Presented at the 1999 International Conference on Testing Computer Software

Graph Theory Techniques in Model-Based Testing

Harry Robinson
Semantic Platforms Test Group
Microsoft Corporation

Abstract

Models are a method of representing software behavior. Graph theory is an area of mathematics that can help us use this model information to test applications in many different ways. This paper describes several graph theory techniques, where they came from, and how they can be used to improve software testing.

What's Wrong with Traditional Software Testing?

Traditional software testing consists of the tester studying the software system and then writing and executing individual test scenarios that exercise the system. These scenarios are individually crafted and then can be executed either manually or by some form of capture/playback test tool.

This method of creating and running tests faces at least two large challenges:

First, these traditional tests will suffer badly from the "pesticide paradox" (Beizer, 1990) in which tests become less and less useful at catching bugs, because the bugs they were intended to catch have been caught and fixed.

Second, handcrafted test scenarios are static and difficult to change, but the software under test is dynamically evolving as functions are added and changed. When new features change the appearance and behavior of the existing software, the tests must be modified to fit. If it is difficult to update the tests, it will be hard to justify the test maintenance costs.

Model-based testing alleviates these challenges by generating tests from explicit descriptions of the application. It is easier, therefore, to generate and maintain useful, flexible tests.

What's Modeling?

Modeling is a way of representing the behavior of a system. Models are simpler than the system they describe, and they help us understand and predict the system's behavior.

A common type of model in computing is the state graph, or finite state machine. State graphs are a useful way to think about software behavior and testing (Beizer 1995).
The application begins in some state (such as “main window displayed”), the user applies an input (“invoke help dialog”) and the software moves into a new state (“help dialog displayed”).

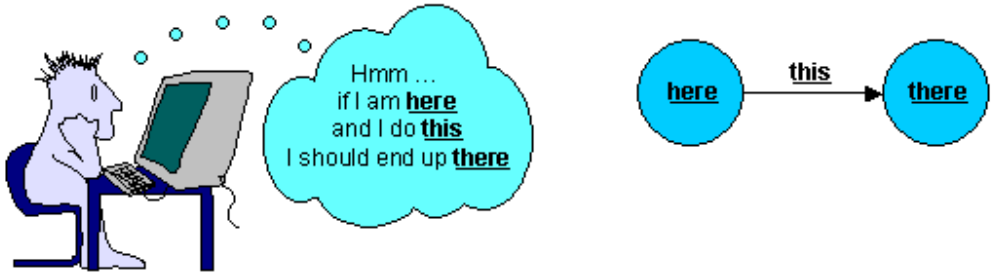

Figure 1: A user and his model

We use models all the time to understand behavior, as shown in Figure 1. In fact, much software testing can be viewed as the tester “moving through” the various states of an application under test and verifying that each step worked correctly.

What’s Model-Based Testing?

In recent years, there has been a growing movement in software testing to use the information contained in explicit models of software behavior to make it simpler and cheaper to do testing. (Beizer 1995, Apfelbaum 1997)

Model-based testing is a black-box technique that offers many advantages over traditional testing:

- Constructing the behavioral models can begin early in the development cycle.
- Modeling exposes ambiguities in the specification and design of the software.
- The model embodies behavioral information that can be re-used in future testing, even when the specifications change.
- The model is easier to update than a suite of individual tests.

And, most importantly for the purpose of this paper, a model furnishes information that can be coupled with graph theory techniques to generate many different test scenarios automatically.

What’s Graph Theory?

Graph theory has nothing to do with graph paper or x- and y-axes. Graph theory is an area of mathematics that deals with entities (called **nodes**) and the connections (called **links**) between the nodes. For instance, in Figure 1 above, the circles inscribed with “**here**” and “**there**” are nodes; the line labeled “**this**” is a type of link.

A Quick Tour through Graph Theory

Graph theory began in the Prussian town of Königsberg in 1736. The town was built on both sides of the Pregel River and on two islands in the middle of the river. On lazy Sunday afternoons, the happy citizens of Königsberg would stroll across the seven bridges that joined the different parts of the town.

