A Force-Directed Program Graph Visualizer for Teaching White-Box Testing Techniques

Dr. Weifeng Xu, Gannon University

Dr. Weifeng Xu is an associate professor in department of computer science of Bowie State University. He received his B.S. and M.S. degrees in Computer Science from Southeast Missouri State University and Towson University at Maryland, respectively. He also received his Ph.D. in Software Engineering from North Dakota Statement University. His current research efforts focus on search-based software engineering, mining software engineering data, and software testing. He is a senior member of the IEEE.

Mr. Aqeel Raza Syed, Gannon University

Aqeel Raza Syed obtained his master's degree from the department of Computer and Information Sciences, Gannon University in 2013. He is currently working as a software engineer at AT&T labs.

Dr. QING ZHENG, Gannon University

Qing Zheng received the M. Eng. degree from the National University of Singapore in 2003 and the D. Eng. degree from the Cleveland State University in 2009, both in electrical engineering. She is currently an assistant professor in the Electrical and Computer Engineering Department at Gannon University. Her research interests include active disturbance rejection control, multivariable control and optimization with emphasis on their applications in biomedical systems, power systems, MEMS, fuel cell, cargo ship steering, chemical processes, and other real industrial problems.

A Force-Directed Program Graph Visualizer for Teaching White-Box Testing

Abstract

White-box testing is a critical validation technique commonly used to examine if a unit under the test works as expected. However, students taking software testing related courses often find that studying and practicing white-box testing technique is tedious and error prone, e.g., manually derived paths for a given coverage. In this paper, we demonstrate an automated testing visualization tool to help software engineering students study white-box testing technique. First, we use Java bytecode to represent the Java method source code so that a compound condition can be decomposed to several simple predicates. The control structure of the bytecode is depicted by a program graph. To visualize the program graph and generated paths for a given coverage, we extend a force-directed graph auto-layout algorithm to compute the positions of vertices and edges of the program graph. The visualization tool is used to demonstrate the control structures of several legacy programs. The empirical study (includes three subjects) results show that the average number missing and incorrect test cases drops from 16% to 5.8% when the tool is utilized for generating test cases.

1. Introduction

Software engineering, as an emerging discipline, has been distinguished from computer science. There are 22 ABET [1] and 101 Department of Education's recognized institutes in U.S [2] offering a bachelor degree program in software engineering. In addition to the traditional needs of preparing graduates to analyze, design and implement systems, both organizations indicate that software engineering curriculum must prepare graduates to validate software systems.

White-box testing is a critical validation technique commonly used by software testers to examine if their unit code works as expected. White-box testing requires testers generating test paths for a given test criterion. However, students taking software testing related courses often find that learning and practicing white-box testing is tedious and error prone due to the nature of the testing process. The testing process for white-box testing usually consists of three key activities: (1) convert source code to a program graph, (2) generate paths for a given test criterion, e.g., decision coverage, and (3) verify the correctness of generated paths and the total number of paths generated. It is hard for students to derive and verify correct paths from unit under testing manually based on a variety of test coverages.

Developing such an automated testing tool is a challenging task. Two critical issues need to be taken into consideration: (1) how to construct a program graph and handle compound conditions automatically. Compound condition consists of multiple simple conditions, e.g., `(a>b) && (b>c)` contains two simple predicates `a>b` and `b>c`. Compound conditions need to be recognized automatically so that different paths can be derived from a program graph based on decision, condition, and multi-condition coverages; (2) how to visualize program graphs. Visualizing

program graphs is essentially a graph auto-layout problem, which arranges the positions of each vertex and edge of a graph automatically. However, existing graph auto-layout algorithms do not address the problem of positioning source and sink vertices of a program graph. These special vertices often are positioned on top and at the bottom of the graph, respectively.

To address these issues, we use Java byte code to represent the Java source code so that a compound condition can be decomposed to several simple predicates. To arrange the positions of vertices and edges in a given program graph, including source and sink vertices, we have extended a force-directed graph auto-layout algorithm by introducing an additional force, i.e., a gravity force. We successfully implement a teaching tool based on our approach. We demonstrate our tool using several legacy programs, including Triangle program, Commission program, Zodiac program etc.

