**<Testing the usefulness of concept of Hamiltonian Path in Software Testing**

**and Test Case Generation>**

Sargam Badyal, Ankita Bihani, Pooja Waikar

School of Computer Science and Education,VIT University, Vellore - 632014, TamilNadu, India

[sera.bravo004@gmail.com](mailto:sera.bravo004@gmail.com), 8870698010

**Abstract**

Software testing is the process of evaluating the capability or any functionality of the system and comparing it with its desired performance. Testing is not a simple debugging process. It spans across a wide array of activities that include quality assurance, verification, validation and estimation of its reliability, scalability and maintainability. Software testing is a trade-off between budget, time and quality. But broadly the software testing can be classified as Unit testing, Integration testing, Validation testing and System testing. By the inculcation of the concept of Hamiltonian paths we can improve greatly on the facet of software testing of any project. This paper shows how Hamiltonian paths can be used for requirement specification phase and acceptance testing phase for checking if all the user requirements are met or not. Further it gives calculations and algorithms to show the feasibility of its implementation.

**Keywords:** Hamiltonian, nodes, walk, requirements, software testing, test case, edge weights, intermediate vertices, terminal vertices, user requirement phase, specification and analysis phase, project planning

**Literature review**

The two basic terminologies involved with Hamiltonian graphs are- Hamiltonian circuits and Hamiltonian paths. Hamiltonian circuit of a graph is defined as a closed walk of an undirected graph which includes all its vertices. The path that is left after the removal of an edge from a Hamiltonian circuit is known as Hamiltonian path (no vertex repeated). And, any graph that contains a Hamiltonian path is called a Hamiltonian graph. The basic concept of Hamiltonian graph revolves around covering each node of the graph exactly once. This is what forms the base of this paper. In Fig 1 the circles are the nodes that are marked by the numerals 1,2,3,4 and 5. The line connecting the circles is the edge and the red line shows the Hamiltonian path for the given undirected graph.

**Problem Statement**

In the current scenario, as the complexity of the software being developed is increasing, software testing is becoming more and more challenging. In cases where no proper software testing is involved, a huge amount of money is spent to compensate the errors occurring. Effective development of a project requires choosing the most appropriate testing techniques suiting the project requirements. Application of genetic algorithms for software testing seems to be a solution but suffers from major drawbacks like it has too many test cases, is an extremely lengthy process and is expensive .Software testing process can become much easier if the right combination of the various testing techniques is employed. A good test case is a test case whose chances off finding a bug are more. The challenge is to simplify the process of software test data generation and make it more efficient.

**Methodology & Solutions**

**Initialization**

Before we start off with the explanation of how the Hamiltonian graphs help in making software requirement validation and testing easy, we must discuss about some of the notations and assumptions that we have made for this paper. For the easy understanding and interpretation of how the graph concepts are related to software engineering we will be representing the requirements in the form of a graph. The various components of the graphs are vertices (nodes), edges and weights of edges. Since in the concept of Hamiltonian graphs the original graph is undirected that means that means there is no particular sequence for moving from one node to the other. Thus for making the Hamiltonian graph we don’t have to have a starting or ending edge. But while dealing with software engineering, we have to make an assumption and include extra parameters for beginning and ending requirement.

The requirements are represented as nodes or vertices and the edges specify the flow or dependency of one requirement on the other. The weights of the edges represent the total cost (computational resources/time) of implementing the front requirements from the rear one. We must note that there is more than one way to implement a requirement going from different set of requirements since the requirements can exist in graph which is not circuit less. Before plotting the graph we must decide which node is the starting and which is the stopping node. We can fix vertices as start and stop by initiating the algorithm with start and end requirements. Or we can just put start phase and end phase into graph which has to be traversed irrespective of other requirements specifies in intermediate nodes. Once the terminal nodes are set we have to device a way to construct the graph and defining dependencies of the requirements and finding the alternate paths for fulfilling the user requirements. This can be done during the user requirement phase & requirement specification/ analysis phases. The weights of the edges can be determined in the project planning phase. The requirement engineer and the user can sit together to plot a graph of the user requirements more effectively taking help from project manager for estimating the cost that forms the weight of edge.

**Algorithms**

Once the initialisation is done now we are ready with a graph of the requirements. Now the following algorithm is run on the graph for finding the optimal Hamiltonian graph for finding the optimised path in the graph previously generated. Since the path is Hamiltonian thus all vertices are covered ie all requirements are fulfilled or implemented only once. With the help of optimal Hamiltonian path we can check for the optimal path for fulfilling requirements. Thus the path can be followed for best and optimal use of resources.

In testing environment: other thing we can do is once the implementation of the requirements is done. We can mark the graph (original) with the ‘path’ followed during the implementation process. If the path is not Hamiltonian ie not covering all the requirements that means it has a loophole.

