Journal of Software Engineering and Applications, **2013, 6, 65-73**
http://dx.doi.org/10.4236/jsea.2013.62011 Published Online February 2013 (http://www.scirp.org/journal/jsea)

Using Genetic Algorithm as Test Data Generator for Stored PL/SQL Program Units

Mohammad A. Alshraideh, Basel A. Mahafzah, Hamzeh S. Eyal Salman, Imad Salah

The Department of Computer Science, The University of Jordan, Amman, Jordan.
Email: mshridah@ju.edu.jo

Received December 8th, 2012; revised January 9th, 2013; accepted January 20th, 2013

ABSTRACT

PL/SQL is the most common language for ORACLE database application. It allows the developer to create stored program units (Procedures, Functions, and Packages) to improve software reusability and hide the complexity of the execution of a specific operation behind a name. Also, it acts as an interface between SQL database and DEVELOPER. Therefore, it is important to test these modules that consist of procedures and functions. In this paper, a new genetic algorithm (GA), as search technique, is used in order to find the required test data according to branch criteria to test stored PL/SQL program units. The experimental results show that this was not fully achieved, such that the test target in some branches is not reached and the coverage percentage is 98%. A problem rises when target branch is depending on data retrieved from tables; in this case, GA is not able to generate test cases for this branch.

Keywords: Genetic Algorithms; SQL Stored Program Units; Test Data; Structural Testing; SQL Exceptions

1. Introduction

PL/SQL is an imperative third generation language (3GL) that was designed specifically for the processing of SQL commands. It provides specific syntax for this purpose and supports exactly the same data types as SQL. ORACLE database can be accessed by calling PL/SQL named block that include functions and procedures. Therefore, they must be executed properly in order to guarantee a reliable and confidence database system [1].

Software testing is an important stage of software development life cycle (SDLC). It is an activity that helps finding out bugs and errors in a software system that is under development in order to provide a bug free and reliable system/solution to the customer [2]. Testing has two main types based on the knowledge of the system: black box testing (functional) and white box testing (structural) [2-4]. The functional testing deals with the system as a black-box that does not explicitly use knowledge of the internal structure; which means it usually makes sure that the system is working according to the system requirements, while the structural testing generates the test data depend on the knowledge of internal code of the system. During structural testing, the goal is to generate a test data which satisfy a given testing criterion to cover given elements of the program. In this paper, the branch coverage criterion is considered, where each branch of the program should be reached by some test data [2].

Generally, structural testing techniques are classified into two categories: static testing (manual) and dynamic testing (automatic). In the static testing, a code reviewer reads the source code statement by statement and visually follows the logical program flow by feeding an input, so it is costly. In contrast, dynamic testing techniques execute the program under test on test input data and then simply observe the results. Consequently, dynamic testing reduces the cost of software development and maintenance [2]. Search-based software testing is an example of dynamic method used to generate test set that can be successfully applied in structural testing. It relies on a cost function that can be used to compare candidate test data [5].

Genetic algorithms (GA) have been very interesting area of study in many disciplines, such as optimization, automatic programming, economics, immune systems, ecology and social systems. In this paper we apply the GA as a search technique to find test data to test named block in ORACLE; specifically, IF-statement and While-statement and their combinations are considered [6].

The rest of the paper is organized as follows: Section 2 presents background and related work. Section 3 presents a strategy for applying GA to test named block in ORACLE. Section 4 presents experimental environment and Section 5 presents experimental results. Finally, Section 6 concludes the paper.

2. Background and Related Work

This section presents an overview of evolutionary algorithms; such as random test data generation and Hill Climbing, and meta-heuristic search algorithms which proposed a potential better alternative for developing test data generators [7,8]. Efficient existing meta-heuristic search algorithms include Simulated Annealing, Tabu Search, GA and Ant Colony Optimization. Each of these search algorithms has its own advantages and disadvantages over the others. They are strongly domain dependent problem, because they use domain dependent knowledge or heuristics related to the problem domain under consideration. Also in this section, stored program units are explained and an overview about Jordan University Hospital Computer systems is presented.

2.1. Random Test Data Generation

Random test data generation is a technique based on selection test data randomly until the suitable test data is found. It only explores the search space by randomly selecting solutions and evaluating their fitness. This is quite an unintelligent strategy but it does not take much effort to be implemented [9].