The rest of this paper is organized as follows: Section 2 introduces basic definitions with a running example. Section 3 demonstrates the program graph visualizer. Section 4 shows learning outcomes assessment. Section 5 concludes the paper.

2. Program Graph

2.1 A Running Example

We use the classical Zodiac problem [3] as a running example to illustrate how the teaching tool has been developed to visualize white-box testing to enhance students' learning experience. Zodiac problem finds the zodiac sign for a given date, e.g., *Invalid (Type 0), Aries (Type 1), Taurus (Type 2), Gemini (Type 3)* and *Cancer (Type 4)*. Note that only five types of zodiac signs are calculated for the purpose of demonstration. The source code of Zodiac program is shown below.

```
int zodiacSign(int month, int day, int year) {
     int result = 0;
     if (month == 3 && day >= 21 || month == 4 && day <= 19) {
          result = 1;
     } else if (month == 4 || month == 5 && day <= 20) {
          result = 2;
     } else if (month == 5 || month == 6 && day <= 20) {
          result = 3;
     } else {
          result = 4;
     }
     return result;
}
```

All predicates of `if` statements in Zodiac problem are compound conditions. A compound condition can be decomposed into single predicates in Java bytecode, which will be addressed in Section 3.3. Java bytecode is the instruction set of the Java virtual machine. Java source code needs to be compiled into bytecode in order to be executable. Java bytecode is a stack-oriented language, which pops data (operand) from the top of the stack and pushes data back on the top of

the stack. The stack is commonly referred to as an operand stack [4]. For example, the bytecode instruction `if_icmpne n` pops the top two integers off the stack and compares them. If the two integers are not equal, execution branches to the instruction line *n*. The two integer values are pre-loaded from a *local variable table* using `iconst` or `iload` instructions. To facilitate the discussion, the bytecode of Zodiac instructions is shown below.

Table I. *Zodiac* Byte code Instructions

----------Block 1	**----------Block 10**	**----------Block 20**
[1] iconst_0	[17] goto 44	[32] if_icmpeq 39
[2] istore 4	**----------Block 11**	**----------Block 21**
[3] iload 1	[18] iload 1	[33] iload 1
[4] iconst_3	[19] iconst_4	[34] bipush 6
----------Block 2	**----------Block 12**	**----------Block 22**
[5] if_icmpne 9	[20] if_icmpeq 27	[35] if_icmpne 42
----------Block 3	**----------Block 13**	**----------Block 23**
[6] iload 2	[21] iload 1	[36] iload 2
[7] bipush 21	[22] iconst_5	[37] bipush 20
----------Block 4	**----------Block 14**	**----------Block 24**
[8] if_icmpge 15	[23] if_icmpne 30	[38] if_icmpgt 42
----------Block 5	**----------Block 15**	**----------Block 25**
[9] iload 1	[24] iload 2	[39] iconst_3
[10] iconst_4	[25] bipush 20	[40] istore 4
----------Block 6	**----------Block 16**	**----------Block 26**
[11] if_icmpne 18	[26] if_icmpgt 30	[41] goto 44
----------Block 7	**----------Block 17**	**----------Block 27**
[12] iload 2	[27] iconst_2	[42] iconst_4
[13] bipush 19	[28] istore 4	[43] istore 4
----------Block 8	**----------Block 18**	**----------Block 28**
[14] if_icmpgt 18	[29] goto 44	[44] iload 4
----------Block 9	**----------Block 19**	[45] ireturn
[15] iconst_1	[30] iload 1	
[16] istore 4	[31] iconst_5	

Note that the operand of `iload` points to the index of the local variable table, according to JVM specification [4]. The local variable table contains bytecode instruction input parameters after initialization of method invocations. For example, three input variables `month`, `day`, and `year` of Zodiac program are stored in the first three spots of the local variable table when the method is invoked. During the execution, `iload 1`, `iload 2`, and `iload 3` push values at the indices `1, 2,` and `3` of the local variable table to an operand stack, respectively.

