PWiseGen: Generating Test Cases for Pairwise Testing Using Genetic Algorithms

Pedro Flores and Yoonsik Cheon

TR #11-06
January 2011

Keywords: combinatorial testing, genetic algorithms, pairwise testing, software testing, test coverage.

1998 CR Categories: D.2.5 [*Software Engineering*] Testing and Debugging — testing tools (e.g., data generators, coverage testing); I.2.8 [*Artificial Intelligence*] Problem Solving, Control Methods, and Search — Graph and tree search strategies.

To appear in the *2011 IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011)*, June 10-12, 2011, Shanghai, China.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

PWiseGen: Generating Test Cases for Pairwise Testing Using Genetic Algorithms

Pedro Flores
Information Technology Department
Universidad Autónoma de Ciudad Juárez
Juárez, Mexico
Email: pedro.flores@uacj.mx

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
Email: ycheon@utep.edu

***Abstract*—Pairwise testing is a combinatorial testing technique that tests all possible pairs of input values. Although, finding a smallest set of test cases for pairwise testing is NP-complete, pairwise testing is regarded as a reasonable cost-benefit compromise among combinatorial testing methods. In this paper we formulate the problem of finding a pairwise test set as a search problem and apply a genetic algorithm to solve it. We also describe an open-source tool called PWiseGen for generating pairwise test sets. PWiseGen produces competitive results compared with existing pairwise testing tools. Besides, it provides a framework and a research platform for generating pairwise test sets using genetic algorithms; it is configurable, extensible, and reusable.**

Keywords: combinatorial testing, genetic algorithms, pairwise testing, software testing, test coverage.

I. Introduction

Pairwise testing is an effective, combinatorial testing technique that, for each pair of input parameters to a software system, tests all possible combinations of these parameters [1]. It is based on the observation that most software errors are caused by interactions of at most two factors such as input values. Its test suite is much smaller than that of exhaustive testing yet still very effective in finding errors. However, one problem of pairwise testing is that finding the least number of test cases has been proven to be an NP-complete problem [2]. This means that an efficient way to find an optimal solution is not known and that the time required to find a minimum number of test cases grows rapidly when the numbers of parameters and possible values increase.

A genetic algorithm is a technique that simulates the natural process of evolution [3]. It was discovered as a useful tool for dealing with search and optimization-related problems and is known to be effective for finding solutions for problems with a huge search space and complexity. In a genetic algorithm, a population of candidate solutions, called *individuals*, to a problem evolves toward better solutions. The evolution is governed by so-called genetic operators such as mutation and crossover that select and modify individuals to form a new population. In general, a fitter individual has a better chance to survive and prevail in a population.

In this paper we formulate the problem of generating pairwise test sets as a search problem and apply genetic algorithms to solve it. We describe (a) how a candidate solution—a set of test cases—is encoded as an individual of a population, (b) how the fitness of an individual is calculated to measure the individual's potential, and (c) how the various genetic operators are defined. The description focuses on domain-specific features of our genetic algorithm, i.e., generating pairwise test sets. We also describe tool support called PWiseGen that could serve as a framework for generating pairwise test sets using genetic algorithms. The tool is configurable, extensible, and reusable, and thus will facilitate experimenting with genetic algorithms. It lets one tune the various parameters of genetic algorithms to find the best configuration, or to develop a new algorithm, for a specific testing problem. To our knowledge, it is the only open-source tool available for generating pairwise test sets using genetic algorithms.

We performed a series of experiments to assess, measure, and evaluate the effectiveness of our genetic algorithm and the PWiseGen tool. For the evaluation, we used the benchmark problems available from the Pairwise Testing website [4]; the website lists many tools for generating pairwise test sets, some along with their efficiency measures given in terms of the numbers of test cases generated for the benchmark problems, however, most of these tools use some sort of deterministic algorithms or strategies. Our approach is competitive in that, compared with the existing tools of which efficiency measures are available, the PwiseGen tool showed equal or better efficiencies on all benchmark problems except for one. Our experiments also showed the effectiveness of domain-specific heuristics such as fitness calculation and genetic operations that we introduced to our genetic algorithm.