The basic algorithm goes like:

Int main()

{

Function testing()

{

Check\_hamiltonian() //checks for Hamiltonian path in the initialise graph

While(requirement \_miss()==true || node\_unreached==true)// if one or more requirements are not implemented

{

Display\_unreached \_nodes/requiremnts ( );//show the unfulfilled requirements

Implement\_requirements( );// call for implementing the requirement

Check\_hamiltonian();

exit();// exit the loop

}

}

Function get\_optimised\_path( )

{

Intitialise\_graph(start,stop)

{

Set node 1 as start;

Input other nodes;

Set stop node;

Set\_weights();// sets the weights of edges, project planning phase provides this information

}

Make\_hamiltonian(graph);

Display\_hamiltonian(start,stop)

{

Display start;

Display intermediate nodes;

Display stop;

}

}

}

The algorithm used to make the Hamiltonian path is specified below:

The algorithm proposed is a brutal-force algorithm:

Make\_hamiltonian()

{

Start()// choose a starting vertex

Label:

Travel\_to\_unvisited\_vertex();

If(weight==smalest)

{ Mark\_visited();

Goto label1;

}

Else

{ goto label; }

Label1: If(all\_vertices\_covered==true)

{ break;}

Else{ goto label;}

}

Chuz\_optimal\_hamil()

{

List\_all\_graphs();

Calc\_weight\_of\_each();

Pick\_min\_weight\_graph();

}

**Result**

These are the set of requirements specified by the user for “hospital management”(lets say)

1. Login
2. Home
3. Get appointment
4. View confirmed appointment
5. Refer Specialist
6. Upload medical reports
7. Apply for medical leave
8. Get information about disease
9. Download personal health info

R4

R3

R2

R1

R5

R6

1. Sign out

Flow graph of above requirements

Finding paths from start to stop:

1-2-3-4-5-6-5-7-10

1-2-3-4-5-6-7-10

1-2-3-4-5-6-9-10

1-2-3-4-5-7-10

1-2-3-4-6-5-7-10

1-2-3-4-6-5-6-7-10

1-2-3-4-6-5-6-9-10

1-2-3-4-6-7-10

1-2-3-4-6-9-10

1-2-8-10

Hamiltonian path solution:1-2-3-4-5-7-6-9-10-8

R4

R3

R2

R1

R5

R6

As you can see in the above diagram representing the graph of requirements we can say that there are:

No of vertices (n): 10

No of edges (e): 14

Lets apply the formula for cyclomatic complexity is: f= e-n+2;

14-10+2= 6 namely r1,r2,r3,r4,r5,r6

Let us see how we got the formula

In the above mentioned method we have only applies the concept of cyclomatic complexity to the user requirements directly for finding the complexity of the requirements and finding the flow of executing the requirements. Since this paper is more central to the idea of use of concept of graphs in the software testing phase that the requirement phase, thus we will discuss the testing and use of Hamiltonian path in software testing.

* In the unit testing phase we have to test the code implemented by two methods namely black box testing and white box testing. In the black box testing we have to simply give the input and obtain an output from an implemented code whose structure is not visible to the tester. Here since we don’t know the internal structure and functions that are to be covered in testing, the Hamiltonian concept can’t be applied. But in case of white box testing, Hamiltonian paths can plan an essential role in finding a path. This path is drawn by the developer who develops the code, obviously when a developer is writing a code he will be making one code at a time, and before starting off with the coding he will have to be clear with all the requirements and functions he has to call during writing the code. Once the requirements for code is ready the cost of implementing each requirement is checked and according to how the developer codes the requirements and submits artifacts we can tract his process of development and while by drawing the Hamiltonian graph we can check what requirements are met and what are not. The graph below represents the implementation flow of one function to another, the red and black line combined is the initialised graph and the red line represents the implemental flow of developer. The oval represents the unfinished requirement.

Func1( )

Func2( )

Func3( )

Func4( )

Func5( )

Func6( )

* In the integration testing phase we can apply the concept to two different graphs that can be represented by a single connected line, such graphs will be separable graphs or one connected graphs. The below diagram represent such graphs.

Func1( )

Func2( )

Func3( )

Func4( )

Func5( )

Func6( )

Func1( )

Func2( )

Func3( )

Func4( )

Func5( )

Func6( )

* Similarly in validation and system testing, we can plot the graphs for the user requirements and system requirements and find the Hamiltonian path for finding the flow and checking if the requirements are met or not.

**Conclusions**

Process can be extended to construction on more graphs for the testing platform requirements, compatibility test, hardware support test, standardisation test, reliability test, portability test, quality assurance, scalability and maintainability). With the help of this paper we can highlight that the use of Hamiltonian graphs and paths can be of great use in software testing phase. Not only the Hamiltonian concept helps us to find the missing requirements but it also helps us in finding the optimal Hamiltonian path for the implementation of various requirements whether it be user requirements or during coding requirements. Also we have come up with the proof of the Cyclomatic complexity for the flow graphs formed during the process.

This paper can form the basis for the future work in the actual implementation of Hamiltonian graphs in software engineering projects and by experimentally finding the usefulness of the concept in making the software testing and acceptance testing easy for both the developer and the user.

**References**

Software product line testing – A systematic mapping study by Emelie Engström , Per Runeson, September 2010

Application of Genetic Algorithm in Software Testing by Praveen Ranjan Srivastava1 and Tai-hoon Kim2, October 2009