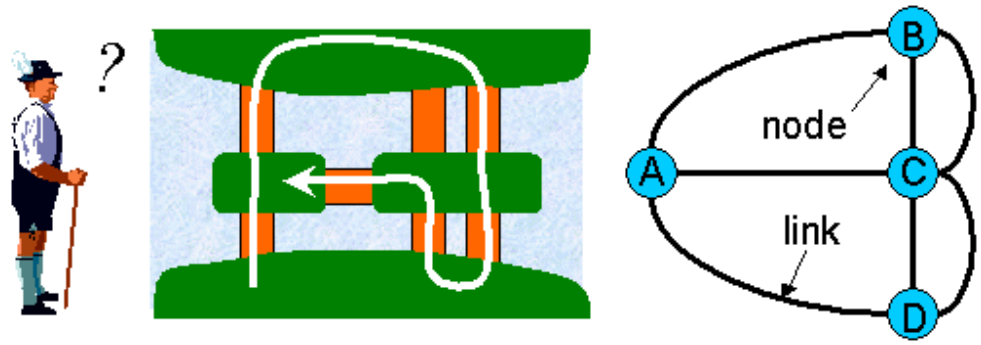

Figure 2: From Bridges to Graphs

A common pastime during these Sunday strolls was to try to walk across each of the bridges exactly once, finally returning to one's starting place. After several years of trying without success, someone thought to ask mathematician Leonhard Euler (pronounced "oiler") if he could figure out if such a walk was even possible.

Euler realized that he could model the Königsberg problem with nodes (labeled A, B, C and D above) representing the landmasses and links connecting the nodes wherever a bridge connected two landmasses. Once the problem was distilled down to this essence, Euler could see that the proposed walk was impossible. The key factor was the number of links attached to each of the nodes.

For a walker to cross every bridge once and return home there must be an even number of links touching each node. Having an even number of links attached to each node means that every time a walker enters a node there will be an available link out of that node. In the case of the Königsberg bridges, each of the four nodes has an odd number of links, so there is no way that a stroller could complete the tour without crossing some bridges multiple times.

Since that time, a graph that contains only nodes with even numbers of links (which would permit the walker to complete the desired stroll) has been called an "Euler graph" and the traversal of the various links is called an "Euler tour". (Gross and Yellen 1998)

And the citizens of Königsberg evidently had to find some other way to occupy their Sunday afternoons.

So what does all this have to do with the Cost of Delivering Mail in China?

In 1962, a mere 226 years after the Königsberg incident, a mathematician named Kwan Mei-Ko asked a sensible follow-up question to the bridge-crossing problem. (Kwan 1962)

Given that it is impossible to cross each bridge exactly once and return to the starting point, what is the minimal amount of re-crossing a walker needs to do?

This is the type of problem a postman faces when delivering mail. Suppose that each of the links in the graph in Figure 3 is a street where mail must be delivered, and each of the nodes is an intersection where the postman can change direction. (Note that the node icons in Figure 3 display the number of links touching the node.) The postman must travel across each link to deliver the mail, but would like to minimize the amount of additional walking (called “deadheading”) that is needed.

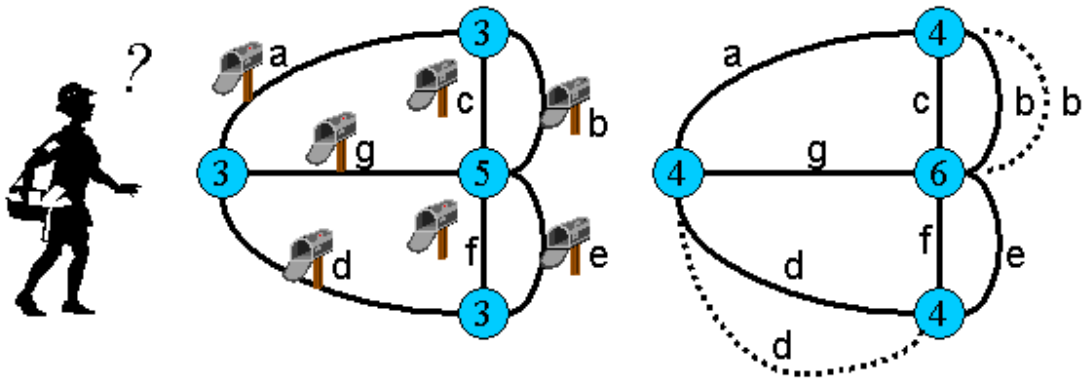

Figure 3: A Postman’s Problem and a Postman’s Solution

Kwan’s solution was to treat re-crossing of any link as if it were adding duplicate links to the graph. These “duplicated” links could provide each node in the graph with an even number of attached links. This process is called “Eulerizing” the graph.