2.2 Definition of Bytecode Program Graph

White-box testing relies on the analysis of program graph. A program graph for Java bytecode is a directed graph in which nodes are bytecode statement instructions and edges represent flow of bytecode control instructions. Bytecode statement instructions include (1) load and store (e.g. `aload`, `istore`), (2) Arithmetic and logic (e.g. `iadd`, `fcmpl`), (3) Type conversion (e.g.

i2b, d2i), (3) Object creation and manipulation (new, putfield), and (4) Operand stack management (e.g. swap, dup2). Bytecode control instructions include (1) Control transfer (e.g. ifeq, goto) and (2) Method invocation and return (e.g. invokespecial, ireturn). To construct a bytecode program graph for a given method, the instructions of a method need to be divided into program blocks to where the control flow may change. A block marker is the first instruction in these blocks.

(Definition 1: Block Markers) A Java method M consists of a sequence of bytecode instructions represented by numbered IDs from 1 to n. Block markers of M is represented by L ={*l*| *l* is the start of an instruction to where the control flow may change, *l* < n}. Block markers are often organized in an ascending order. Formally, a block *l* is defined as:

$$l = \begin{cases} i \text{ and } i+1 & \text{if } i \text{ is a control instruction } \boldsymbol{if}XXX, \boldsymbol{invoke}XXX \quad (a) \\ s & \text{if } i \text{ is a control instruction } \boldsymbol{if}XXX \text{ } s \text{ or } \boldsymbol{goto} \text{ } s \quad (b) \\ i & \text{if } i = 1 \quad (c) \\ i-1 & \text{if } i \text{ is a control instruction } X\boldsymbol{return} \quad (d) \end{cases}$$

Where ifXXX and invokeXXX represent instructions start with if and invoke. They are referred to as if and invoke statements in the rest of paper, respectively. Note that formulas (c) and (d) the starting and end markers. For example, for a given instruction [5] (i.e., if_icmpne 9), the next discovered block markers are {5, 6, 9} based on formula (a) and (b). Zodiac program bytecode instructions are divided into 28 segments shown in Table I based on the definition of block markers. The dashed lines are the visual representing of block markers used for dividing instructions into instruction segments. The IDs of instructions under the dashed lines are block markers.

(Definition 2: Bytecode Program Graph) A bytecode program graph of a Java method M is a 5-tuple G(V, E, s, t, e), where G′(V, E) is a simple digraph. The vertex set V = Vs ∪ Vc and Vs and Vc represent statement and control blocks (i.e., blocks contain statements or control statements) in M, respectively. The edge set E represents the flow of controls between statement and control blocks in M, i.e., E ⊆ {Vs →Vc ∪ Vc $\xrightarrow{d}$Vs} where d is a predicate decision with either True or false value. s is a start vertex represents the entry point of M and t is a termination vertex represents the exit point of M. e contains one edge e1=s →V and a set of edges e2 ⊆ {v →t}. It indicates that a program only has one incoming edge and may have a set of e2 if it has multiple return statements.

2.3 Construct Bytecode Program Graphs

Constructing a program graph G(V, E, s, t, e) mainly is to (1) identify vertices V and (2) identify edges E for a given method M.

Step 1: Identify vertices. There exists a one-to-one mapping relation between vertices and program blocks. The type of the vertices is based on if the marker is a control statement in a program block. For example, blocks 2 and 18 are control vertices because their marker are either if

or goto control statements. Note that control statements, i.e., if and goto, are associated with destination markers l ∈ L to where the control should be redirected. To determine the destination in terms of block, a mapping function is needed to find a block for the given marker.

Step 2: Identify edges. The execution flow of method M is a sequential order of all the blocks start with markers L unless a block is a control block. If a control block is an unconditional control block, such as goto l, it redirects the execution flow to the new block starting with l. In case the control block contains a condition control block, e.g., if l statement, it redirects the execution flow to the new block starting with l and next separating marker in L as well. The latter indicates that the conditional statement can be evaluated either as true or false. The algorithm is described as follows.

```
Algorithm: Identifying Edges E
Inputs: L: a set of markers
Outputs: E: a set of edges

Procedure
     E ←{∅}
     For each l ∈ L do
          v1 ← f(l)
          If l is a control statement with a new target l'∈ L
               v2 ← f(l`)
               E ←  E ∪ e (v1, v2)
          End if
          If l is a conditional control block and l is the last
          marker
               l' ← next l
               v2 ← f(l')
               E ←  E ∪ e (v1, v2)
          End if
     End for
     return E
End procedure
```

