Optimizing Software Testing and Test Case Generation by using the concept of Hamiltonian Paths

Ankita Bihani\textsuperscript{*1}, Sargam Badyal*2
\textsuperscript{*}VIT University
Vellore, India

Abstract—Software testing is a trade-off between budget, time and quality. Broadly, software testing can be classified as Unit testing, Integration testing, Validation testing and System testing. By including 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. It can also be used in acceptance testing phase for checking if all the user requirements are met or not. Further it gives the necessary calculations and algorithms to show the feasibility of its implementation.

Keywords—Hamiltonian, software testing, test case, edge weights, terminal vertices, user requirements

I. INTRODUCTION

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 correct the errors post-development phase [5]. Software testing is the process of evaluating the capability or functionality of the system and comparing it with its desired performance or outcome. 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. 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 too many test cases are involved. Besides, it is an extremely lengthy and expensive process [2]. Software testing process can become much easier if the right combination of the various testing techniques is employed [3]. A good test case is a test case whose chances of finding a bug are more. The challenge is to simplify the process of software test data generation and make it more efficient.

II. BACKGROUND

The two basic terminologies involved with Hamiltonian graphs are- Hamiltonian circuits and Hamiltonian paths. Hamiltonian circuit [4] 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). Any graph that contains a Hamiltonian path is called a Hamiltonian graph [4] [7] [8] [9] [10]. 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.

![Fig.1. Graph with Hamiltonian path](image)

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 are the edges and the red line shows the Hamiltonian path for the given undirected graph.

III. METHODOLOGY

A. 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 in this paper. For 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, it means that there is no particular sequence for moving from one node to the other. Thus for constructing the Hamiltonian graph we do not need to have a starting or an ending edge. But while dealing with software engineering, we have to make an assumption and include extra parameters for beginning and ending requirement handling since there is a predetermined sequence to be followed.

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 latter requirements from the former ones. 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 constructing the graph we must decide which node is the starting node and which is the ending node. We can fix the vertices as starting and ending by initiating the algorithm with start and end requirements. Alternatively, we can put start phase and end phase into graph which has to be traversed irrespective of other requirements specified in intermediate nodes. Once the terminal nodes are set we have to devise a way to construct the graph, define dependencies of the requirements and find 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 when we frame our Bill Of Materials (BOM) for the system specifying the resources required and accordingly resulting cost is computed. The requirement engineer and the user can sit together to construct a graph of the user requirements more effectively taking help from the project manager for estimating the cost that forms the weight of the edge.

B. Algorithm used

Once the initialization is done, we are ready with a graph of the requirements. Now, the following algorithm is run on the graph for finding the optimal Hamiltonian path in the graph previously generated. Since the path is Hamiltonian all vertices are covered i.e. all requirements are fulfilled or implemented exactly once. With the help of optimal Hamiltonian path, we can find the optimal path for fulfilling requirements. This can thus be followed for judicious and optimal use of available resources.

In testing environment, another thing we can do once the implementation of the requirements is done is that we can mark the original graph with the ‘path’ followed during the implementation process. If the path is not Hamiltonian i.e. does not cover all the requirements it means that the process followed so far has a loophole. Then, we go for backtracking.

The basic algorithm is as follows:

1. Declare a function Function_testing () that calls the function check_hamiltonian().

2. Do the following till a requirement is missed or node is unreach ed, i.e. one or more requirements are not implemented.

   2.1 Display the unfulfilled requirements

2.2 Implement Requirements

2.3 Call check_hamiltonian() function that checks whether the graph is Hamiltonian or not.

2.4 Exit the loop

3. In the function get_optimized_path():

   3.1 Declare the function initialize_graph(start,stop) where start and stop are the initial and final nodes respectively. In the initialize_graph() function, do the following:

      3.1.1 Set node 1 as start

      3.1.2 Input the other nodes

      3.1.3 Set the weights of edges (the Project Planning phase will provide this information).

   3.2 Call the function make_hamiltonian(graph).

   3.3 Declare the function display_hamiltonian()

      3.3.1 Display start

      3.3.2 Display intermediate nodes

      3.3.3 Display stop

The algorithm proposed to make the Hamiltonian path is a brute force algorithm and is specified below:

1. The function make_hamiltonian() is as under:

   1.1 Choose a starting vertex.

   1.2 Travel to unvisited vertices from a visited vertex.

      1.3 If the weight is equal to smallest, mark it as visited and go to step 1.5

      1.4 Else go to step 1.2

   1.5 If all vertices are covered, come out of the loop, else go to step 1.2

2. In the function choose_optimal_hamiltonian(), do the following:

   2.1 List all Hamiltonian paths.

   2.2 Calculate weight of each
2.3 Pick minimum weight graph

**Example scenario:**
Let us say that these are the set of requirements specified by the user for “Hospital Management System”

