November 2010, Volume 7, No.11 (Serial No.72)
Journal of Communication and Computer, ISSN 1548-7709, USA

Effective Generation of Test Cases Using Genetic Algorithms and Optimization Theory

Izzat Alsmadi, Faisal Alkhateeb, Eslam Al Maghayreh, Samer Samarah and Iyad Abu Doush
Computer Science and Information Technology Faculty, Yarmouk University, Irbid 21163, Jordan

Received: July 22, 2010/ Accepted: September 16, 2010 / Published: November 25, 2010.

Abstract: In software projects, one of the main challenges and sources of success or failure is the effective use of available resources. Using effective techniques in regression testing is important to reduce the amount of required resources. This is accomplished through reducing the number of executed test cases without affecting coverage. In this research, genetic algorithms and optimization theory concepts are applied on test case generation and reduction optimization. The methods start by generating an initial pool of test cases through selecting valid paths in the GUI graph that is generated from the tested software dynamically using an in-house developed tool. The selected test cases are then improved through measuring and evaluating fitness functions. The two fitness functions used in this research were the test set generation speed and the test set coverage. Optimization theory is also used to find the best set, measured according to a particular fitness function that can best represent the whole testing database while preserving all other constraints.

Key words: Test case generation, software testing, software engineering, genetic algorithms, optimization theory, GUI graph, and test automation.

1. Introduction

An optimization algorithm tries to find the best feasible solution that conforms to all problem constraints.

Such algorithm may usually begin with a random process to create an initial population that consists of a number of chromosomes where each chromosome represents a possible solution for the problem being solved. It then follows a process of continuous selection and adjustment based on evaluating the output of the initial population.

Artificial Intelligent (AI) algorithms such as Genetic Algorithms (GAs) are used to find the best solution for a particular problem. They were invented by John Holland in 1975 and elaborated in his book “Adaption in Natural and Artificial Systems” [1]. Later, John Koza used GAs in programming in what is then called Genetic Programming (GP) to perform certain tasks effectively. Since then, they were used in several different applications and fields. In particular, they are used to solve several types of optimization problems [2]. 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.

Testing usually takes a large portion of the software project resources. Cost and time saving in this stage can be a great help for the software development process. Manual testing can be slow and expensive. Artificial Intelligent (AI) algorithms (such as genetic algorithms)

Corresponding author: Izzat Alsmadi, assistant professor, research interests: software, testing, and metrics. E-mail: ialsmadi@yu.edu.jo.
Faisal Alkhateeb: assistant professor, research interests: knowledge-based systems, knowledge representation and reasoning, intelligent systems, constraint satisfaction and optimization problems. E-mail: alkhateebf@yu.edu.jo.
Eslam Al Maghayreh: assistant professor, research interests: runtime verification of distributed programs, natural language processing and information retrieval. E-mail: eslam@yu.edu.jo.
Samer Samarah: assistant professor, research interests: wireless sensor networks, mobile Ad hoc networks, data mining, and software engineering. E-mail: samers@yu.edu.jo.
Iyad Abu Doush: assistant professor, research interests: HCI, web semantic and evaluation, and virtual environments. E-mail: eyad_win@yahoo.com.

can be used then to generate test cases automatically while ensuring that the generated test cases are not redundant. This will eventually maximize the test coverage for those generated test cases.

The remainder of this paper is organized as the following: The next section will present related work in using GA and similar algorithms for test case reduction. Section 3 introduces the methodology and experiments. Section 4 finishes the paper with the conclusion and future work.

2. Related Work

There are several papers that proposed and implemented test case generation algorithms that are completely or partially automated.

Avritzer et al. used Markov model to automatically generate test cases for load testing [3]. Load testing measures system performance and response time under known loads and once loaded are steadily increased. System state failure is defined and then the algorithm is executed to generate a test suite accordingly. Markov chain solver is then used to obtain the transient solution of the Markov chain, for the specified system execution time.