2.2. Hill Climbing

Hill Climbing is a well known local search algorithm. Hill Climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighborhood of this solution is investigated. If a better solution is found, then the current solution is replaced. The neighborhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again, and so on, until no improved neighbors can be found for the current solution. Hill climbing is simple and gives fast results. However, it is easy for the search to yield sub-optimal results when the Hill Climbing leads to a solution that is locally optimal, but not globally [10].

2.3. Simulated Annealing

Simulated Annealing (SA) extends Hill Climbing such that it accepts poor solutions with low probability. SA allows for less restricted movement around the search space. The probability of acceptance (p) of an inferior solution changes as the search progresses, and is calculated as in Equation (1) [11,12].

$$p = \mathrm{e}^{-\delta/t} \tag{1}$$

where (δ) represents the difference in the objective value between the current solution and the neighboring inferior solution being considered, and (t) is a control parameter known as the temperature. The temperature is cooled according to a cooling schedule. Initially the temperature is high, in order to allow free movement around the search space. As the search progresses, the temperature decreases. However, if cooling is too rapid, not enough of the search space will be explored, and the chance of the search becoming stuck in the local optima is increased [12].

2.4. The Principles of Genetic Algorithms

The basic concepts of GAs are developed by Holland [13]. GAs is commonly applied to a variety of problems involving searching and optimization. GAs search methods are rooted in the mechanisms of evolution and natural genetics. GAs draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. GAs generates a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms [14-16].

The principle behind GAs is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous to the chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to the rules of selection, mutation and reproduction. Each individual in the environment (represented by a chromosome) receives a measure of its fitness. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given in **Figure 1**. The algorithm in **Figure 1** will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have occurred.

```
Simple Genetic algorithm ( )
 {
   initialize population;
   evaluate population;
   while termination criterion not reached
       {
            select solutions for next population;
            perform crossover and mutation;
            evaluate population;

       }
 }
```

Figure 1. The structure of a simple GA.

2.5. Stored PL/SQL Program Units

There are three types of stored program units in PL/SQL; procedures, functions, and packages. Every stored program unit has a declarative part, an executable part or body and an exception handling part which is optional [1]. Declarative part contains variable declarations. Body of the named block contains executable statements of SQL and PL/SQL. Statements to handle exceptions are written in exception part. However, subprograms provide the following advantages [13]:

- They allow you to write PL/SQL program that meets our need.
- They allow you to break the program into manageable modules.
- They provide reusability and maintainability for the code.

Procedure is a subprogram used to perform a specific action. A procedure contains two parts; specification and body. Procedure specification begins with the procedure name and ends with parameters list. Procedures that do not take parameters are written without a parenthesis. The body of the procedure starts after the reserved word (IS) or (AS) and ends with keyword END [17]. A function is PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is the function must always return a value, but a procedure does not return a value [1,17]. A package is an encapsulated collection of related program objects (for example, procedures, functions, variables, constants, cursors, and exceptions) stored together in the database.

Using packages is an alternative to creating procedures and functions as standalone schema objects. Packages have many advantages over standalone procedures and functions. For example, they:

- Let you organize your application development more efficiently.
- Let you grant privileges more efficiently.
- Let you modify package objects without recompiling dependent schema objects.
- Enable Oracle Database to read multiple package objects into memory at once.
- Can contain global variables and cursors that are available to all procedures and functions in the package.
- Let you overload procedures or functions. Overloading a procedure means creating multiple procedures with the same name in the same package, each taking arguments of different number or data type.

2.6. Jordan University Hospital Computer System

The core of Jordan University Hospital (JUH) information system is bought in 1994, and then the JUH IT team developed the Hospital Information System (HIS) using Oracle forms, and upgrades it to Oracle 10 g. HIS developed to provide best medical services for patients and physicians. Delivering these services require hospitals to review the way they manage their business processes and supply more efficient features to physicians, patients, and hospitals officials as well as other decision makers. In order to provide such services, the health facility must focus on developing a solution to connect all its resources and makes it available to all who needs utilizing it using latest technology. This kind of solution will enhance the performance and optimize the efficiency and will reduce the cost of ownership.

IT department in JUH creates a solution suite that transforms the hospital to a community allowing the access to all resources and data as needed. HIS is a comprehensive solution developed specifically for health facilities in the region. It is flexible, comprehensive, multilingual, integrated and secured solution that supports clinical, financial, administration and higher management needs.

