,6,9,7,2,4,6,9,7,2} {2,4,6,9,7,2,4,5,7,2}	{0,1,2,4,6,9,7,2,4,6,9,7,2,3,10}{0,1,2,4,6,8,10} {0,1,2,4,5,7,2,4,6,9,7,2,3,10} {0,1,2,3,10}{0,1,2,4,6,9,7,2,4,6,8,10} {0,1,2,4,5,7,2,4,5,7,2,3,10} {0,1,2,4,5,7,2,4,6,8,10} {0,1,2,4,6,9,7,2,4,5,7,2,3,10}

TABLE 6. Graph Structure and Generated PPs of the input CFGs.

CFG	Graph Structure							Prime Paths					TestPaths
	Nodes	Edges	SCC	SccNodes	SccEdges	Cc	Npath	SccInternal	Complete	SccExit	SccEntry	Total	
1	60	89	4	35	50	31	33411	208	31	181	403	823	316
2	88	129	3	43	58	43	26276	146	584	437	649	1819	1167
3	56	76	2	51	70	22	26598	881	1	203	2334	3419	498
4	60	88	1	29	42	30	21376	21464	159	18490	1680	41793	29381
5	119	186	2	35	116	69	21476	4168	357	16695	79	21299	19136
6	120	143	13	56	60	24	1843200	67	72	540	1608	2287	679
7	184	217	18	78	83	35	2123366400	90	2047	15550	17941	35629	17688
8	215	258	24	103	110	44	611529523200	118	4101	31114	141153	176481	35328

Algorithm 8 SccInternalPrimePath to SccEntryExitPath Conversion

```
Input: $SccInternalPrimePaths$, $SccEntryPaths$, $SccExitPaths$
Output: $SccEntryExitPaths$ covering all $SccInternalPrimePaths$
Initialize: $L = SccInternalPrimePaths$;
1:  for each path $p(v_s, v_t) \in L$ where $p$ is not an $SccEntryExitPath$ do
2:      if $v_s$ is not an $SccEntryVertex$ then
3:          find a path $q \in SccEntryPaths$ that include $v_s$
4:          $indexE$ = position of $v_s$ in $q$;
5:          $r$ = subpath $q$ from the beginning to $indexE$;
6:          $p$ = merge $r$ and $p$ ;
7:      if $v_t$ is not an $SccExitVertex$ then
8:          find a path $q \in SccExitPaths$ that include $v_t$
9:          $indexS$ = position of $v_s$ in $q$;
10:         $r$ = subpath $q$ from $indexE$ to the end;
11:         $p$ = merge $p$ and $r$ ;
12:     for each path $k \in L$ do
13:         if $k$ is a subpath of $p$ then
14:             remove $k$;
15: return $L$;
```

and SccEntryPrimePaths are incomplete prime paths. While we need just SccInternalPrimePaths for test paths generation, we generate SccEntryPaths and SccExitPaths to produce all prime paths of the input CFG, for the sake of generality and completeness. Moreover, these paths will be useful for other testing activities such as test data generation. First, we use Algorithm 8 to generate a set of SccEntryExitPaths that cover all SccInternalPrimePaths. Then, we apply Algorithm 5 to merge these paths with CCFG's complete paths, thereby yielding complete test paths that cover all incomplete prime paths.

The inputs of Algorithm 8 include SccInternalPrimePaths, SccExitPaths, and SccEntryPaths of the given SCC. In each iteration (Line 1), Algorithm 8 takes a non-EntryExitPath p, and transforms it into an EntryExit path. If p does not start from an entry vertex, Algorithm 8 determines an appropriate SccEntryPath and merges it in p (Lines 2 to 6). Also, if p does not finish at an exit vertex, then Algorithm 8 determines an appropriate SccExitPath and merges it in p (Lines 7 to 11). At the end of each iteration, all covered internal prime paths are pruned (Lines 12 to 14). The first Column of Table 5 illustrates the result of applying Algorithm 8 on the generated SccInternalPrimePaths of the SCC in Figure 1(b). The second Column of Table 5 presents all test paths generated by Algorithm 5 using the first column of Table 5 and the first column of Table 3 as its inputs.