In Ref. [4], 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 test case generation method or planner.

Each planning operator has two controls that should 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. 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 our research, test case generation is fully automated without user intervention. A previous paper for the corresponding author [5] discussed building a test automation framework and proposing several random test generation algorithms that are generated and evaluated automatically.

As a continuation of Memon GUI test automation framework, Yuan worked on the automatic generation of test cases from the GUI [6]. The software runtime information is collected and used as a feedback during GUI test case execution, and used to generate additional test cases.

Goldberg showed the advantage of using genetic algorithms' simple and accumulative power in seeking an optimal solution for a particular problem [2]. In a similar subject, Holland described the advantage of using and simulating some of the human intelligent activities to be used in the programming or design world [1].

Berndt et al evaluated previous experiments of using genetic algorithms in software testing [7]. They summarized several types of possible fitness functions and divided them into: historical, external, absolute and relative fitness functions.

Huang et al. used symbolic techniques to automatically generate test cases, with time and region related coverage annotations [8].

Jones et al. [9, 10] 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. In our GA approach, our focus is on the GUI graph rather than the Control Flow Graph (CFG) followed in this research.

Lin et al. [11] developed a metric or a fitness function (called Similarity) 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. The Similarity algorithm determines the fitness between current executed path and the target path. A greater similarity means a better fitness. The system will automatically generate the next generation of test cases until one of the test cases covers the target path. In a similar goal and approach to this paper, Krishnamoorthi et al. used GAs for test case selections from a regression database [12]. The generated test sets are used to detect seeds faults or mutants in several Java programs. Fitness is measured using method coverage.

In our approach, the coverage that the test sets evaluated is the GUI graph coverage. This can be particularly useful for testing the user interface rather than the structure of the code. The fitness function that they measure which is related to the seeded errors (i.e., error detection fitness) usually depends on the way and the algorithms used to inject those errors which may not simulate actual errors that may exist in real or operational scenarios.

3. Goals and Approaches

In genetics, humans have cells; cells have chromosomes, which have genes and then blocks of DNA. In those biological scenarios, chromosomes are the composite elements of the problem domain. Chromosomes here represent the population or the set of elements to select from the solution. Solutions from one population are taken and used to form a new "better representatives from the population". This loop is repeated until some best feasible solution is reached with all conditions are satisfied.

The ultimate goal of test case prioritization and reduction algorithms is to find the most effective test cases out of a large pool of possible or generated test cases within the shortest possible time. This indicates the two most important parameters as indicators of test case effectiveness (i.e. fitness); the amount of possible faults that a test case may expose and the time it takes for this test case to discover those faults. In reality, "operational faults" are defined to represent the actual faults that the user may be exposed to once we know the amount and the percentage of usage for the system components. For example, a sub system that may contain many faults but is not used very often by the user will have less value of the operational fault relative to a component that is heavily used with less number of errors. As lab experiments cannot accurately predict the operational profiles of the components usage, this part will not be considered in this research.

3.1 Test Case Generation and Selection Optimization Using the Optimization Theory.

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 (syntax, logical, or operational) 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 maximal (i.e., through studying operational profiles).

(5) All selected test scenarios are valid and represent actual paths in the application under test. Fig. 1 shows the summary optimization requirements for the test case generation and selection problem.

The requirements for solving the above optimization problem may require more than a simple linear solution

Max TE (TE: Test Effectiveness),
subject to:
Min T (T: No of Test cases)
Min XT(EXT: Execution Time)
Max OP (OP: Operational Profile)
For Every TC in the suit, for every Edge: E and
Leg: L in the TC, E, and L are in the GUI graph

Fig. 1 Optimization requirements for the test case generation and selection problem.

as there are many goals in the problem (number of faults, generation, execution time, and operational profile issues).

The effectiveness of the generated set of test cases can be measured in different ways. It can be measured based on:

(1) The number of paths visited in the test set relative to the total number of paths in the application (i.e., test coverage or adequacy).