To construct a complete program graph, we need to (1) add the source and sink vertices s, t to a program graph, (2) add an edge (s, f(l)) from the source vertex to the vertex containing the first marker, and add edges e from return statements to the sink. The function f(l) determines v for a given marker l ∈ L. The edges are defined as e = {(f(l), t) | where l is a return statement}. The program graph of Zodiac program is shown in Figure 1. Vertices 0 and 29 are the source and sink. e= {(0, 1) (28, 29)}, which represent the edges connected to source and sink vertices, respectively.

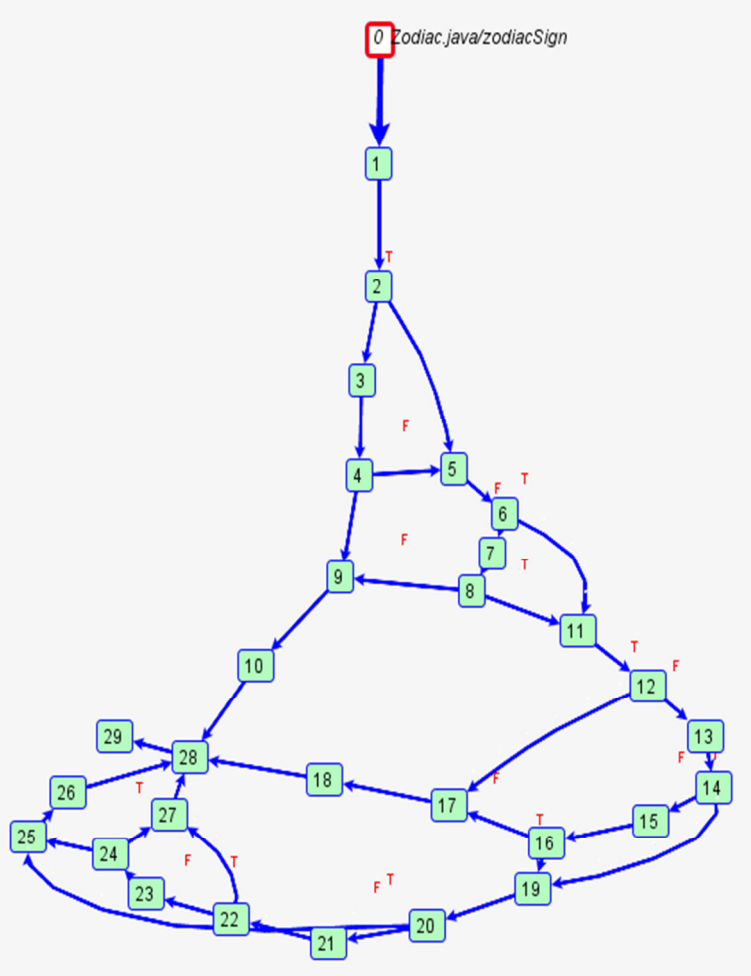


Figure 1. The Program Graph of Zodiac Bytecode

3. A Program Graph Visualizer

A Program Graph Visualizer (PGV) is responsible for determining the layout of a program graph automatically. A PGV needs to address three issues: 1) how to calculate the layout of a program graph as a normal graph, 2) how to position source and sink vertices, and 3) how to determine loops and jumps.

3.1 Visualizing A Program Graph as A Normal Graph

Force-directed algorithms are the most flexible and popular algorithms for calculating layouts of simple undirected graphs. These algorithms calculate the layout of a graph using only information contained within the structure of the graph itself. For a given directed graph `G = {V, E}`, these force-directed algorithms model edges as springs and vertices as charged particles. Springs represent attractive forces based on Hooke's law, which are used to attract pairs of connected vertices towards to each other. Charged particles represent repulsive forces based on Coulomb's law, which are used to separate all pairs of vertices. Force is represented in a vector, which includes magnitude and direction. Force-directed algorithm starts with assigning a random position for each vertex. Then, each vertex calculates attractive and repulsive forces applied to

them and moves to a new position. The calculating and moving activities repeat until the graph reaches equilibrium states. In equilibrium states for a given graph, edges tend to have uniform length because of the spring forces, and nodes that are not connected by an edge tend to be drawn further apart because of the electrical repulsion. Figure 2 shows the automated layout calculation with two forces applied to the program graph of Zodiac problem. Vertices in Figure 2 (a) are assigned with random positions. Figure 2 (b) shows the equilibrium states of the program graph. The algorithm based on Eades' idea [5] is described as follows:

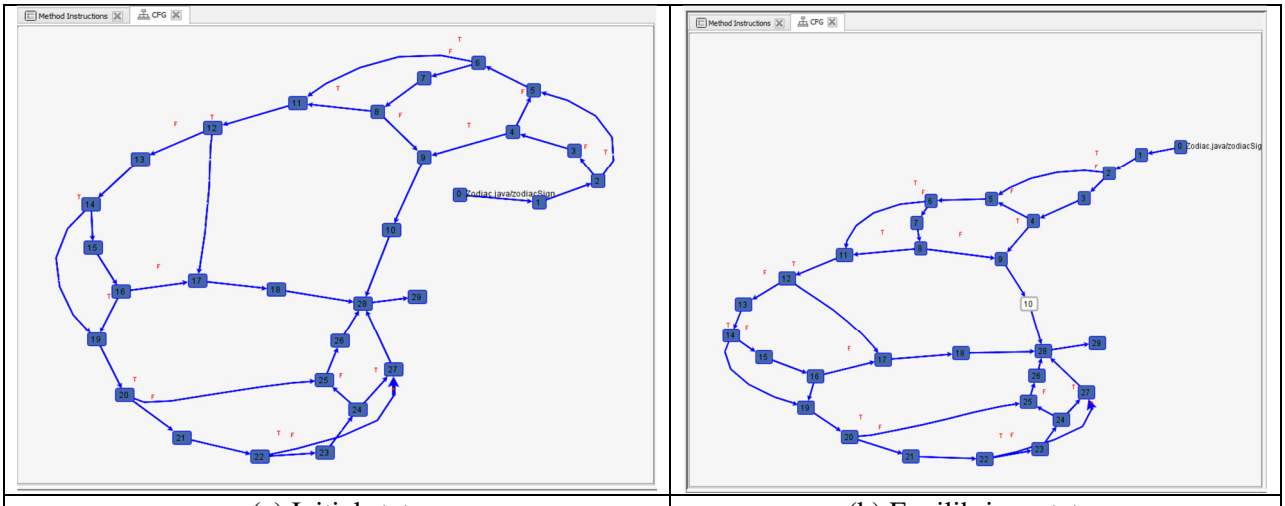
```
Algorithm SPRING
Input: G = graph
Output: new location of each v

Procedure
  Place vertices of G in random locations
  Repeat M times
  Calculate the force F(v) on each vertex
  Move the vertex based on force on vertex
  Draw graph on screen
End of Procedure
```

The force $\vec{F}(v)$ is defined as:

$$\vec{F}(v) = \sum_{(u,v)\in V\times V} \vec{H}_{uv} + \sum_{(u,v)\in E} \vec{C}_{uv} \quad (1)$$

Where $\vec{H}_{uv}$ represents the attractive force between two connected vertices, u and v, calculated based on Hooke's law and $\vec{C}_{uv}$ represents the repulsive force among any vertices calculated based on Coulomb's law.



(a) Initial state | (b) Equilibrium states

Figure 2. The Layout of *Zodica* Program Graph with Two Forces

3.2 Positioning Source and Sink Vertices

The bytecode program graph `G = {V, E, s, t, e}` has two additional vertices, source and sink vertices, referred as `s` and `t`, respectively. A source vertex is a vertex with in-degree of zero, while a sink vertex is a vertex with out-degree of zero. The program graph of an empty function, i.e., a function without any statements consists of `V = {s, t}` and `E = {(s, t)}`.