In general, a hospital management system can be sub categorize into the following groups (**Figure 2**):

- Medical Information System (Administrative and Clinical).
- Enterprise Resource Planning (ERP) (Material, Financial and Human Resources).
- Support System.

Medical systems are developed to deliver all needed services to the hospital community (Physicians, Patients and Administration). The systems manage all patients' data and information during their treatment episode in a professional and efficient manner. Medical systems strategically support a full range of hospital functions. It contains a repository of all patients' clinical, billing and

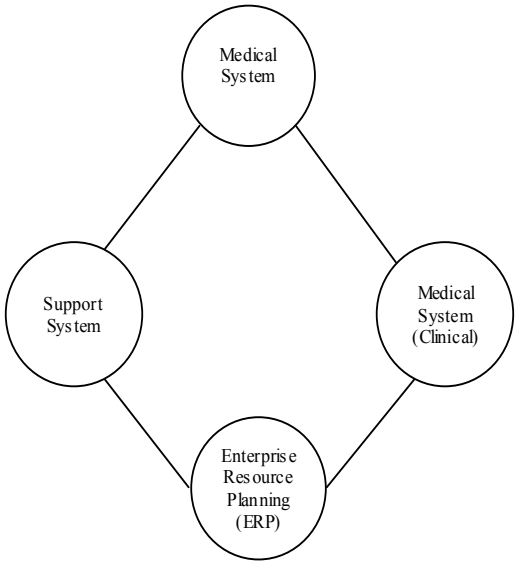

Figure 2. Hospital management system sub-categorizes.

demographic data, reducing paper work, manual effort and errors. Furthermore; it allows for better staff utilization allowing for more time to focus on planning and goals achievements. This enables the hospital to provide better quality and more efficient services, needed by patients and physicians. Medical systems are integrated with financial, administration, human resources, and material management systems. It contains vast collection of data including patient data, treatment data, hospital visit data, patient transactions data, hospital data, and statistical information.

HIS medical systems provide many key functions including:

- ♦ Medical administrative including:
- Patient master index
- Admission, discharge and transfer
- Scheduling and appointments
- Medical records
- Medical reports
- Medical statistics
- Catering
- Order entry and results communication
- ♦ Medical clinical including:
- Out-patient clinics
- Accidents and emergency
- Operation theater
- Maternity
- Doctors desktop
- Nurse station
- Laboratory
- Radiology
- Pharmacy
- ♦ Patient accounting including:
- Pricing and package deals
- Patient billing
- Insurance contract management
- Claims management

In order to test our system in this research we will select different procedures and functions, which will be described later in this paper.

3. A Strategy for Applying GA to Test Stored PL/SQL Program Units

In general, the process of automatic structural test data generation for branch coverage consists of three major steps [7,18,19]:

1) Construction of control logic graph, e.g. control flow graph (CFG) or control dependency graph (CDG).

2) Selection the target according to branch coverage criterion.

3) Finding out a set of test data that satisfies the selected adequacy criterion.

In order to use GA for solving an optimization problem, there are multiple issues must be considered such as: how to build the fitness function and how to represent the problem in a chromosome expression (individual), *i.e.* sort of a sequence of binary digits that resembles the chromosome sequence, which GA can understand and manipulate. GA works on this encoded problem and delivers the result as the problem solution; hence, the user should provide the meaning of the encoded problem [20, 21]. In this paper integer vector and binary string representation will be considered.

3.1. Branch Cost Functions

To use a control dependency path or any set of branches as a search goal, it is necessary to determine the cost values for each branch predicate. To accomplish this, each conditional node in the program is associated with a real-valued predicate cost function that is evaluated whenever the conditional node is executed. This predicate cost function returns a positive value whenever the predicate is false and a negative value if the predicate is true. The cost of an evaluation of a logical negation of a predicate is the arithmetic negation of the cost of the evaluation of the predicate. Each reached branch maintains two cost values, both derived from the associated predicate cost function. One cost value is the cost that all attempts to execute the branch are successful. This is called the cumulative ***and-cost***. The other cost value is the cost when any attempt is successful, is called the cumulative ***or-cost***. These costs can be illustrated with an example showing three failed and two successful attempts to execute the predicate $\boldsymbol{a \le b}$ for various integer values of $\boldsymbol{a}$ and $\boldsymbol{b}$ (**Table 1**). The predicate cost function is $\boldsymbol{a - b}$ when the predicate is false and $\boldsymbol{a - b - 1}$ when the predicate is true. The cost function of ***or-cost*** *and* ***and-cost*** are shown in **Table 1**, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are positive (*false*), and $\boldsymbol{a}$` and $\boldsymbol{b}$` are negative (*true*), also $\boldsymbol{a}$ and $\boldsymbol{b}$ are never zero.