(2) It can be also calculated based on the generation and/or the execution time of the test set.

(3) It can be measured based on the number of faults discovered. This can be divided into two parts: faults in general, and operational faults (which is more dynamic and relevant).

The authors will use the first two as evaluators for the effectiveness of the generated test set as it is very rare for a tester to know the location of all faults prior to testing. Some researchers inject errors in the program and then measure the ability of the generated test set to discover those faults.

In the second step, the optimization algorithm should find the effective coverage in the lease amount possible of the generated test cases. More test cases mean more resources to generate those test cases and more time to execute and verify them. However, it may not be easy to develop an algorithm that can know whether it will come up with the minimum number of test cases. This may also contradict with the time to generate those test cases as it will need more time and resources for such algorithm to find that this was the best solution in terms of the number of generated test cases. As such, a trade off is required to stop the algorithm at an earlier time where number of test cases are continuously added and stopped once coverage reach a steady state.

All visited paths that are generated within the test cases should be valid or actual edges in the GUI graph. This project focus on user interface testing and that's why the input to the test generation tool or algorithm is the GUI graph that is generated from the user interface of the tested application.

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 chro- mosomes 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.

An algorithm is developed that includes the optimization matrix described above with the goal of: maxi- mum testing coverage in terms of the number of GUI graph paths visited. Seven different open source

Table 1 Possible test cases in a set for an application.

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

Table 3 Test case effectiviness using the optimization algorithm.

AUT	Paths number and coverage percentage					
	At 25 test cases	At 50 test cases	At 100 test cases	At 200 test cases	At 300 test cases	At 500 test cases
1	12,100	12,100	12,100	12,100	12,100	12,100
2	15,40	21,65	51,100	51,100	51,100	51,100
3	17,23	26,41	51,75	105,100	105,100	105,100
4	20,74	35,100	35,100	35,100	35,100	35,100
5	8,21	9,23	11,29	11,29	11,29	11,29
6	19,16	38,32	65,55	116,90	156,100	156,100
7	27,07	42,12	71,20	128, 36	174,49	275,77.5

projects written in .NET languages are selected for testing. Those projects vary in terms of their size in general and their number and complexity of the user interface forms in particular. Table 3 and Fig. 2 show the results of applying the optimization algorithm on those projects.

Increasing the number of test cases shows improvements on most tested projects except one (number 5 in table 3). Depending on the size of the project, some small applications do not need more than a small number of test cases to cover all its possible paths.

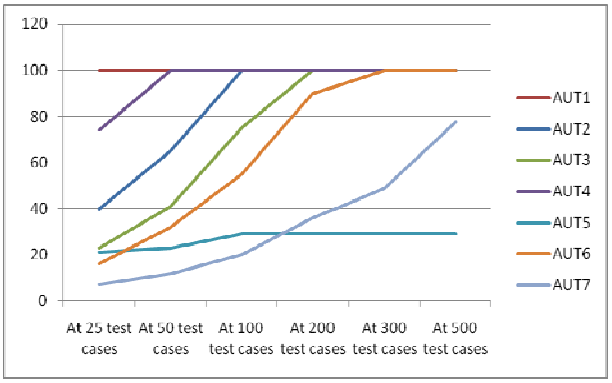

Fig. 2 Test case effectiviness using the optimization algorithm.

3.2 A Weight Selection Algorithm Based on Selecting Representatives.

In this research, the fitness functions (i.e., the effectiveess of the selected test set) assigns each test set a fitness value based on:

(1) The percentage of GUI controls covered in the set relative to the total number of controls in the application.

(2) The time at which each test set covers its associated GUI controls.

In genetic algorithms, the first pool of representatives are usually selected randomly and then optimized or modified according to a fitness function. Similarly, in this algorithm, the first pool of selected GUI components for test case generation will be selected randomly. Later on, all controls that are considered similar to this control will not be considered for the next round of controls' pool selection.