Once the graph was Eulerized, one could perform an Euler tour of the resulting modified graph. For example, the postman in Figure 3 could take the following minimal route to deliver his mail:

a b c b e f g d d

repeating only the “b” and “d” segments of his route.

In Kwan’s honor, this problem became known as the “**Chinese Postman Problem**”.

From China to the Streets of New York

Several years later, a useful variation on the Chinese Postman Problem asked for the minimal length tour when the links only allow passage in one direction. Since this is the problem faced by street sweepers in the one-way streets of New York, it became known as the **New York Street Sweeper Problem**. (Beltrami 1977, Bodin & Tucker 1983)

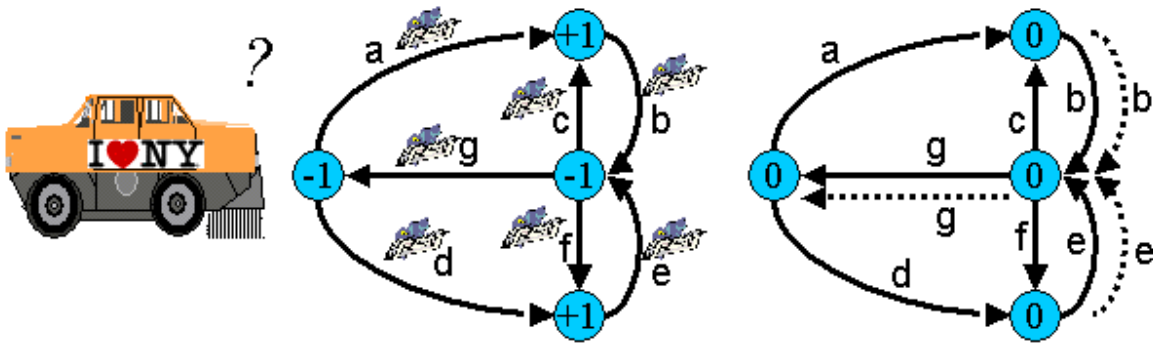

Figure 4: A Street Sweeper's Problem and a Street Sweeper's Solution

[Note: Some additional terminology is helpful here: links that can only be crossed in one direction are called **arcs**. And a graph consisting only of arcs is called a "directed graph" or **digraph** for short.]

In constructing an Euler tour in the Chinese Postman Problem, we were concerned with the number of links touching each node. In the case of a ***digraph***, an Euler tour must also take account of the ***direction*** of the arcs touching each node. To "Eulerize" a digraph, it is necessary have equal numbers of arcs coming into the node and arcs leaving the node.

So, counting every incoming arc as "+1" and every outgoing arc as "-1", we can calculate the "polarity" of each node in the digraph.

For example, a node with two outgoing arcs and one incoming arc would have a polarity of " 1 – 2 = -1".

(Note that the node icons in Figure 4 display each node's polarity before and after Eulerization.)

To create an Eulerian digraph, we duplicate arcs between positive polarity nodes and negative polarity nodes so that each node ends up with zero polarity. This balancing of incoming and outgoing arcs achieves the same purpose as having even numbers of links in the case of the undirected graph: anyone entering a node via an incoming arc is guaranteed to be able to exit the node on an outgoing arc.

In Figure 4, we duplicated arcs "b", "e" and "g" to balance the digraph. The street sweeper in Figure 4 can now use the following minimal directed route:

a b c b f e g d e g

repeating only the "b", "e" and "g" segments of his route.

So what does all this have to do with the Cost of Delivering Software in Redmond?