The cost values produced by relational predicates are normalized, but the un-normalized values are used in **Table 2**. The cost of a conjunction of two false costs is the sum of the costs of the conjuncts. However, the cost of a disjunction of two false costs (Cost_d) is shown in Equation (2), where P and Q are the disjunct costs.

$$\text{Cost}_d = \frac{PQ}{P+Q} \tag{2}$$

Table 1. Logical *or-cost* and logical *and*-cost table.

a	*b*	*or-cost*	*and-cost*
a	b	(ab)/(a + b)	a + b
a	b`	b`	a
a`	b	a`	b
a`	b`	a` + b`	(a`b`)/(a` + b`)

Table 2. Cumulative or-cost and and-cost for the predicate $a \le b$ for the listed values.

a	*b*	*cost*	*or-cost*	*and-cost*
8	3	5	5	5
6	3	3	15/8	8
5	3	2	30/31	10
3	3	−1	−1	10
1	3	−3	−4	10

These costs can be illustrated with an example showing three failed and two successful attempts to execute the predicate $a \le b$ for various integer values of a and b as shown in **Table 2**. Note that when both branches at a conditional node have been executed, the ***and-cost*** is positive and the ***or-cost*** is negative. Moreover, the magnitude of the ***and-cost*** is an indication of the number and magnitude of the failures to satisfy the predicate. A high ***and-cost*** indicates that the predicate has hardly been satisfied. A low ***and-cost*** indicates that the predicate has not been satisfied. The cost of a search goal is calculated, according to Equation (2), as the conjunction of the individual branch goal costs. Each individual branch cost is either a branch ***or-cost*** or a branch ***and-cost***. Using this method, there is a disadvantage that a single large branch cost may dominate the overall cost value. Normalization or costs reduces this risk. The cost values produced by relational predicates (Cost_r) are normalized to lie within [−1, 1] using Equation (3), where c is the branch distance value.

$$\text{Cost}_r = \begin{cases} 1 - \dfrac{1}{1+c} & \text{if } c > 0 \\ \dfrac{1}{1-c} - 1 & \text{if } c < 0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

An alternative method, not used in the work reported here, is to compute a cost consisting of two components. One component, counts the branch goals that have yet to be satisfied. This cost component is the analogue to the ones used by Wegener *et al.* [22]. The second component is applicable only if the first component is nonzero and it is calculated as the disjunction of the unsatisfied branch goals. For each branch, there are two associated branch search goals that may be specified to guide a search, namely ***branch-or*** (on at least one occasion that the branch is reached, and it is executed) and ***branch-and*** (on every occasion that the branch is reached, and it is executed). A branch goal is satisfied if the associated ***or-cost*** or ***and-cost*** is negative.

If the execution of a branch is required to satisfy branch coverage or a control dependency condition then ***branch-or*** is the relevant branch goal. Note that the goal of satisfying the branch ***or-cost*** is not adopted primarily to avoid creating an excessive number of sub-goals. Moreover, if it is necessary to find an input that executes both branches at a predicate, it is hoped that such an input will be found during the search for an input to satisfy the ***and-cost***. Recall that, the ***and-cost*** is adopted as the branch goal after one branch at a predicate has already been executed. The above rules are applied to non-loop branches only. Loops are treated different than if-statement, because for programs that terminate loop entries are eventually followed by a loop exit. For this reason, a subgoal that specifies a loop predicate is always true is not sensible and thus the two possible branch goals at a loop condition are loop entry and no loop entry.

4. Experimental Environment

An experimental study was designed to feature test goals that cause problems for evolutionary testing. The experimental study featured JUH real six test objects. These objects are drawn from the system applied at JUH hospital.