In this scenario, the tool will randomly select controls from the application under test and then reduce their weight or probability of selection for later cycles. Each time a control is selected in a test case, this will reduce its chance of being selected in the next rounds or cycles. If the same control is selected again, its weight or probability of selection is reduced further and so on. The pseudo code for this algorithm is:

(1) Select the first level control;

(2) Select randomly a child for the control selected in step one; Give equal weights for all children; Decrease weight for the selected one by a fixed value;

(3) Find all the children for the control selected in step two and randomly pick one child control; Give equal weights for all children and decrement the weight for the selected one by the same fixed value (this value can be the same for all levels, or each level can have a different value);

(4) Repeat step three until no child is found for the selected control;

(5) The test scenario for this cycle is then the sequence of the selected controls from all the previous steps;

(6) Repeat the above steps for the total number of the required test scenarios unless a termination process is called. Keep the decreased weights from the earlier scenarios.

Fig. 3 is a sample output from the weight selection algorithm applied on one AUT.

The number in the start of the test case represents its sequence of generation. The sample in Fig. 3 shows that all test cases of odd numbers are selected or 50 % of the generated test cases are eliminated as they are redundant.

3.3 Test Case Reduction through Selecting Representatives

Another approach that is inspired from genetic algorithms is “test case set reduction through selecting representatives”. This approach advances test selection reduction through selecting representative test scenarios. Human representatives are selected from the different categories, classes or areas to best or equally

```
1,NOTEPADMAIN,HELPTOPICSFORM,HELPTOPICS,INDEX,
LABEL4
3,NOTEPADMAIN,HELP,ABOUT NOTEPAD,,
5,NOTEPADMAIN,LABEL2,,,
7,NOTEPADMAIN,OPEN,OPENFILEBUTTON1,,
9,NOTEPADMAIN,VIEW,STATUS BAR,,
11,NOTEPADMAIN,OPEN,OPENFILECOMBOBOX2,,
13,NOTEPADMAIN,SAVE AS,SAVEFILELABEL4,,
15,NOTEPADMAIN,TXTBODY,,,
17,NOTEPADMAIN,BUTTON1,,,
19,NOTEPADMAIN,BUTTON2,,,
21,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL4,,
23,NOTEPADMAIN,LABEL1,,,
25,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL2,,
27,NOTEPADMAIN,PAGESETUP,PAGESETUPGROUPBOX1,
PAGESETUPRADIOBUTTON1
29,NOTEPADMAIN,LABEL1,STATUS BAR,,
31,NOTEPADMAIN,FORMAT,WORD WRAP,,,
33,NOTEPADMAIN,SAVE AS,SAVEFILECOMBOBOX3
35,NOTEPADMAIN,BUTTON1,ABOUTHELPLABEL4,,
37,NOTEPADMAIN,BUTTON3,,,
39,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,RE
PLACETABTXTFINDR,,
41,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL3,,
43,NOTEPADMAIN,FIND,TABCONTROL1,TABGOTO,GOTO
TABBTNGOTO,
45,NOTEPADMAIN,HELPTOPICSFORM,HELPTOPICS,CONT
ENT,LABEL2,
47,NOTEPADMAIN,PRINTER,PRINTERLABEL1,
49,NOTEPADMAIN,HELP,HELP TOPICS,,,
51,NOTEPADMAIN,FORMAT,FONT,FONTTEXTBOX3
53,NOTEPADMAIN,FIND,TABCONTROL1,TABGOTO,GOTO
TABLABEL4,
55,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,RE
PLACETABBTNREPLACEAL,
57,NOTEPADMAIN,OPEN,OPENFILELABEL1,,,
59,NOTEPADMAIN,PAGESETUP,PAGESETUPBUTTON2,,,
61,NOTEPADMAIN,FORMAT,FONT,FONTLABEL4,,
```

Fig. 3 A sample output from the weight selection algorithm.

represent the whole country and its different sectors. In this approach, the algorithm arbitrary selects a test scenario that includes controls from the different levels. The difference between this scenario and the earlier one is that in this case we looked at the test case as the chromosome whereas the control itself was the chromosome in the earlier one. Starting from the lowest level control, the algorithm excludes from selection all those controls that share the same parent with the selected control. This reduction shouldn’t exceed half of the tree depth. For example if the depth of the tree is four levels, the algorithm should exclude controls from levels three and four only.