In the remainder of this section, we provide a quick overview of pairwise testing and genetic algorithms. Section II describes the problem of generating pairwise testing sets. Section III explains our genetic algorithm for generating pairwise test sets by focusing on its key elements such as encoding of individuals, fitness calculation, and genetic operations. Section IV describes our tool support, the PWiseGen framework. Section-V describes experiments that we performed to evaluate the effectiveness of our genetic algorithm. Section VI discusses related work, and Section VII concludes this paper.

A. Pairwise Testing

Pairwise testing is a combinatorial testing technique in which every pair of input parameters of software is tested [4] [1]. It is regarded as a reasonable cost-benefit compromise

among combinatorial testing methods; it can be performed much faster than exhaustive testing that tests all combinations of all input parameters, and is more effective than less exhaustive methods that fail to exercise all possible pairs of input parameters. The reasoning behind pairwise testing is that the majority of software errors are caused by a single input parameter or a combination of two input parameters. Pairwise testing thus requires that each pair of input parameter values be captured at least by one test case. As an example, let us consider software that takes three input parameters, say x, y, and z. If each parameter can have three different values, then there will be 27 different pairs: (x_1, y_1), (x_1, y_2), ..., (y_3, z_3). A test case (x_1, y_3, z_2), for example, captures three of these 27 pairs: (x_1, y_3), (x_1, z_2), and (y_3, z_3). By selecting test cases judiciously, all pairs of input parameters can be exercised with a minimum number of test cases; e.g., a set of nine test cases can capture all 27 pairs of three parameters, each with three different values.

B. Genetic Algorithms

Genetic algorithms use biological models to emulate the process of evolution, where a population is made of a set of possible solutions called *individuals* [3]. The search starts with an initial population of which individuals are typically generated randomly. The population is evolved into a new generation by applying operations inspired by genetics and natural selection, such as selection, crossover, and mutation. This evolution process is repeated until a solution is found in the population or a certain stopping condition, e.g., the maximum number of iterations, is met. The search is guided by a fitness function that calculates the fitness values of the individuals in the population in that the fitter ones have a better chance to survive and thus evolve into the next generation. The effectiveness of a genetic algorithm is thus determined in part by the quality of its fitness function.

For an algorithm to be considered to be genetic, it should at least have the following key elements.

- *Chromosome encoding.* This is a way to represent a possible solution. A chromosome consists of genes representing a feature of an individual, and the possible values for a gene are called alleles. For example, the eye color feature of a person is a gene, and the alleles for the gene could be black, brown, blue, and green. The combination of genes in a chromosome is what defines an individual's set of features, and its encoding can vary widely depending on the specific problem to be solved.
- *Fitness function.* This is a means to measure each individual's potential. It determines how good an individual is amongst all the others. The fitness value—calculated by a fitness function and associated with each individual—is the element used to determine which individuals have more opportunities to prevail in a population.
- *Genetic operations.* These are the rule for evolution, as they are applied to the individuals of a population to facilitate their evolutions. The most common genetic operations are (a) *selection* that selects individuals for reproduction, (b) *crossover* that combines the genes of two parents and generates two new children, (c) *mutation* that modifies the genes of individuals randomly, and (d) *replacement* that defines the rules of replacing existing individuals in a population with the newly created individuals.

II. The Problem

One challenge of performing pairwise testing is to find a test set consisting of the least number of test cases that covers all pairs of input parameters of the software under test. As mentioned earlier, this problem is known to be NP-complete [2], meaning that an efficient way to find an optimal solution is not known and that the time required to generate the test cases grows rapidly with increased numbers of parameters and possible values; there exist several algorithms that provide acceptable solutions, however it remains uncertain if the solutions are optimums [4]. In this paper we address this problem by formulating it as a search problem and applying genetic algorithms. The specific research problem is thus to develop a genetic algorithm capable of generating and minimizing as much as possible the number of test cases that contain all pairs of input values to a software system, in order to perform pairwise testing.

