USING GENETIC ALGORITHMS FOR TEST CASE GENERATION AND SELECTION OPTIMIZATION

Izzat Alsmadi

Yarmouk University

ABSTRACT

Genetic Algorithms (GAs) are adaptive search techniques that imitate the processes of evolution to solve optimization problems when traditional methods are considered too costly in terms of processing time and output effectiveness. In this research, we will use the concept of genetic algorithms to optimize the generation of test cases from the application user interfaces. This is accomplished through encoding the location of each control in the GUI graph to be uniquely represented and forming the GUI controls' graph. After generating a test case, the binary sequence of its controls is saved to be compared with future sequences. This is implemented to ensure that the algorithm will generate a unique test case or path through the GUI flow graph every time.

Index Terms— Test case generation, genetic algorithms, GUI controls' graph, and test automation.

1. INTRODUCTION

An optimization algorithm tries to find the best feasible solution that conforms to all problem constraints. The algorithm begins with a random process for selecting the chromosome (i.e. the GUI control in our application or the software testing domain) and keeps adapting, adjusting and selecting others to the process.

Artificial Intelligent (AI) algorithms such as GA are used to find the best solution for a particular problem. Testing takes a large portion of the software project resources. Saving in this stage can be a great help for the software development process. Manual testing can be slow and expensive. We can use Artificial Intelligent (AI) algorithms (e.g. genetic algorithms) to generate test cases automatically while ensuring that the generated test cases are not repetitive from each other. This will eventually maximize the test coverage for those generated test cases.

2. RELATED WORK

GA was invented by John Holland by the year 1975 and elaborated in his book "Adaption in Natural and Artificial Systems" [8]. Later, John Koza used GAs in programming in what is called Genetic Programming (GP) to perform certain tasks effectively. They can be used in several different applications and fields. In particular, they are used to solve several types of optimization problems [7].

There are several research projects tried to propose and implement test case generation algorithms that are completely or partially automated. In [1], Planning Assisted Tester for graphical Systems (PATHS) takes test goals from the test designer as inputs and generates sequences of events automatically. These sequences of events or plans become eventually test cases for the GUI. PATHS first performs an automated analysis of the hierarchical structure of the GUI to create hierarchical operators that are then used during the plan generation. The test designer describes the preconditions and effects of these planning operators, which subsequently, become the input to the planner. Each planning operator has two controls that represent a valid event sequence. For example, File_Save, File_SaveAs, Edit_Cut, and Edit_Copy are examples of planning operators. The test designer begins the generation of particular test cases by identifying a task, consisting of initial and goal states. The test designer then codes the initial and goal states or uses a tool that automatically produces the code (that is not developed yet). However, the process to define, in a generic way, the current and the goal states automatically, can be very challenging. This approach relies on an expert to manually generate the initial sequence of GUI events and, then uses genetic algorithm techniques to modify and extend the sequence. The test case generator is largely driven by the choice of tasks given to the planner. In this research, test case generation is fully automated without user intervention.

Jones, et. al. [9, 101 showed that appropriate fitness functions are derived automatically for each branch predicate using genetic algorithms. The tests are derived from both the structure of the software and its formal specification in the Z formal language. All branches were covered with two orders of magnitude fewer test cases than random testing.

Lin et al [11] developed a metric or a fitness function to determine the distance between the exercised path and the target path. The genetic algorithm with the metric is used to generate test cases for executing the target path.

3. GOALS AND APPROACHES

In genetics, humans have cells; cells have chromosomes, which have genes and then blocks of DNA. Chromosomes here represent the population or the set of the solution. Solutions from one population are taken and used to form a new "better population".

This loop is repeated until some feasible condition is satisfied.

In GUI test case generation, GUI controls represent the chromosomes or the population. The challenge is in defining the solution or when to stop the search for a better solution. The challenge also is in the definition of a “good” solution. How can we tell, during test case generation, that this is the best solution?!

The chromosome should in some way contain information about solution which it represents. The most used way of encoding is a binary string. The chromosome then may look like Figure 1.

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Figure1. Chromosome binary representation