Table 4 shows the results of applying the weight algorithm on several Applications Under Test (AsUT). The reduction percentage indicates the percentage of scenarios eliminated from the newly selected test set without affecting the test adequacy or coverage. This was a condition set in the earlier assumptions of those experiments (i.e., to reduce the number of generated test cases while sustaining the amount of coverage).

It should be noticed that we assumed that at least three controls are required for a test scenario (e.g., Notepad–File–Exit). Five test scenarios are continuously selected using the same reduction process descry-bed above. The selection of the number five for test scenarios is arbitrary. The idea is to select the least amount of test scenarios that can best represent the whole GUI of the AUT. Table 5 is a sample output of measuring test case reduction using the above algorithm. The five selected scenarios are listed along with their total reduction.

Table 4 Weight algorithm reduction percentages.

AUT (i.e., application under test)	Total number of test scenarios	Reduction percentage, (100–selected scenarios/ all scenarios)* %
Notepad	174	94.25
FP analysis	28	82.14
WordNet	8	75
Gradient	153	92.81
GUI controls	51	88.23
Hover	10	90

Table 5 A sample results of level-reduction testing algorithm.

Test scenarios	Total percent of test reduction%
Notepadmain, printer, printerbutton1,,, Notepadmain,save, savelabel7,, Notepadmain,edit,find,tabcontrol1, tabfind,findtabbtnnext Notepadmain,file,print, printtab,printlabel7, Notepadmain,save,savelabel5	65.1
Notepadmain,file,print,printtab, printlistbox1 Notepadmain,font,fontlabel2 Notepadmain,helptopicform, helptopics,search,button1 Notepadmain,font,fonttextbox2,, Notepadmain, printer, printerbutton2,,,	41.67
Notepadmain,file,print,printtab, printgroupbox1 Notepadmain, pagesetup,printer, Notepadmain,font,fontlistbox2,, Notepadmain,open,openfilelabel4,, Notepadmain,saveas, savefilecombobox2,	51.56

3.4 Weighting Controls from User Sessions

The previously described algorithms for test case generation that imitate genetic algorithms in principle depend on statistics pulled from the implementation model. As an alternative, we can analyze several user captured sessions (e.g., from testers or users in beta testing, or log files) to automatically weight the GUI controls. Higher weight for a GUI control means more probability of being selected in test cases. User sessions' data is the set of user actions performed on the AUT from entering the application until leaving it.

In order to record user events, the interface IMessageFilter is used to capture messages between Window applications and components. In the AUT, each GUI control that is triggered by the user is logged to a user session file. The minimum information required is the control, its parent and the type of event. The user session file includes the controls triggered by the user in the same sequence. Such information is an abstract of the user session sequence.

Controls are then given weights according to their occurrence in user sessions. The selected scenario includes controls from the different levels. Starting from the lowest level control, the algorithm excludes from selection all those controls that share the same parent with the selected control. Similar to the algorithm used earlier, this reduction shouldn't exceed half of the tree depth.

The developed application extracts the logging information in a universal text format that is independent of the application. We used this output as an input to the automated test execution process.

3.5 Using Genetic Algorithms for Test Case Generation and Optimization

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 the solution which it represents. The most used way of encoding is a binary string. The chromosome then may look like Table 6.

The main tasks that occur in the GA process are crossover, selection and mutation. In our experiment, an initial test set will be selected randomly through a test automation tool built for this purpose [5]. Crossover is used to optimize the selected set of test cases continuously. If in any test case, an invalid set of controls is selected (e.g., File-Copy-Exit), a mutation repair process will occur to switch a control for one of its alternatives to make sure that the generated test case is valid.