VII. EXPERIMENTAL RESULTS

This section evaluates the performance of the proposed prime and test paths generation approaches. Our experimental benchmark includes two sets of CFGs, the first of which

TABLE 7. Execution time and memory consumption of the input CFGs.

CFG	Edge Based			Vertex Based		SCC based	
	WebApp	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)
1	yes	0.088228	0.062	0.07674	0.031	0.073877	0.030
2	yes	0.441872	0.635	0.471598	0.421	0.269541	0.256
3	yes	0.34305	0.679	0.232032	0.067	0.124864	0.065
4	Timeout	4.940941	133.042	4.460267	33.785	1.673184	1.674
5	Timeout	22.84568	1660.958	39.347689	548.220	1.545037	19.723
6	yes	0.907876	0.978	2.380003	0.281	0.447896	0.139
7	Timeout	25.245804	499.798	86.560787	241.599	16.693728	63.699
8	Timeout	147.415781	15525.860	312.610825	3262.446	46.845684	938.064

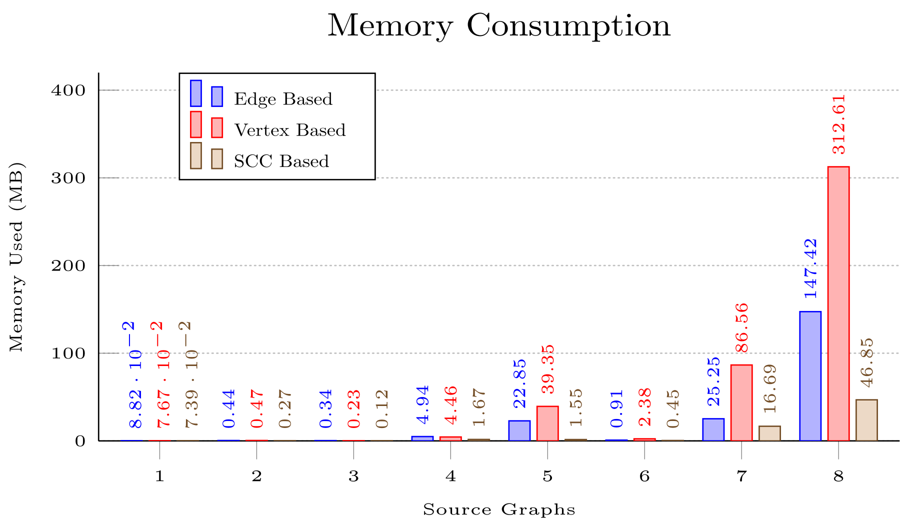

FIGURE 4. Space costs of the vertex-based, edge-based and the SCC-based algorithms on the CFGs of Table 6.

includes five CFGs adopted from PAC [5] (which are taken from Apache Commons libraries). Rows 1 to 5 of Table 6 illustrate the structure of these five CFGs. The selected code has a relatively complicated control structure because CFGs with simple structure (e.g., DAGs) introduce fewer prime paths and it is rather trivial to generate them. In the second set, we synthetically modified three methods from some open source projects by introducing nested loops, and additional conditional statements. Rows 6 to 8 of Table 6 present the structure of these three CFGs. We conduct a comparative study by implementing the three prime paths generation approaches, namely edge-based [3], [4], vertex-based, and SCC-based in the C programming language. Then, we evaluate the time and space efficiency of these three methods on the 8 CFGs introduced in Table 6. We ran all the experiments on an Intel Core i7 machine with 3.6GHz X 8 processors and 16 GB of memory running Ubuntu 17.01 with gcc version 5.4.1.

Table 6 summarizes the structure of the input CFGs as well as their corresponding number of prime paths generated by our implementations. Columns 2 to 4 of Table 6 present the number of nodes, edges, and SCCs of each CFG. The total numbers of nodes and edges of all SCCs are denoted by SccNodes and SccEdges, respectively. Cyclomatic Complexity (CC) [14] and Npath complexity [15] are two well-known metrics of the structural complexity of a program. The CC determines the number of linearly independent complete paths in a given CFG. Columns 7 and 8 provide the CC and Npath complexities of the input CFGs. Columns 9 to 13 show the number of prime paths for each one of the four types of PPs described in the SCC-based approach. The Total column captures the total number of prime paths in each CFG. The last column presents the number of test paths produced with the SCC-based approach.