One problem of applying search-based algorithms such as genetic algorithms to pairwise testing is that there are simply too many variables or parameters to the algorithms themselves, e.g., crossover and mutation rates. These variables are often needed to be adjusted or tweaked through many experiments to find a best configuration for a given, specific testing problem. To our knowledge, there is no open-source tool available for test practitioners or researchers to find a pairwise testing set using genetic algorithms, not to mention its configurability. The second problem to be addressed in this paper is thus to develop an open-source tool that could serve as a framework for generating pairwise test sets using genetic algorithms. The tool should be configurable, extensible, and reusable so that variations of the same algorithm could be experimented, or a new algorithm could be easily developed using the framework, for a given problem.

III. Genetic Algorithms for Pairwise Testing

As explained in Section I-B, to use a genetic algorithm we first need to represent a solution—in our case, test cases—to our problem as a chromosome. The genetic algorithm then creates an initial population of solutions and applies genetic operators such as crossover and mutation to evolve the solutions in order to find the best one (see Figure 1). In general, the production of a new generation of the population consists of three steps. First, two parents are selected from the population based on the fitness values of the individuals, and then two children are produced through crossover by copying genes from parents. Second, mutation may be introduced into the reproduction process by randomly changing some of the genes forming the children chromosomes. Third, a number of immigrants may be introduced into a new generation. An

```
P = initializePopulation()
i = 0
while (i < MAX_GEN && !hasSolution(P)) do
  calculateFitness(P)
  C = ∅
  while (|C| < NUM_CROSSOVER) do
    (p1, p2) = selectParents(P)
    (c1, c2) = crossover(p1, p2);
    if (mutate?) then
       c1 = mutate(c1);
       c2 = mutate(c2);
    end
    C = C ∪ {c1, c2}
  end
  if (immigration?) then
    I = createImmigrants();
  end
  P = updatePopulation(P, C ∪ I)
  i = i + 1
end
```

Fig. 1. Genetic algorithm

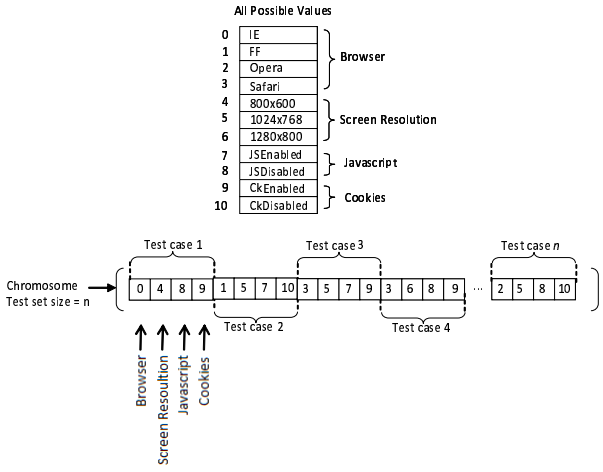


Fig. 2. Integer array chromosome encoding

immigrant is a randomly generated chromosome that replaces one of the existing individual of the population. This cycle of reproduction is repeated until either a solution is found or a predefined number of generations is produced.

There are many variables that affect the performance of genetic algorithms (see Section IV), but the three most important aspects of using genetic algorithms are: (a) definition of the genetic representation, known as chromosome encoding, (b) definition of the fitness function, and (c) definition of the genetic operations. Below we explain these key aspects in details.

A. Chromosome Encoding

We need to encode a set of test cases as an individual of a population. In the literature, there are several different encoding methods such as binary encoding, permutation encoding, value encoding, and tree encoding [3]. We decided to use an *integer array encoding* as suggested in [5], mainly because of its easiness of manipulation. This encoding essentially stores a set of test cases as an array of integer numbers, where each number identifies a possible value of an input parameter of the software under test.

As an example, let's consider a Web based system that has four different input parameters to be tested.

- Browser: Internet Explorer (IE), Firefox (FF), Opera, and Safari
- Screen resolution: 800×600, 1024×768, and 1280×800
- JavaScript: JSEnabled and JSDisabled
- Cookies: CkEnabled and CkDisabled

Figure 2 shows a sample chromosome representing a set of test cases. As shown, a test case is a sequence of numbers, where each number identifies a particular value of a certain input parameter of the software under test, e.g., 0 for the Internet Explorer and 1 for Firefox. Thus, a chromosome is simply a collection of sequences of numbers. As each sequence represents a test case, all the sequences in a chromosome have the same length, which is equal to the number of input parameters.