As explained earlier, using the GA first starts through defining the genes and chromosomes. The chromosomes are representing solutions to the problem. They should be improved from one generation to the

Table 6 Chromosomes' binary representation.

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

next. A fitness function takes a chromosome as input and returns usually a number as fitness value. The better the chromosome, the higher the fitness value will be. The genes represent individual components of a solution. The population is the set of chromosomes forming a generation. This population consists of chromosomes. Each chromosome contains a random collection of genes. The steps for GA generation are:

- Start by creating the initial population of chromosomes. The number of chromosomes or the population size is an important factor affecting the solution and the processing time it consumes. Larger population size (i.e., in the order of hundreds) increases the probability of obtaining a global best possible solution. However, it significantly increases the processing time.
- Evaluate the fitness of each chromosome and based on this fitness, select the chromosomes that will mate or produce better results.
- Cross over the selected chromosomes for possible better solutions.
- Randomly mutate some of the genes of the chromosomes. Repeat the previous steps until a new population is created. The algorithm ends when the best solution is found.

In test case generation from GUI components or controls, we have two choices for selecting the genes and chromosomes:

(1) Type 1. To consider each GUI control as a gene and then take each test case to be the chromosome (Fig. 4). The population then will be the test set or the set of the generated test cases.

(2) Type 2. To consider each test case as a gene and then take the test set as the chromosome (Fig. 5).

The main advantage of the first type is that it is straightforward, simple and quick to generate. However, it will not be easy to specify a fitness function for each test case in isolation. The execution time is relatively very short and the path coverage is very minor. In addition, crossover will, in most of the time, produce infeasible or invalid test cases (Fig. 6). A crossover repair process can be implemented whenever such infeasible solutions occur.

This is not the case in Type 2 where crossover will always produce valid chromosomes (Fig. 7). As a result, all next experiments will be applied to Type2 only.

In this experiment, 4 scenarios of population size of 10 chromosomes are selected. We will also select 4 scenarios for the number of test cases in every test set or chromosome: 10, 20, 50, and 100 test cases respectively. All test cases are generated and implemented on a small Notepad application built specifically for this

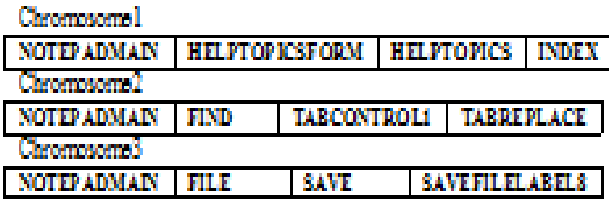

Fig. 4 Type 1 chromosomes presentation.

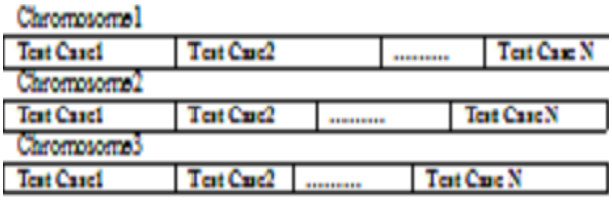

Fig. 5 Type 2 chromosomes' representation.

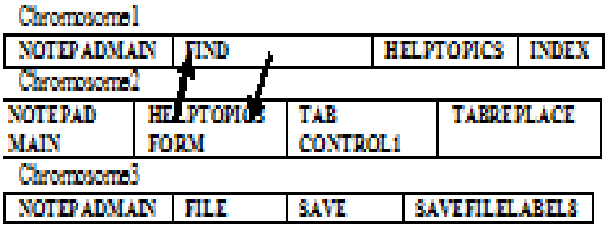

Fig. 6 Type1 chromosomes crossover.