Unlike the layout solution shown in Figure 2, traditionally, all vertices of a program graph are arranged in the form of top-to-bottom where `s` and `t` are placed on the top and bottom positions, respectively. In order to rearrange `s` and `t` in Figure 2, the third force, named *Earth Gravitational Force*, is added to formula (1). The gravity of Earth, denoted as $\vec{T}$, refers to the acceleration that the Earth impacts to objects on or near its surface.

The *Earth Gravitational Force* is defined as:

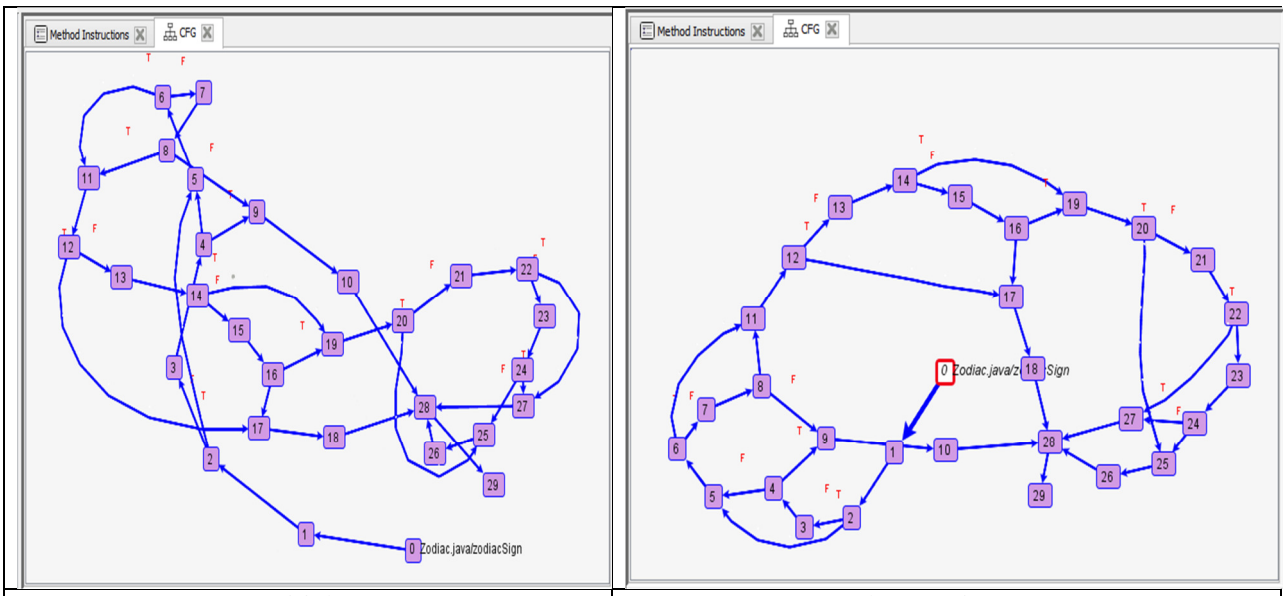
$$\vec{T}(v) = mg \tag{2}$$

Where `m` is the mass of the vertex and `g` is the gravitational content.

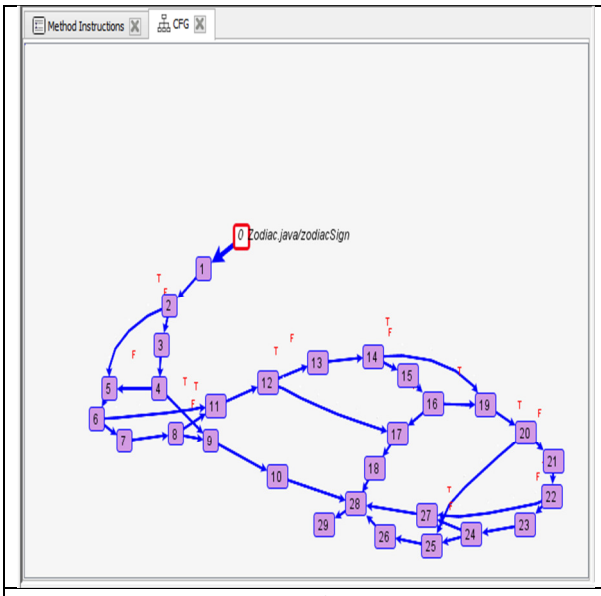
The extended formula for handling source and sink vertices is defined as:

$$\vec{F}(v) = \sum_{(u,v)\in V\times V} \vec{H}_{uv} + \sum_{(u,v)\in E} \vec{C}_{uv} + \sum_{(u)\in E} \vec{T}_u \tag{3}$$