![Graph showing the requirements specified by the user for “Hospital Management System”](image)

- Login
- Home
- Get appointment
- View confirmed appointment
- Refer Specialist
- Upload medical reports
- Apply for medical leave
- Get information about disease
- Download personal health info
- Sign out

Flow graph of above requirements
Finding paths from start to stop:
1. 1-2-3-4-5-6-7-10
2. 1-2-3-4-5-6-7-10
3. 1-2-3-4-5-6-9-10
4. 1-2-3-4-5-7-10
5. 1-2-3-4-6-5-7-10
6. 1-2-3-4-6-5-6-7-10
7. 1-2-3-4-6-5-9-10
8. 1-2-3-4-6-7-10
9. 1-2-3-4-6-9-10
10. 1-2-3-4-6-9-10

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

Let us apply the formula for cyclomatic complexity is: \( f = e - n + 2 \)

Here \( n = 10 \), \( e = 14 \). Thus,

\( f = 14 - 10 + 2 \)

Hence, \( f = 6 \) namely \( r_1, r_2, r_3, r_4, r_5, r_6 \)

---

**IV. RESULTS AND DISCUSSIONS**

A connected planar graph with \( n \) vertices has \( e \) edges has cyclomatic complexity \( f \) has cyclomatic complexity \( 6, 7, 8, 9, 10 = e-n+2 \).

**A. Proof:**

We assume that the graph is a simple graph, i.e., it does not have self-loops and/or parallel edges. This is because adding a self-loop or parallel edge will increase the value of cyclomatic complexity by 1 and simultaneously increase \( e \) by 1. Thus the effect is negated.

Similarly, we also disregard (or remove) the edges that do not form boundaries. Addition (or removal) of any such edge increases (or decreases) \( e \) by one and increases (or decreases) \( n \) by one. Hence \( e-n \) is unaltered.

Since any simple planar graph can have a plane representation such that each edge is a straight line, any planar graph can be drawn such that each region is a polygon (or a polygon net). Let the polygon net representing the given graph have cyclomatic complexity \( f \) and let \( k_r \) be the number of \( p \)-sided regions. Since each edge is on the boundary of exactly two regions,

\[ 3 \cdot k_3 + 4 \cdot k_4 + 5 \cdot k_5 + \ldots \ldots + r \cdot k_r = 2 \cdot e \]

Where \( k_r \) is the number of polygons with maximum edges

Also, the sum of all angles subtended at each vertex in the polygon net is \( 2\pi \).
The sum of all interior angles of a p-sided polygon is \( \pi (p-2) \) and the sum of all exterior angles is \( \pi (p+2) \).

The sum of all the interior angles of all the regions plus the sum of all the exterior angles of the polygon is equal to \( 2\pi n \).

Thus
\[
\pi (3-2)k_1 + \pi (4-2)k_4 + \ldots + \pi (r-2)k_r + 4\pi = \\
\pi (2e-2f) + 4\pi \quad \text{(1)}
\]

Equating the above two equations we get,
\[
2\pi (e-f) + 4\pi = 2\pi n \quad \text{(2)}
\]
\[
e-f+2=m \quad \text{(3)}
\]

Therefore the cyclomatic complexity \([6] [7] [8] [9] [10]\) is:
\[
f=e-n+2 \quad \text{(4)}
\]

**B. Discussion**

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

- In the unit testing \([1] [11]\) 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. In Black box testing, 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 play an essential role in finding a path. This path is drawn by the developer who develops the code. It is obvious that 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 that he has to call while writing the code. Once the requirements for the code are ready, the cost of implementing each requirement is checked and according to how the developer codes the requirements and submits the artefacts we can track his process of development. Simultaneously, by constructing a Hamiltonian graph, we can check whether all the requirements are met or not. The graph below in Fig.4 represents the implementation flow of one function to another. The red and black arrows combined show the initialised graph. The red arrows represent the implementation flow of the developer. The oval represents the unfinished requirement.

\[\text{Fig.4. Graph representing the flow of execution of the functions}\]

- In the integration testing \([1] [11]\) 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 diagram below represents such graphs.

- Similarly in validation and system testing\([1] [11]\), we can construct 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.

**V. CONCLUSION**

The algorithm discussed in this paper can be extended to construct more complex 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 have highlighted that the use of Hamiltonian graphs and paths can be of great use in software testing phase. The algorithm presented in this paper helps us in finding not only the missing requirements but also in finding the optimal Hamiltonian path for the implementation of various requirements whether it is user requirements or coding requirements.

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

**Acknowledgements**

We convey our sincere gratitude to Prof Senthil J for providing us with the platform and opportunity to work in this area. His encouragement and valuable guidance throughout the course of the paper was a constant support. Also we thank the VIT library management & staff for helping us utilise the library resources, namely –journals and books available, effectively.
VI. References