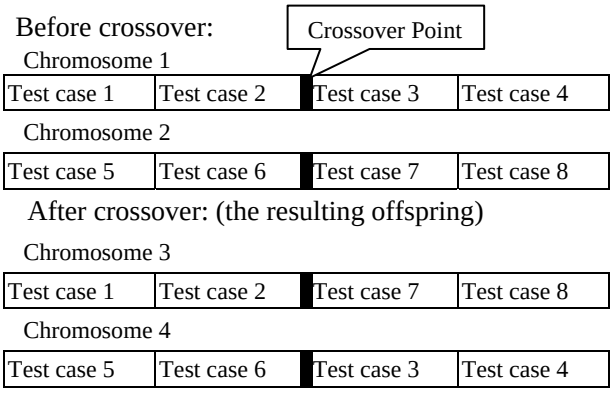

Fig. 7 Type 2 chromosomes crossover.

purpose. Two algorithms are used for the automatic generation of test cases. The difference between the two algorithms is that the first algorithm generates test cases randomly while the second algorithm uses some AI techniques to improve the coverage of the generated test cases. Table 7 shows the performance and coverage values for chromosomes of size 10 for the 2 algorithms.

The 3 pairs of parameters measured are: edges visited, processing time in milliseconds, and coverage, respectively. Those are abbreviated by the column ETC.

Table 8 and Fig. 8 summarize the results of the two proposed algorithms. Coverage reaches 100 % in about 400 test cases. However, this depends on the tested application and will vary from one application to another. Fig. 7 shows those results in columns graph. As units in the 3 attributes are different, algorithmic scale is used to be able to display all results in one graph. However, results shows that in both algorithms there is a direct positive correlation between the number of test cases selected and the 3 attributes: Edges, Time and Coverage.

The authors stopped the number of test cases in the first algorithm at 300 test cases as its coverage was not improving much relative to increasing the number of the selected test cases. There was no need to do crossover in the generated test sets as the algorithms are implemented with the constraint that all generated test cases are valid. Results indicate that GA can be used to determine the best selected chromosome based on the selected fitness functions.

Most chromosomes of all sizes show difference fitness values for the two algorithms. The larger the selected chromosome or test set is, the better we can judge the difference in the fitness functions.

The second test case generation algorithm was able to achieve full GUI paths' coverage for less than 400 test cases for the tested application.

The differences or the variations between the different chromosomes in the second algorithm were smaller than the differences or the variations in the first

Table 7 Performance and coverage values for chromosomes of size 10 for the 2 algorithms.

Chromo-some	ETC1 (Edges, time, coverage)	ETC2
10-1	11-31-0.055	13-31-0.065
10-2	12-15-0.06	9-31-0.045
10-3	12-15-0.06	16-46-0.08
10-4	10-15-0.05	12-31-0.06
10-5	8-15-0.04	13-31-0.065
10-6	7-31-0.035	11-31-0.055
10-7	4-15-0.02	10-31-0.05
10-9	3-31-0.015	13-31-0.065
10-10	3-15-0.015	14-31-0.07

Table 8 Edgec, time, and coverage overall results from the two developed algorithms.

Test NO	EDGS2	Time1, sec	Covg1	EGS2	Time2, Sec	Covg2
10	7	23	0.04	12	31	0.065
20	21	42	0.11	23	70	0.11
50	34	65	0.21	47	160	0.23
100	33	130	0.175	75	430	0.37
200	36	200	0.18	128	1135	0.64
300	46	210	0.25	177	1920	0.9
400	46	210	0.25	225	3145	1

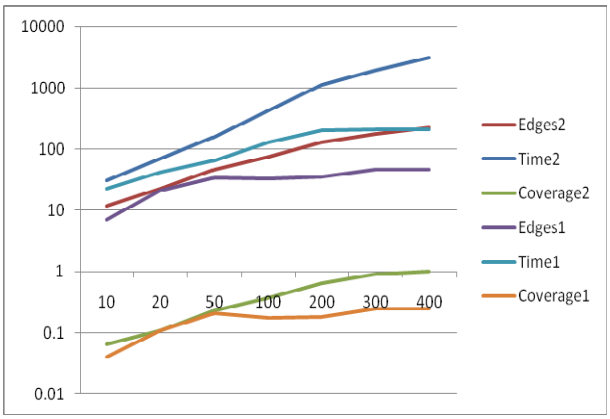

Fig. 8 Total ETC results, logarithmic scale.