Table 7 presents the average execution time and maximum memory consumption by the three approaches for each

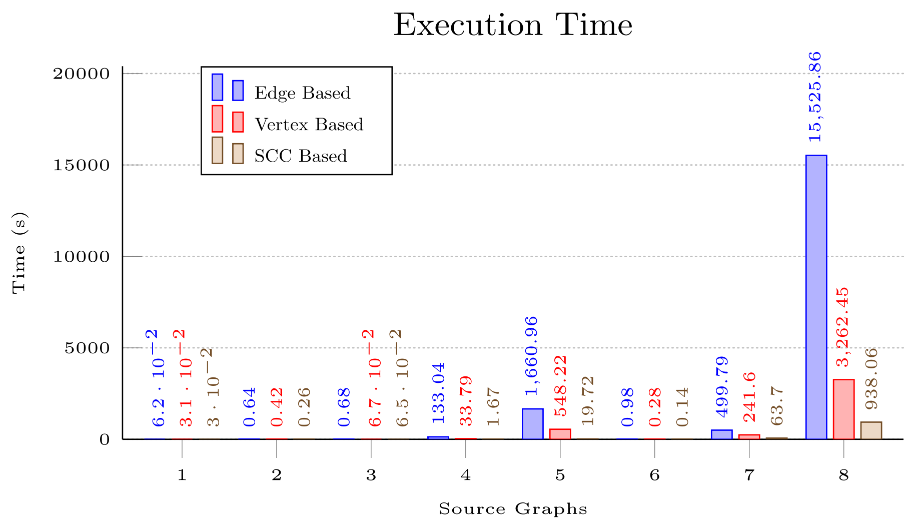

FIGURE 5. **Time costs of the vertex-based, edge-based and the SCC-based algorithms on the CFGs of Table 6.**

subject CFG over five experiments. The WebApp column in Table 7 indicates whether the web-based implementation of the edge-based approach [4] was able to generate a solution on the input CFGs.

For more insightful comparison of the approaches using the values provided in Table 7, Fig. 4 and Fig. 5 depict a graphical summary. The data in Table 7 reflects that the edge-based approach has less memory requirements than the vertex-based approach of Section IV. On average, the edge-based approach consumes 67% less memory for the input CFGs. On the other hand, this approach's time costs are 64% more than that of the vertex-based approach. As mentioned in Section IV, every round of the vertex-based algorithm requires each vertex to save all simple inward paths and then remove the covered paths. This process increases the memory consumption, especially for CFGs with larger indegrees. Additionally, the lower processing time of the vertex-based algorithm (compared to the edge-based approach) is attributed to the rapid growth of the paths.

Table 7 shows that the SCC-based approach outperforms both the vertex-based and the edge-based approaches regarding space and time efficiency for the CFGs of Table 6. On average, the SCC-based approach consumes 62% and 53% less memory than the vertex-based and the edge-based approaches, respectively. Furthermore, the SCC-based approach uses 54% and 83% less processing time than vertex-based and edge-base approaches, respectively. These reductions in memory consumption and processing time have two reasons. First, the vertex-based algorithm as the core of the SCC-based approach performs much better with smaller CFGs. Second, the vertex-based approach generates most of the prime paths (including the SccExit and SccEntry prime paths) in the merging phase without running the main algorithm on the entire CFG.

VIII. CONCLUSION AND FUTURE WORK

This paper proposed two methods for time and space-efficient generation of prime and test paths in structurally complex programs. Specifically, we presented a vertex-based algorithm that takes the Control Flow Graph (CFG) of a program and generates all prime paths reaching each vertex of the CFG. This algorithm outperforms the state-of-the-art in terms of time efficiency; nonetheless, incurs a relatively high space cost. To address this deficiency, we put forward a compositional method for prime and test paths generation of programs with extremely large Npath complexity [15]. The proposed compositional method (i) computes the component graph of the input CFG; (ii) calculates the set of prime paths of each SCC in the component graph, and (iii) generates the set of prime and test paths of the given CFG with very low time and space costs. We implemented and evaluated the proposed methods versus existing approaches, and our experimental results show that the proposed methods significantly outperform the state-of-the-art in dealing with programs that have extremely large Npath complexities (see Table 6).