For test case generation, we encoded horizontal and vertical level values for each control. The main window in the application user interface is considered level 0 (i.e. top level) as it has no parent. Numbers encoded with the controls represent the control vertical and horizontal location in the tree). A tool is developed to serialize GUI control properties with the control horizontal and vertical values added to those properties [3]. Figure2 shows an example of a GUI XML file generated dynamically from an Application Under Test (AUT) using the developed tool. Note that the encoded control level and control unit are different from the control horizontal location (i.e. locationX), and its vertical location (i.e. locationY) which represents the control location in the form.

```
<GUI-Forms>
<Root>GUI-Forms</Root>
<Open> <Root>Open</Root> <Open>
<Parent-Form>NotepadMain</Parent-Form>
<Name>Open</Name>
<Control-Level>1</Control-Level>
<ControlUnit>0</ControlUnit>
<Text>Open</Text>
<LocationX>66</LocationX>
<LocationY>87</LocationY>
<Forecolor>Color [ControlText]</Forecolor>
<BackColor>Color [Control]</BackColor>
<Enabled>True</Enabled>
<Visible>False</Visible> <Label>
<Parent-Form>Open</Parent-Form>
<Name>OpenFilelabel8</Name>
<Control-Level>1</Control-Level>
<ControlUnit>0</ControlUnit>
<Text>my computer</Text>
<LocationX>12</LocationX>
<LocationY>307</LocationY>
<Forecolor>Color [ControlText]</Forecolor>
<BackColor>Color [Control]</BackColor>
<Enabled>True</Enabled>
<Visible>False</Visible> </Label> <Total>
<Total-FileControls>2</Total-FileControls>
</Total> <Label>
<Parent-Form>Open</Parent-Form>
<Name>OpenFilelabel7</Name>
<Control-Level>1</Control-Level>
<ControlUnit>1</ControlUnit>
<Text>My Documents</Text>
<LocationX>28</LocationX>
<LocationY>259</LocationY>
<Forecolor>Color [ControlText]</Forecolor>
<BackColor>Color [Control]</BackColor>
<Enabled>True</Enabled>
<Visible>False</Visible> </Label> <Total>
<Total-FileControls>3</Total-FileControls>
</Total> <Label>
<Parent-Form>Open</Parent-Form>
<Name>OpenFilelabel6</Name>
<Control-Level>1</Control-Level>
<ControlUnit>2</ControlUnit>
<Text>Desktop</Text>
<LocationX>20</LocationX>
<LocationY>195</LocationY>
<Forecolor>Color [ControlText]</Forecolor>
<BackColor>Color [Control]</BackColor>
<Enabled>True</Enabled>
<Visible>False</Visible> </Label> <Total>
<Total-FileControls>4</Total-FileControls>
<Forecolor>Color [ControlText]</Forecolor>
```

Figure 2. A sample of a dynamically created XML file.

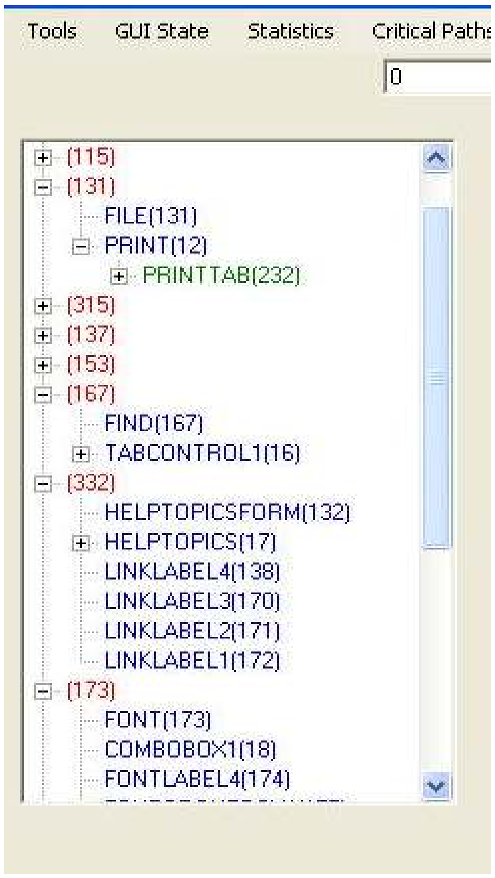
Similar to chromosomes, each control in the GUI graph should be uniquely identified by the combination of its vertical and horizontal locations. This means that there should not exist two controls in the GUI graph that have identical value for the vertical and the horizontal levels.

The decimal value that represents the control vertical and horizontal location is then converted to a binary value (Figure 3). Those values are saved for all graph GUI components.

Controls	Decimal values	Binary values
File	131	10000011
Help Topics	17	10001000
Font	173	10101101
Print	12	11000000
Print Tab	232	11101000

Figure3: Decimal and binary values for the location of GUI controls

Figure3 shows a small part of the GUI graph with the controls’ horizontal and vertical levels encoded and displayed.



```
⊞ (115)
⊟ (131)
    FILE(131)
  ⊟ PRINT(12)
      ⊞ PRINTTAB(232)
⊞ (315)
⊞ (137)
⊞ (153)
⊟ (167)
    FIND(167)
  ⊞ TABCONTROL1(16)
⊟ (332)
    HELPTOPICSFORM(132)
  ⊞ HELPTOPICS(17)
    LINKLABEL4(138)
    LINKLABEL3(170)
    LINKLABEL2(171)
    LINKLABEL1(172)
⊟ (173)
    FONT(173)
    COMBOBOX1(18)
    FONTLABEL4(174)
```

Figure 4. The Notepad GUI tree.