algorithm. The first algorithm generates test cases randomly. To improve the test set coverage, the second algorithm uses some techniques to reduce redundancy in the random generation through eliminating the redundant test cases in the set and replace them always with unique ones. The test set coverage is the number of paths the test set visited to the total number of possible paths in the GUI.

Most traditional genetic algorithms depend on an initial set of chromosomes that are generated randomly. As such, the second algorithm can be considered as a hybrid algorithm that initially improves the possibility of generating test sets with good coverage. Second, it generates several chromosomes or test sets and select the best based on the selected fitness functions.

4. Conclusions

In this research, the authors proposed and evaluated test case generation and reduction techniques that depend on the principles of genetic algorithms. The goal was to automatically generate test cases that provide good coverage in terms of the paths it tests or visits within the user interface of the tested application. The idea of encoding the location of the controls allowed us to automatically test the overall sequence generated by each test case. The fitness functions selected in these experiments were the test set coverage of paths relative to the overall number of paths in the tested application and the test set execution time.

The performance measured here was the time it takes to generate the test cases. A fitness function that will be measured in future is the test set execution time. The authors will compare the correlation between test set generation and execution time.

The number of faults discovered is another fitness function that will be evaluated to select the best chromosome out of the pool of the generated ones. The tool can keep generating unique test sequences or scenarios until it finds errors.

Using the optimization theory for optimizing the process of test case generation was introduced in principles in this research. The goal and the parameters required to build the optimization matrix is defined. Results from this approach should be compared with results from GA algorithms for possible correlations and enhancements.

References

[1] J.H. Holland, Adaptation in natural and artificial systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, The MIT Press, 1992.

[2] D. Goldberg, Genetic Algorithm in Search, Optimization, and Machine Learning, Addison-Wesely, 1989.

[3] A. Avritzer, E.J. Weyuker, The automatic generation of load test suites and the assessment of the resulting software, IEEE Transactions on Software Engineering (1995) 705-716.

[4] A.M. Memon, M.E. Pollack, M.L. Soffa, Hierarchical GUI test case generation using automated planning, IEEE Transactions on Software Engineering 27 (2001) 144-155.

[5] I. Alsmadi, K. Magel, An object oriented framework for user interface test automation, available online at: http://www.micsymposium.org/mics_2007-/papers/Alsmadi.pdf, 2007.

[6] X. Yuan, Feedback-Directed Model-Based GUI Test Case Generation, Ph.D. dissertation, available online at: http://www.lib.umd.edu/drum-/bitstream/1903/8621/1/umi-umd-5735.pdf, 2008.

[7] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, A. Watkins, Breeding software test cases with genetic algorithms, In Proceedings of the 36th Annual Hawaii International Conference on System Sciences , Hawaii, USA, 2003, pp. 338.

[8] G. Huang, W. Fang, Automatic test case generation with

region-related coverage annotations for real-time systems, ATVA, 2005, pp. 144-158.

[9] B. Jones, H. Sthamer, X. Yang, D. Eyres, The automatic generation of software test data sets using adaptive search techniques, Third International Conference on Software Quality Management, 1999, pp. 435-444.

[10] B. Jones, D. Eyres, H. Sthamer, A strategy for using genetic algorithms to automate branch and fault-based testing, The Computer Journal 41 (1998) 98-107.

[11] J. Cherng, L.Y. Pu, Using genetic algorithms for test case generation in path testing, 9th Asian Test Symposium, 2000.

[12] R. Krishnamoorthi, S.A. Mary, Regression test suite prioritization using genetic algorithms, International Journal of Hybrid Information Technology (2009) 103-108.