A Testing Problem

Suppose you are a tester and you have a behavioral model of some software you would like to test. And suppose further that the model looks like the left hand digraph in Figure 5, where the nodes represent the states of the application and the arcs are the actions you can perform.

One of the first approaches you might think of is to execute every possible action. But how can you do it in an efficient manner? A quick look at the application's state graph shows you that this is just the New York Street Sweeper Problem in a thin disguise!

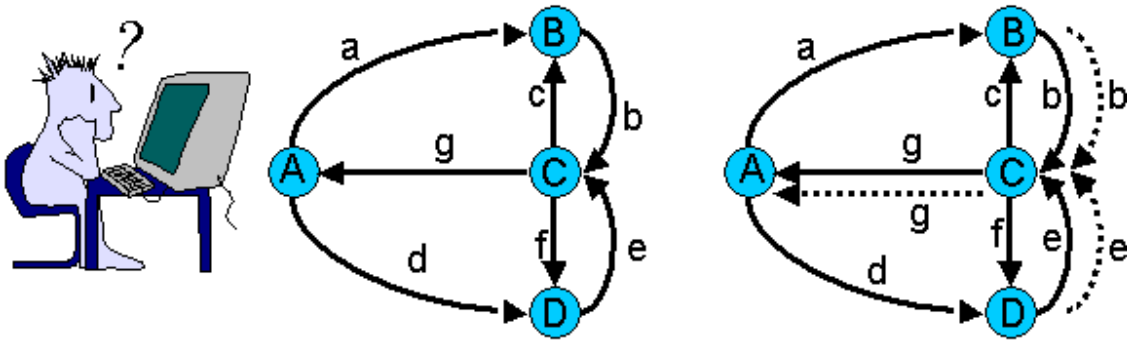

Figure 5: A Tester's Problem and a Tester's Solution

Since you already have the behavioral model available to you as a graph, you can apply the appropriate graph algorithm to generate an efficient graph traversal automatically. In the example in Figure 5, our tester could test all available actions as follows:

a b c b f e g d e g

Testing Combinations of Actions

Executing every action in the model is nice; it's like achieving statement coverage when testing code. But since one of the defining characteristics of models is that there are typically a large number of possible "next actions" at every node in the graph, what can we do if we would like to test these combinations of actions? For instance, suppose the tester in Figure 5 wants to make sure to test *link combinations*
such as "b c", "b f", and "b g"? A quick look at the sequences we generated shows us that "b g" appears nowhere in our test sequence. This is because the Chinese Postman and its variations guarantee visiting every link, but not every combination of links.

The approach of testing combinations of actions is called "switch cover" in finite state machine testing (Chow 1978). And there is a simple graph theory algorithm known as de Bruijn (pronounced "de-broyn") sequences that generate the appropriate actions to get switch cover. (Gross and Yellen 1998)

In keeping with graph theory's colorful nomenclature, de Bruijn sequences are also known as "safecracker sequences" because they describe "the shortest sequence of dial turns … sufficient to try out all combinations of length ***n***." (Skiena 1998)

The de Bruijn sequence for combinations of length 2 (all pairs of adjacent actions) is generated as follows:

1. Create a **dual** graph of the original graph (i.e., a graph in which the arcs of the original graph are converted to nodes)
2. Everywhere in the original graph that arc 1 is an incoming to a node and arc 2 is outgoing from the same node, create an arc in the dual graph from node 1 to node 2. For instance, in the left hand graph in Figure 6, arc "a" is incoming to the node from which arc "b" is outgoing; therefore in the dual graph (on the right hand side of Figure 6), there is an arc from node "a" to node "b".
3. Figure 6 shows the completed dual graph.
4. Eulerize the dual graph (by duplicating arcs to balance node polarities)
5. Traverse the Eulerized graph, noting the names of the nodes that you pass.