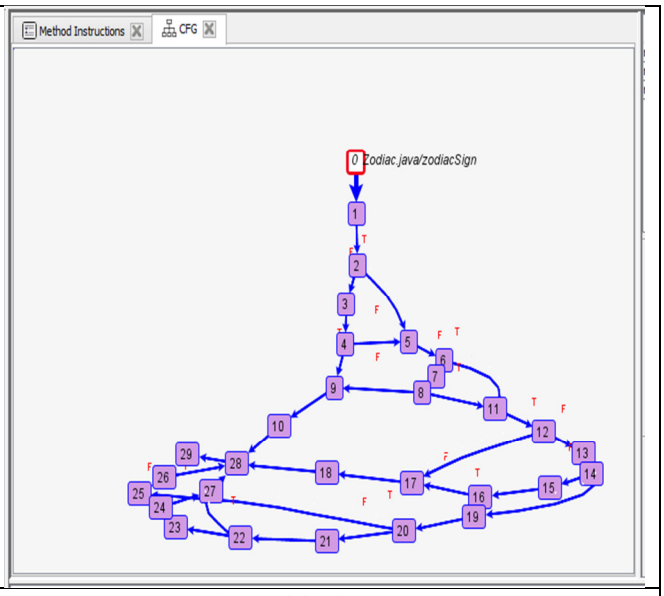
Figure 3 shows the automatically calculated Zodiac Program layout with the additional Earth gravitational force. Figure 3 (a) (b) (c) (d) illustrates the program graph layouts at different iterations.



(a) Initial state | (b) Iteration 10



(c) Iteration 20



(d) Equilibrium states

Figure 3. The Layout of Zodiac Program with Earth Gravitational Force

4. Learning Outcomes Assessment

The objectives of the learning outcomes assessment aim at answering the following two questions:

1. How and to what extent can the visualization tool help students improve their test cases? i.e., we are interested in investigating if students can improve their test sets to improve the condition coverage.
2. What are the student feedback regarding the tool?

Three programs are chosen to evaluate our approach and answer the two questions. These programs include the `Triangle` program, the `NextDate` program [3], and the `Vending Machine` program [11]. The `NextDate` program simply computes the next date for a given date. The `Vending Machine` program is a classical job-interview question for software testing positions. The size of three programs (in Java and Jimple) and the number of predicates are listed in Table III. The three programs cover different number of Jimple predicates, which can be categorized into 3 groups: small (8), median (19), and large (41).

TABLE II. Program Size

	Line of Code		The number of Predicates	
	Java	Jimple	Java	Jimple
Triangle	22	27	3	8
Next Date	48	51	9	19
Vending Machine	82	68	21	41

To answer the first question, we have collected the number of missing and incorrect test cases (#MI) from two groups of students who took software testing courses in fall 2013 and fall 2014,

respectively. Student Group 1, i.e., students took the course in fall 2013, has 20 students, and student Group 2, i.e., students took the course in fall 2014, has 28 students. Students in Group 1 were asked to write test cases manually to exercise condition coverage, including the simple condition and multiple condition coverage. We have only collected #MI data without using the visualization tool since we did not have the tool at that time. Students in Group 2 had two tasks: besides the same task assigned to Group 1's, they then utilize the visualization tool to help them generate test cases again. For Group 2, we have collected # MI using and without using the tool. We have calculated the average #MI = the total #MI found in each subject from all students' submissions / the total number of students. Table IV shows the average #MI for each subject. The letters S and M represent simple condition and multiple condition coverage, respectively.