4.1. Test Objects

This section describes the test objects and the input domain sizes used. The following are source code for the test objects:

- **OutPricing:** Determines the pricing of treatments at outpatient clinics. Depending on his insurance the patient, this function calculates the amount of money the patient has to pay (depending on the type of insurance, the patient pays different ratios for his treatment) and the amount of money the insurance has to pay for the patient's treatment.
- **InPricing:** This procedure calculates the invoice value of the patient inside the hospital based on the type of patient insurance and the type of medical procedure offered to patients (accommodation, scouting, doctors' fees, operations, laboratory, radiology, medicine, etc.). Also, this program calculates the percentage paid by the patient and the percentage paid by the insurance company, if any. Moreover, this function bills the patient with the amount of money he has to pay and bills the insurance company with the amount of the money it has to pay.
- **JU-Med-fees-deduction:** This package used for Jordan University staff, where there is an allocated account number for each staff in the system of JUH. It calculates bill value based on Jordan University insurance. Then deported the total amount of bill after deduct the hand-collect from the patients into tables to be used in Jordan University financial department

later on.

- **Pat-info-ibr:** This function calculates the invoice value for private patients (in patients and out patients), then bills the patients with the amount of money he or she has to pay.
- **Lab-interface:** The main goal of this function is to transfer the results from medical machines (lab devices) to HIS system automatically (without user interaction). So, the function receives the message from medical devices then converts it to be entered to HIS system.
- **Salup_new_calc_all:** This procedure calculates staff incentives as follows: it selects the category that owns the nursing, administrative, officer or a medical technician, by the department and qualifications. Then it determines the share of the incentives that the employee is entitled, as his career (Branch Chief, Chief, Division of, etc.). It discounts days leave without pay from the employee share incentives.

4.2. Hardware and Software Environment

In this section, the specifications of the experimental environment utilized by this work are presented. These specifications include both hardware and software modules used in implementing the simulator. More specifically, the hardware specifications that are used in the experiments include a Dual-Core Intel Processor (CPU 2.66 GHz), 2 MB L2 Cache per CPU, and 1 GB RAM. Moreover, the software specifications that are used in the experiments include windows XP. Also, the tested programs that have been used to evaluate this algorithm are described in this section.

Moreover, in order to assess the reliability of the cost functions introduced in the previous Section 4.1, an empirical investigation was done. A number of test programs were assembled from JUH system including functions and procedures. These programs are described in **Table 3**. The size of each program is given as Lines of Code (LOC), number of branches, where the number of input variables ranges from 3 to 21, as shown in **Table 3**. The programs have been selected from JUH HIS system. **Figure 3** shows a cyclomatic complexity for each program. The cyclomatic complexity metric is described by Watson and McCabe [23], which provides an objective measure of the complexity of a given module of a program code by examining its decision structure. Cyclomatic complexity is calculated as $e - n + 2$, where n is the number of nodes in a graph and e is the number of edges between nodes, or we can calculate cyclomatic complexity as $P + 1$, where P is the number of predicate nodes in the flow graph (While and If statements). Predicate nodes are those representing control structures and have one or more edges emanating from them. Cyclomatic complexity gives an upper bound on the number of test cases required to cover all feasible branches if collateral coverage is taken into account. Each program has special characteristics to investigate the performance of GA as test data generator in order to test named block in ORACLE. **Figure 3** shows a cyclomatic complexity of the test programs. The range of program's cyclomatic complexity is between 25 and 270.

Table 3. The functions used for empirical investigation.

Program name	Lines of code	Number of branches	Number of input variables
OutPricing	295	116	14
InPricing	362	148	13
JU-Med-fees-deduction	307	92	4
Pat-info-ibr	259	48	3
Lab-interface	1389	538	21
Salup_new_calc_all	707	326	6

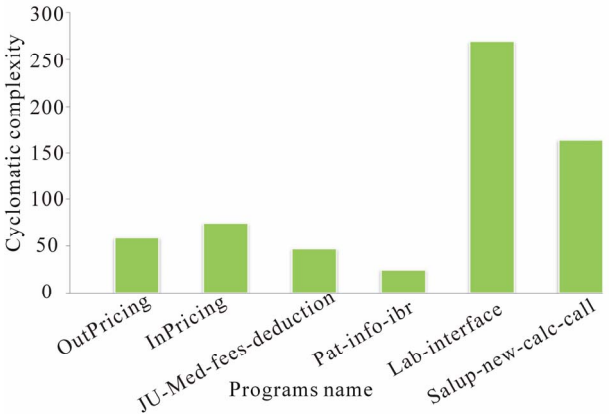

Figure 3. A cyclomatic complexity of the test programs.