B. Fitness Function

A fitness function determines how good an individual is. An individual's fitness is what gives the individual the possibility to remain in a population and to be chosen for reproduction. This is one of the most important parts of a genetic algorithm because we need to have a way to measure each individual in a way that the strongest individuals are selected for producing newer and even stronger individuals to achieve an evolution in the population.

When is an individual strong and promising? As our goal is to find a (minimal) set of test cases that contains all possible pairs of input parameters, a promising individual is the one that contains many different pairs; the more different pairs an individual has, the closer it is to a solution. Based on this observation we defined the following two fitness functions.

- *Counting the number of different pairs*. This function measures an individual's fitness based on the number of different pairs included in its chromosome. This is a very straightforward fitness assessment since a larger number of different pairs means a higher possibility of capturing all pairs [5]. Note that since the total number of pairs that should be captured is known and easy to calculate, it can be easily determined if an individual is a solution. An individual is a solution if the number of different pairs contained in its chromosome equals the total number of required pairs.
- *Penalizing for repeated pairs*. This function is a variation of the above fitness function and penalizes the fitness if the same pair appears more than a certain number of times. An individual's fitness is calculated in the same way as the different pairs function, but the number of repeated or duplicate pairs is also counted to penalize the fitness.

C. Genetic Operations

Defining genetic operators is like establishing the rules of evolution. We implemented common genetic operators such as selection, crossover, replacement, immigration, and mutation. Among these, crossover and mutation have the most influence on the performance of a genetic algorithm. Before we explain these two operators in detail, we first describe the other operators briefly.

The selection operator selects parents for reproduction, and the most common method is the roulette wheel selection that assigns to each individual of the population the probability of being selected [3]. This probability is calculated based on the fitness value of an individual, and thus individuals with higher fitness values have better chances of being selected. When two parents are combined and a reproduction occurs, two new children are generated and become members of the population by substituting two existing members. For the substitution, we implemented two different replacement strategies: the weakest individual replacement and the parent complement replacement. The first strategy replaces the individual of the lowest fitness value. The second strategy adapted from [5] removes those individuals whose ranks are the complement of the parents' ranks. Given a population of 30 individuals, for example, if individuals of ranks 1 and 5, respectively, are selected for reproduction, the individuals to be replaced will be individuals of ranks 29 (complement of rank 1) and 25 (complement of rank 5). The reason is that even the least fit individual may contain valuable information and should have some measure of protection from elimination. The immigration operator is used to introduce some randomness to a population during evolution to diversify the individuals. It introduces a new individual with a randomly generated chromosome to the population by replacing an existing individual. It may avoid stagnation in the population when an improvement is not being achieved and individuals are not able to obtain more pairs by using the genetic operators.

1) Crossover: The crossover allows us to combine individuals that were selected for reproduction. The basic idea is to produce better individuals by combining the chromosomes of the selected parents. There are many different ways to perform the crossover, and we implemented four different variations.

- *Single crossover point.* In this method, a crossover at a single point is made, and the crossover point is the middle of the chromosome. This crossover point remains the same throughout the algorithm's execution.
- *Single random crossover point.* This method is used when a single crossover point at a variable position is desired. Each time a crossover is made, the position where the crossover happens is determined randomly.
- *Multiple crossover points.* In this method, the crossover is made at various points of the chromosome. The number of crossover points can vary, and depending on the number of crossover points, the crossover positions will be divided evenly over the chromosome. Once determined, the crossover positions will remain the same throughout the execution of the algorithm.
- *Multiple random crossover points.* This method also makes the crossover in several points of the chromosome, but unlike the previous, the positions where the crossover happens is always varying. Each time when two individuals are to be combined, random positions for the crossover are determined.