TABLE III. The Average Number of Missing/Incorrect (#MI) Test Case

	Triangle		Next Date		Vending Machine		Sub-Total		Total
	S	M	S	M	S	M	S	M	
Group 1 (without using tool)	8	8	12	14	21	30	13.7	17.3	15.5
Group 2 (without using tool)	7	9	13	17	18	33	12.7	19.7	16.2
Group 2 (using tool)	1	2	6	7	7	12	4.6	7	5.8
Sub-Total	5.3	6.3	10.3	12.7	15.3	25	10.3	14.7	
Total	5.8		11.5		20.1				

The empirical study results show:

1. Without using the tool, the average #MI for both groups is very similar (15.5% vs. 16.2%). However, the average #MI drops from 16% to 5.8% when the tool is utilized for generating test cases. This is what we have expected.
2. The average #MI for multi-condition is 40% (i.e., 10.3% vs 14.7%) higher than the simple condition coverage. This is because the multi-condition coverage subsumes the simple condition coverage. In other words, the test set satisfying multi-condition coverage with respect to a unit under the test also satisfies simple condition coverage with respect to the same unit.
3. The average #MI for each subject is 5.8%, 11.5%, and 20.1%, respectively. This is positively associated with the number of predicates showing in Table III.

We are also interested in collecting feedback from group 2 after using the tool. The feedback is positive. All students like the tool and they indicate the tool:

1. Saves time. It has saved students at least half of the time to generate test cases as the program graph provides visual aid for test case generation.
2. Improves the correctness of test cases. Students have more confidence when they practice white–box testing.
3. Increases students' involvement. Students are interested in test automation and willing to keep working on the project for extension.

5. Conclusion

This paper presents a novel approach to build a program graph visualizer for teaching white-box testing techniques. The program graph is constructed from bytecode to handle compound condi-

tion for a variety of testing coverage. The graph program visualizer is an extension of force-based layout algorithm, which adds a gravity force. ASM [12] is utilized for building bytecode program graphs. The empirical study results show that the average number missing and incorrect test cases drops from 16% to 5.8% when the tool is utilized for generate test cases. The demo video is available at https://www.youtube.com/watch?v=Ey4JfVhhHQg. The source code can be accessed at https://github.com/Gannon-University/GannonJVM.

References

[1] "Accredited Programs," ABET, 1 10 2014. [Online]. Available: http://main.abet.org/aps/Accreditedprogramsearch.aspx. [Accessed 1 10 2014].

[2] "U.S. Department of Education's List of Recognized Institutions Offering a Degree Program in Computer Software Engineering," [Online]. Available: http://nces.ed.gov/collegenavigator/?s=all&p=14.0903. [Accessed 1 10 2014].

[3] P. C. Jorgensen, Software Testing: A Craftman's Approach, 3rd ed., Auerbach Publications, 2008.

[4] T. Lindholm, F. Yellin, G. Bracha and A. Buckley, Java Virtual Machine Specification, Java SE 7 Edition, Boston, USA: Addison-Wesley Professional, 2013.

[5] P. Eades, "A heuristic for Graph Drawing," *Congressus Numerantium,* vol. 160, no. 42, p. 149, 1984.

[6] J. Zhao, "Analyzing Control Flow in Java Bytecode," in *16th Conference of Japan Society for Software Science and Technology*, Japan, 1999.

[7] H. S. Sinha and M. J. Harrold, "Analysis and Testing of Programs with Exception Handling," *IEEE Transction on Software Engineering,* no. 26, pp. 849-871, 2000.

[8] A. Amighi, P. d. C. Gomes, D. urov and M. Huisman, "Sound Control-flow Graph Extraction for java Programs with Exceptions," in *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, 2012.

[9] K. Beck and E. Gamma, Contributing to eclipse: Principles, Patterns, and Plug-Ins., Addison-Wesley, 2003.

[10] I. G. Tollis , G. D. Battista , P. Eades and R. Tamassia , Graph Drawing: Algorithms for the Visualization of Graphs, Prentice Hall, 1998.

[11] "How Would You Test a Vending Machine?," 1 12 2011. [Online]. Available: http://www.softwaretestingquestions.net/category/testing-in-the-wild/. [Accessed 24 1 2013].

[12] O. Consortium, "ASM," [Online]. Available: http://asm.ow2.org/. [Accessed 23 08 2013].