These programs are available from the authors on request. Each of the cost functions and associated search operators were implemented in a prototype test data generation tool. The tool has been constructed by modifying the JScript (JavaScript) language compiler within the Shared Source Common Language Infrastructure (SSCLI) and can therefore be used to test PL/SQL Unit by passing test cases as parameters to these units, while these programs are connected to JUH HIS. The program must include directives to specify any input domain constraints that are to be applied. The tool then inserts instrumentation code at each branch in the function. This instrumenttation code calculates the cost of each branch predicate whenever it is executed. The cost of each relational predicate expression was calculated according to the cost functions given in the previous Section 3.1. Where branch predicate expressions consist of two or more relational predicates joined by logical connectives, ***and***, ***or*** and ***not***,

and the cost values were combined according to the scheme given in Bottaci [24].

The search was directed to generate data for one branch at a time. The order in which the branches of the program were targeted was arbitrary, except that no nested branch was targeted before the containing branch.

A steady-state style genetic algorithm, similar to Genitor [25], was used in this work. The cost function values computed for each candidate input were used to rank candidates within the population in which no duplicate genotypes are allowed. A probabilistic selection function selected parent candidates from the population with a probability based on their rank, where the highest ranking having the highest probability. More specifically, for a population of size n, the probability of selection (P_s) is shown in Equation (4).

$$P_s = \frac{2(n - \text{rank} + 1)}{n(n-1)} \quad (4)$$

In this work, a fixed population size of 100 was used. This parameter was not "tuned" to suit any particular program under test. In a steady state update style of genetic algorithms (as used in this work); new individuals that are sufficiently fit are inserted in the population as soon as they are created. Full branch coverage was attempted for each of the programs under test. Each branch was taken as individual target of the search, unless it was fortuitously covered during the search for test data for another branch. Genetic algorithms search generates inputs for the function containing the current structural target. A vector of floating point, integer, characters, and string variable values corresponding to the input data is optimized. The ranges of each variable are specified. The test subject is then called with this input data. The criterion to stop the search was set up to terminate the search after 50,000 executions of the program under test, when only if full coverage was not achieved. Individuals were recombined using binary and real-valued (one-point and uniform) recombination, and mutated using real-valued mutation. Real-valued mutation was performed using "Gaussian distribution" and "number creep". The size of any of the programs are higher of those typical programs that would used in unit testing, although the number of branches is probably higher than usual, which is why it were selected for the experiment.

5. Experimental Results and Discussions

This section presents the results of the experiments that have been carried out to evaluate the effectiveness of our GA. **Table 4** shows the number of subject program executions required by each genetic algorithm over 20 trials, where >50,000 means that the cost function not able to cover all the branches within this criteria. Also, in this table branch coverage ratio is shown, which is defined as the following:

$$\frac{\text{number of branch executed}}{\text{total number of branches}}$$

The branch coverage ratio ranged from 94% in **JU-Med-fees-deduction** program to 100% in **Pat-info-ibr** program. The total average of branch coverage for all programs is:

$$\frac{1242}{1268} \approx 98\%$$

Analyzing our results, we found that a 100% of condition-decision coverage is impossible to reach in some test programs because there are conditions that cannot be true or false in certain situations. So, a weakness of GA could be observed in the generation of test cases to cover branch, especially when there is a strong dependency in data (records) retrieved from table, such that a specific order is required. As long as there are only few records that play an important role to satisfy a certain condition, it is possible to find adequate test scenarios. For example, this could be observed while applying GA on **OutPricing** function, as shown in **Figure 4**, in line 21 insur ance_status to be equal to 7, this only happen if line 18executed and this happen only if line 16 execute branch to be true (execute insert command and insurance_status = 7) this happen only when dummy variable is equal to "p" in line 2 and this is depends on the value

Table 4. The number of branch covered and uncovered with 50,000 executions of each program.