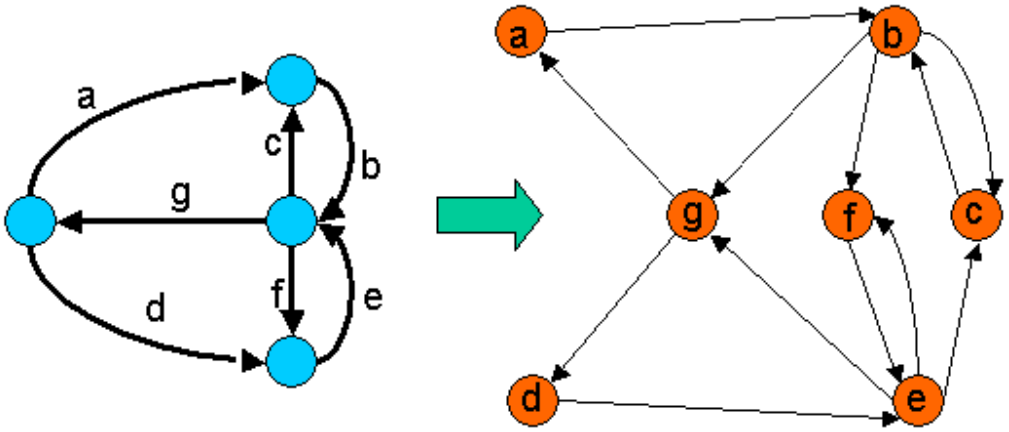

Figure 6: A Model and its Dual Graph

The generated sequence will be the length 2 switch cover for the graph. (You can see that all 2-link combinations, including "b c", "b f" and "b g" are generated.)

a b c b f e c b g d e f e g

Testing under a Time Deadline

Suppose you are faced with the following situation when testing the software modeled in Figure 5:

- You have several test machines available.
- You have very limited time for testing
- Each action in the behavioral model takes an hour to execute (purely for the sake of this discussion)
- The application goes to the reset state when it receives input "g".
- Each test machine can run a test scenario from the reset state.

Look again at the Street Sweeper test sequence given for Figure 5:

a b c b f e g d e g

If we divide these actions between two machines at the reset states (i.e., after executing action “g”), we get the following distribution:

Machine 1: a b c b f e g (running time 7 hours)
Machine 2: d e g (running time 3 hours)

This is an inefficient use of test machines. It would be nice if we could distribute the workload more evenly between the machines. And shorter test sequences would make it easier to reproduce any bugs encountered.

Well, imagine we had a platoon of street sweepers ready to roll. How could we distribute the work among them evenly? There is no easy, elegant, optimal solution to this “limited sub-tour” problem; but there are several very good heuristic approaches.

One approach (Dill, Ho, Horowitz &Yang 1995) sets an upper limit on the number of arcs that can be visited in a single sub-tour. The algorithm searches for unvisited arcs in the graph, visiting as many as possible without exceeding the specified upper limit. Once the number of arcs in the sub-tour approaches the limit, the algorithm ends that sub-tour, proceeds to the reset state and begins generating another sub-tour. The algorithm makes no guarantee of optimality, but the tours are usually efficient. If we limit each sub-tour in Figure 5 to five or six arcs, we might end up with the following distribution, saving 2 hours in execution time:

Machine 1: a b c b g (running time 5 hours)
Machine 2: d e f e g (running time 5 hours)

Random Walks

A random walk (sometimes called a “drunkard’s walk”) is simple to describe: from the current node, choose an outgoing link at random, follow that link to the next node and repeat the process.

Random walks are very simple to implement because they have no real guiding, overall plan. Interestingly, they can be very useful in software testing because their very lack of a plan makes them fairly resistant to the pesticide paradox. Random walks have been used with great success in some of Microsoft’s testing efforts. (Nyman 1998)

On the disadvantage side, random walks tend to be very inefficient about covering a large graph quickly. Since they have no notion of where they have already been

in the graph, they tend to re-traverse links they have already visited. For instance, a random walk on a typical application might invoke the Help screen many times before moving on to testing the parts of the application that you want it to test.