We are currently working on parallelizing the proposed algorithms and scaling them up further through a GPU-based implementation. Moreover, we are investigating the effectiveness of our compositional method in test data generation by decomposing prime paths into four types of partial prime paths. Another extension of our work relates to developing a benchmark for evaluating the effectiveness and efficiency of algorithms for prime and test paths generation.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, vol. 19. Reading, MA, USA: Addison-Wesley, 1983, no. 3, p. 3.
[2] A. V. Aho and D. Lee, "Efficient algorithms for constructing testing sets, covering paths, and minimum flows," AT&T Bell Lab., New York, NY, USA, Tech. Rep. CSTR159, 1987, pp. 1–15.
[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[4] P. Ammann, J. Offutt, W. Xu, and N. Li. (2008). *Graph Coverage Web Applications*. [Online]. Available: https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage
[5] L. Bang, A. Aydin, and T. Bultan, "Automatically computing path complexity of programs," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 61–72.
[6] A. M. Bidgoli, H. Haghighi, T. Z. Nasab, and H. Sabouri, "Using swarm intelligence to generate test data for covering prime paths," in *Proc. Int. Conf. Fundam. Softw. Eng.* Cham, Switzerland: Springer, 2017, pp. 132–147.
[7] P. M. S. Bueno and M. Jino, "Automatic test data generation for program paths using genetic algorithms," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 12, no. 6, pp. 691–709, 2002.
[8] M. Bures and B. S. Ahmed, "Employment of multiple algorithms for optimal path-based test selection strategy," 2018, *arXiv:1802.08005*. [Online]. Available: https://arxiv.org/abs/1802.08005
[9] A. Dwarakanath and A. Jankiti, "Minimum number of test paths for prime path and other structural coverage criteria," in *Proc. IFIP Int. Conf. Test. Softw. Syst.* Berlin, Germany: Springer, 2014, pp. 63–79.
[10] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton Univ. Press, 2015.
[11] B. Hoseini and S. Jalili, "Automatic test path generation from sequence diagram using genetic algorithm," in *Proc. 7th Int. Symp. Telecommun.*, Sep. 2014, pp. 106–111.
[12] N. Li, F. Li, and J. Offutt, "Better algorithms to minimize the cost of test paths," in *Proc. IEEE 5th Int. Conf. Softw. Test., Verification Validation (ICST)*, Apr. 2012, pp. 280–289.
[13] J.-C. Lin and P.-L. Yeh, "Automatic test data generation for path testing using GAs," *Inf. Sci.*, vol. 131, nos. 1–4, pp. 47–64, 2001.
[14] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[15] B. A. Nejmeh, "NPATH: A measure of execution path complexity and its applications," *Commun. ACM*, vol. 31, no. 2, pp. 188–200, 1988.
[16] S. C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 5, pp. 520–529, Sep. 1979.
[17] F. Sayyari and S. Emadi, "Automated generation of software testing path based on ant colony," in *Proc. Int. Congr. Technol., Commun. Knowl. (ICTCK)*, Nov. 2015, pp. 435–440.
[18] P. R. Srivastava, N. Jose, S. Barade, and D. Ghosh, "Optimized test sequence generation from usage models using ant colony optimization," *Int. J. Softw. Eng. Appl.*, vol. 2, no. 2, pp. 14–28, 2010.
[19] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

EBRAHIM FAZLI received the bachelor's and master's degrees, in 2003 and 2007, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Engineering, University of Zanjan. His research interests include software testing, design of multicore/multithreaded programs, formal methods, and dependable and high assurance systems.

MOHSEN AFSHARCHI received the M.Sc. degree in computer science from the Iran University of Science and Technology, in 1996, and the Ph.D. degree in artificial intelligence from the University of Calgary, Canada, in 2006. Since 2006, he has been with the Computer Engineering Department, University of Zanjan, Iran, where he leads the Multi-Agent Systems Lab (MASLab). He is currently an Associate Professor with the Computer Engineering Department, University of Zanjan. His research interests include multi-agent learning, probabilistic reasoning, distributed constraint optimization, and software testing.