Program Name	Number of branch covered	Number of branch uncovered	Branch coverage ratio	Number of subject program executions
OutPricing	112	4	96%	>50,000
InPricing	144	2	99%	>50,000
JU-Med-fees-deduction	87	5	94%	>50,000
Pat-info-ibr	48	0	100%	11,680
Lab-interface	527	11	98%	>50,000
salup_new_calc_all	321	5	98%	>50,000


```
[1]  …
[2]  select  p_insur_type into dummy from pricing   where    …
[3]  …
[4]  select decode(p_insur_type, '1', prc_limit_out_e, '2', prc_limit_out_f)
[5]     into p_max_cov
[6]     from prc_limits
[7]     where prc_group    = p_group_id
[8]       and prc_division = p_div;
[9]    p_max_cov := nvl(p_max_cov, 99999);
[10] exception
[11]   when no_data_found then
[12]       p_error_no := 1; raise exit_proc;
[13]   when others then
[14]       p_error_no := 11; raise exit_proc;
[15] ….
[16] if  dummy = 'p' then
[17]          insert into ….
[18]          insurance_status :=7;
[19] end if;
[20] …
[21]   if  insurance_status  =7 then
[22]   update out_invoice
[23]        set …
[24] end if;
[25] when exit_proc then
[26]   if p_error_no = 1 then
[27]      raise_application_error( -20001,' Coverage Limits do not Exits');
[28]   elsif p_error_no = 2 then
[29]      raise_application_error( -20003,  ' Rate pricing does not exits for this materials!!');
[30]   elsif p_error_no = 3 then
[31]      raise_application_error( -20004,   ' Material is not Defined in the Table Price !!');
[32]   elsif p_error_no = 5 then
[33]      raise_application_error( -20001, 'Pricing Data is incomplte for this patient!!');
[34]   elsif p_error_no = 11 then
[35]            …
[36]   end if;
```

Figure 4. OutPricing program fragment.

retrieved from pricing table, also in line 20 to execute branch to be true this happen only when line 12 is executed. In this case, where the target of the search is node 20 to be true, the fact that **p_error_no** needs to be 1 at line 12 and this happen only when the select statement in line 4 executes and exception no_data_found rose. This also is applied to lines 22, 24, 26, 28, 30 and 32. These situations are occurred in all other programs apart from **Pat-info-ibr**, where GA generates test data for all branch. In these cases, the test generator cannot reach the 100% of coverage due to the test program itself. With respect to the program **Pat-info-ibr**, there are branch conditions depend on data retrieved from table, but by coincidence, these branches are covered. This problem become more difficult when there is more than branch depends on uncovered branch.

In **Figure 5**, we notice that all branches are covered, the number of execution of program ranged from 11,680 for **Pat-info-ibr** to 36,134 for **salup_new_ calc_all**.

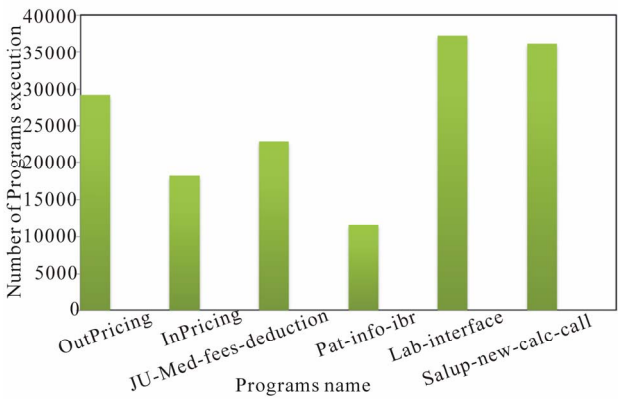

Figure 5. The number of executions required to find test data to achieve branch coverage after excluding uncovered branches in Table 4.

6. Conclusions and Future Work

In this paper we present how GA can be used as test data generator to find suitable test data according to branch criteria to test stored program units (procedures, functions, and packages) in ORACLE. Selected procedures, functions, and packages from Jordan University Hospital Information System are used to test GA. The experimental results show that the test target in all programs under test is not reached and that the average coverage ratio percentage is 98%. A problem occurs when the target branch depends on data retrieved from oracle tables. That GA cannot generate test data to execute the target branch that depends on data retrieved from tables.

The future work will be focused on testing SQL exceptions, SQL statements, and combinations between branch coverage criteria and SQL commands testing try-

ing to increase the coverage ratio to 100%.

REFERENCES

[1] M. G. Alshraideh and L. Bottaci, "Search-Based Software Test Data Generation for String Data Using Program-Specific Search Operators," Special Issue of Software Testing, Verification and Reliability Devoted to Extended Papers from the Third UK Testing Conference (UKTest 2005), Vol. 16, No. 3, 2006, pp. 175-203.