The test case generation optimization algorithm will try always to find new paths for the new test cases. A newly generated test case by the tool or the algorithm is considered “good” if it is never previously generated. The first test case is randomly generated from the GUI graph (for example, a test case can be: Mainform_File_Exit, or Mainform_File_New_writeText_Save, etc.). Each decimal value encoded in the graph for a control will be converted to its binary representation and the whole test case or the GUI controls sequence is saved (in the form of a binary sequence). Each newly generated test case will be compared with the encoded binary sequence to ensure that each test case will represent a uniquely visited path in the GUI graph. This will ensure better test coverage or adequacy in the generated test cases.

We implemented another algorithm to optimize the selection of test cases through selecting representatives. A scenario is randomly selected and uses a same-level reduction technique to reduce the search domain (this can be one example on how to use genetic algorithms for selecting representative test cases).

Through the comparison of components horizontal and vertical values, all controls that share the selected control its level are eliminated (as one representative of them is selected). From testing perspectives, we expect controls that are in the same level to behave similarly.

3.1 Using the optimization theory for optimizing the selection of test cases

In the optimization theory format, the goal of test case generation algorithms in regression testing is to maximize test effectiveness or coverage (ultimately cover all possible paths, executions, decisions, logics, etc) with the following constraints:

1. The number of faults discovered using the selected test suit is maximum.
2. The number of test cases that are in the test suite is minimum.
3. The time it takes to execute those test cases is minimum.
4. The percentage of usage of the selected components is maximum (i.e. operational profiles which are not elaborated in this research).
5. All selected test scenarios are valid and represent actual paths in the application under test.

For example, to demonstrate the first 3 constraints in the optimization model, let’s assume that an application has the test cases described in Table1. The total number of test cases in the suite is 4, the total number of faults to discover is 19, and the time it takes to execute all those test cases is 20. Table2 shows test set1 (TS1) from Table1 as compared to other test sets. TS1 seems to be the best selection given that within 4 test cases, it can discover 19 faults in 20 seconds. In order to be able to compare based on one fitness function, the other possible fitness functions should be fixed. For example, to calculate fitness based on number of faults discovered all generated chromosomes should be given a fixed time and then calculate the number of discovered faults. Calculating fitness functions using the optimization theory is not elaborated in this paper and will be covered and elaborated in a separate experiment and research.

Table 1. Possible test cases in an application set.

Test case	No. of faults discovered	Execution time
T1	3	5
T2	5	8
T3	8	3
T4	1	4

Table 2. Possible test sets for an application.

Test set	No. of test cases	Total No. of faults discovered	Total Time it takes
TS1	4	19	20
TS2	8	22	25
TS3	6	15	20
TS4	12	20	30

4. CONCLUSION AND FUTURE WORK

In this research, we proposed and evaluated a test case generation technique that depends on the principles of genetic algorithms to generate test cases that provide good coverage in terms of the paths it tests or visits within the application. The idea of encoding the location of the controls (in comparison to the chromosomes) and representing them in a binary format, allowed us to test the overall sequence generated by each test case. The goal we selected here is the generation of a "new" test case every time. Other goals can be experimented using the same algorithms. One of the other goals that will be evaluated in the future is the effectiveness of the generated test scenarios. This requires the execution of the test scenario to study its effectiveness. Another goal is to make the fitness function be finding an error. We can keep generating unique test sequences or scenarios until we find errors. This can be the definition of the goal for the test case generation.

5. REFERENCES

[1] Memon, Atef. Hierarchical GUI test case generation using automated planning. IEEE Transactions on Software Engineering. Pages: 144-155. 2001.

[2] Berndt, Donald, J. Fisher, L. Johnson*, J. Pinglikar, and A. Watkins. Breeding software test cases with genetic algorithms. In Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03). Hawaii, USA. Page: 338. 2003.

[3] Alsmadi, I, and Kenneth Magel. "An Object Oriented Framework for User Interface Test Automation". MICS07. 2007.

[4] Geng-Dian Huang, and Farn Wang. Automatic Test Case Generation with Region-Related Coverage Annotations for Real-Time Systems. Springer. 2005.

[5] Alberto Avritzer, and Elaine J. Weyuker. The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. IEEE Transactions on Software Engineering. 1995.

[6] Xun Yuan. Feedback-Directed Model- Based GUI Test Case Generation. Phd dissertation. 2008.

[7] D. Goldberg, "Genetic algorithm in search, optimization, and machine learning". Addison-Wesely, 1989.

[8] Holland, J.H., "Adaptation in natural and artificial systems", The university of Michigan press, 1975.

[9] B.F Jones, H.-H. Sthamer, X. Yang and D.E. Eyres, "The automatic generation of software test data sets using adaptive search techniques", Third International Conference on Software Quality Management", Seville (1 999, pp. 435-444 (BCSICMP).

[10] SI B. F. Jones, D. E. Eyres, H.-H Sthamer, "A strategy for using genetic algorithms to automate branch and fault- based testing," The Computer Joumal, Vol. 41, 1998, pp.98-107

[11] Jin-Cherng Lin and Pu-Lin Yeh, "Using Genetic Algorithms for Test Case Generation in Path Testing", 9th Asian Test Symposium (ATS'00), 2000.