Testing the Most Likely Paths

It would be helpful if there were a way to guide the random walk into areas that are of more interest to the tester. For instance, you might want to have the random walk biased toward the activities that a user is more likely to perform. A technique known as Markov chains can help "guide" the randomness of the walk by assigning probabilities to the links leaving a node so that a random walk is statistically more likely to follow the higher probability links. There is, in fact, a growing body of work along these lines. (Whittaker & Thomason 1992)

One disadvantage to statistical Markov chain approaches, however, is that they are still at the mercy of the dice. Suppose you had the model and probabilities shown in Figure 7. For the sake of this example, action **a** has probability 0.99999 of being invoked from node **A**; action **b** has probability 0.00001 of being invoked. If we choose randomly between them based on these probabilities, we might execute **a** thousands of times before we ever execute **b**. This is inefficient.

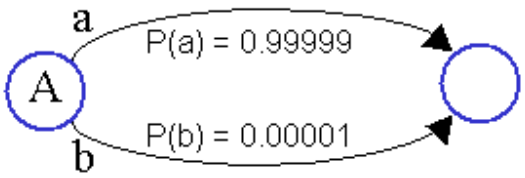

Figure 7: A Markov Chain approach

Markov Chains meet Graph Theory

It would be useful if we could combine the notion of Markov chain probabilities and graph traversal techniques to produce an efficient traversal of the paths that interest us.

Here's how to do it:

1. Assign probabilities to the links in the graph (as with regular Markov chains)
2. Establish a minimum probability for the paths that you are interested in, such as 0.0000001
3. Starting at the initial node, move through the graph, keeping track of the cumulative probability of the links you've traversed so far.
4. Continue traversing links in the graph until you reach the final node of the graph or until the cumulative probability of the path falls below the minimum you've established.
5. If you have reached the final node, record the path that got you there as well as its probability.
6. If you have merely fallen below the minimum probability, back up and try a different link.
7. Once you have collected all the paths in the graph that have cumulative probability greater than your minimum, sort them in order of their cumulative probability, from highest to lowest.

By testing the highest probability paths first, you will be assured of exercising and testing the application in the most efficient way to assure the reliability of the software.

Conclusion

Models are an excellent way to represent and understand system behavior, and they provide an easy way to update tests to keep pace with applications that are constantly changing and evolving.

Testing an application can be viewed as traversing a path through the graph of the model. Graph theory techniques therefore allow us to use the behavioral information stored in models to generate new and useful tests.

Because graph theory techniques deal directly with the model,

- New traversals can be automatically generated when the model changes
- Tests can be constantly changing on the same model
- Different types of traversals can meet different needs of testers
- The traversal techniques are general and can be re-used on different models

References

Apfelbaum, L. (1997) "Model-Based Testing", **Proceedings of Software Quality Week 1997**

Beizer, B. (1990) **Software Testing Techniques,** 2nd Edition

Beizer, B. (1995) **Black Box Testing**

Beltrami, E. (1977) **Models for Public Systems Analysis**

Bodin, L. and Tucker, A. (1983) "A Model for Municipal Street Sweeping Operations" in
Modules in Applied Mathematics Vol. 3: Discrete and System Models

Chow, T.S. (1978) "Testing Software Design Modeled by Finite-State Machines", **IEEE Transactions on Software Engineering 4**

Dahbura, A. and Uyar, M. (1986) "Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931", **IEEE GLOBECOM,** Dec. 1986

Dill, D., Ho, R., Horowitz, M. and Yang, C. (1995) "Architecture Validation for Processors", **Proceedings of the 22nd annual International Symposium on Computer Architecture**

Gross, J. and Yellen, J. (1998) **Graph Theory and its Applications**

Kwan, M-K. (1962) "Graphic Programming Using Odd and Even Points", **Chinese Journal of Mathematics**, Vol. 1

Nyman, N. (1998) "GUI Application Testing with Dumb Monkeys", **Proceedings of STAR West 1998**

Shen, Y. and Lombardi, F. (1992) “Graph Algorithms for Conformance Testing Using the Rural Chinese Postman Tour”, **SIAM Journal on Discrete Mathematics**, Vol. 9

Skiena, S. (1998) **The Algorithm Design Manual**

Whittaker, J. and Thomason, M. (1992) “A Markov Chain Model for Statistical Software Testing”, **IEEE Transactions on Software Engineering,** Vol. 20