[2] M. Alshraideh, B. A. Mahafzah and S. Al-Sharaeh, "A Multiple-Population Genetic Algorithm for Branch Coverage Test Data Generation," *Software Quality Control*, Vol. 19, No. 3, 2011, pp. 489-513. doi:10.1007/s11219-010-9117-4

[3] M. Alshraideh, L. Bottaci and B. A. Mahafzah, "Using Program Data-State Scarcity to Guide Automatic Test Data Generation," *Software Quality Control*, Vol. 18, No. 1, 2010, pp. 109-144. doi:10.1007/s11219-009-9083-x

[4] A. Baresel, H. Pohlheim and S. Sadeghipour, "Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms," *Proceedings of the Genetic and Evolutionary Computation Conference*, Chicago, 12-16 July 2003, pp. 2428-2441.

[5] B. Korel, "Automated Test Generation for Programs with Procedures," *Proceedings of the International Symposium on Software Testing and Analysis*, San Diego, 8-10 January 1996, pp. 209-215.

[6] S. N. Sivanandam and S. N. Deepa, "Introduction to Genetic Algorithms," 1st Edition, Springer, New York, 2010.

[7] C. C. Michael, G. E. McGraw and M. A. Schatz, "Generating Software Test Data by Evolution," *IEEE Transactions on Software Engineering*, Vol. 27, No. 12, 2001, pp. 1085-1110. doi:10.1109/32.988709

[8] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, 1990, pp. 870-879. doi:10.1109/32.57624

[9] J. A. Edvardsson, "Survey on Automatic Test Data Generation," *Proceedings of the Second Conference on Computer Science and Engineering*, Linkoping, 21-22 October 1999, pp. 21-28.

[10] J. Duran and S. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 438-444. doi:10.1109/TSE.1984.5010257

[11] M. Harman and P. McMinn, "A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation," *Proceedings of the* 2007 *International Symposium on Software Testing and Analysis*, London, 9-12 July 2007, pp. 73-83. doi:10.1145/1273463.1273475

[12] P. McMinn, "Search-Based Software Test Data Generation: A Survey: Research Articles," *Software Testing, Verification & Reliability*, Vol. 14, No. 2, 2004, pp. 105-156. doi:10.1002/stvr.294

[13] H. S. Eyal Salman, "Using Genetic Algorithm in Test Data Generation for ORACLE Named Block," Master Thesis, The University of Jordan, Amman, 2010.

[14] H. W. Arthur and J. Thomas, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," National Institute of Standards, Gaithersburg, 1996.

[15] M. Mitchell, "An Introduction to Genetic Algorithms," 1st Edition, Massachusetts Institute of Technology, Cambridge, London 1996.

[16] M. Srinivas and L. M. Patnaik, "Genetic Algorithms: A Survey," *IEEE Computer*, Vol. 27, No. 6, 1994, pp. 17-26. doi:10.1109/2.294849

[17] S. Kirkpatrick, C. D. Gellat and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, No. 4598, 1983, pp. 671-680. doi:10.1126/science.220.4598.671

[18] S. Urman, R. Hardman and M. McLaughlin, "Oracle Database 10g Pl/SQL Programming," 1st Edition, McGraw-Hill, New York, 2004.

[19] A. H. Watson and T. J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," NIST Special Publication, No. 500-235. National Institute of Standards and Technology, Gaithersburg, 1996.

[20] N. J. Tracey, J. Clark, K. Mander and J. McDermid, "An Automated Framework for Structural Test-Data Generation," *Proceedings* 13*th IEEE Conference in Automated Software Engineering*, Hawaii, 13-16 October 1998, pp. 285-288.

[21] R. Pargas, M. Harrold and R. Peck, "Test-Data Generation Using Genetic Algorithms," *Software Testing, Verification and Reliability*, Vol. 9, No. 4, 1999, pp. 263-282. doi:10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y

[22] J. Wegener, A. Baresel and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology*, Vol. 43, No. 14, 2001, pp. 841-854. doi:10.1016/S0950-5849(01)00190-2

[23] J. Holland, "Adaptation in Natural and Artificial Systems," University of Michigan Press, Ann Arbor, 1975.

[24] L. Bottaci, "Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data," *Genetic and Evolutionary Computation Conference* (*GECCO* 2003), Chicago, 12-16 July 2003, pp. 2455-2464. doi:10.1007/3-540-45110-2_149

[25] D. Whitley, "The genitor Algorithm and Selective Pressure: Why Rank-Based Allocation of Reproductive Trials Is Best," *Proceedings of the Third International Conference on Genetic Algorithms* (*ICGA*-89), 1989, Morgan Kaufmann Publishers Inc., San Francisco, pp. 116-121.