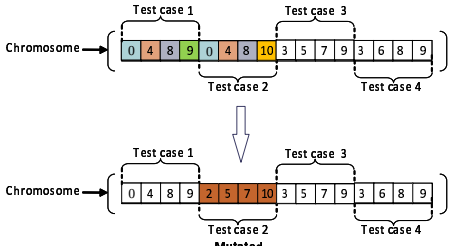


Fig. 3. Similarity mutation

2) Mutation: Mutation plays a very important role in our algorithm. We learned through experiments that without mutation individuals show improvements only in the first few generations, reaching a stagnation in the subsequent generation. We also learned that a higher mutation rate makes the genetic algorithm find a solution faster. There are studies stating that the optimal mutation rate strongly depends on the selection of the chromosome encoding and algorithms that don't use bit encodings could benefit from a higher mutation rate [6]. We implemented several variations for mutation. In random mutation, the genes to be mutated are randomly selected, and their values are replaced with randomly selected values; however, the new values must be valid for the genes. Below we explain the other mutation schemes known as *smart mutations*.

a) Similarity Mutation: The objective of this mutation is to replace a similar gene, i.e., a test case. The logic behind it is that, if an individual has two very similar test cases, it is very likely that the individual's fitness will improve if one is replaced with a different test case. This scheme requires a similarity threshold to be specified, which basically determines whether two test cases are similar or not. With a 75% similarity threshold, for example, a test case that is at least 75% similar to another is mutated. As an example, consider a chromosome, $[0|4|8|9][0|4|8|10][3|5|7|9][3|6|8|9]$, consisting of four test cases (see Figure 3). The first two test cases are similar, and thus one of them will be mutated. Depending on the configuration of the algorithm, either the new values are selected randomly or selected are those values that occur in the chromosome the least frequently.

b) Value Occurrences Mutation: This mutation attempts to replace a duplicate value of an individual with a missing value to improve the individual's fitness. For this, it tries to find a value that is not present in any of the test cases contained within the chromosome of the individual to be mutated. As an example, consider a chromosome, $[0|4|8|9][0|5|7|10][3|5|7|9][2|6|8|9]$, consisting of four test cases. The value 1, denoting Firefox, is not present in this chromosome. It can only appear in the first slot of a test

case. Therefore, the fist slot of every test case is checked for a duplication. It turns out that the value 0, denoting Internet Explorer, appears twice, i.e., in the first and second test cases, thus one of them will be replaced with the value 1.

c) Pair Occurrences Mutation: This mutation is more complex than the value occurrences mutation but has the same principle in that it attempts to increase the number of different pairs appearing in an individual. As an example, consider a chromosome $[0|4|8|9][0|5|8|10][3|5|7|9][2|6|7|10]$, consisting of four test cases. This chromosome contains the pair (0,8) twice, i.e., in the first and second test cases. It can be calculated that the pair (0,7) doesn't appear in the chromosome and can replace one of the (0,8) pairs. This example also illustrates a shortcoming of the pair occurrences mutation. If we modify the first test case to introduce a (0,7) pair, i.e., to $[0|4|7|9]$, we gain two new pairs (0,7) and (4,7) but also loose existing pairs made up of the value 8, i.e., (4, 8) and (8, 9), as well as introducing a duplicated pair (7, 9) in the first and third test cases.

IV. The PWiseGen Tool

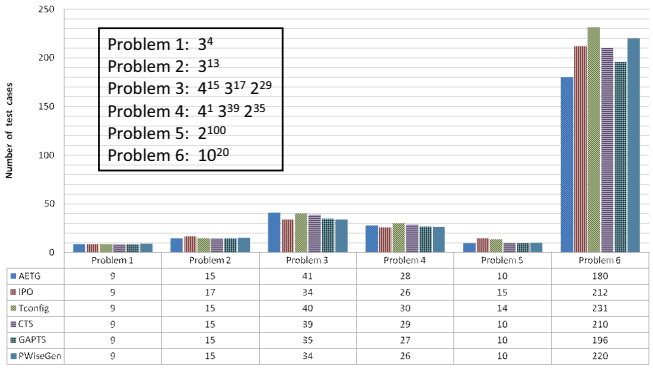
To address the problem of tool support mentioned in Section II, we developed an open-source tool that implements the genetic algorithm explained in the previous section. This tool, called PWiseGen, can also be used as a framework for applying genetic algorithms to pairwise testing. For this, we had several design goals including configurability, extensibility, and reusability. In order to achieve these design goals we heavily used object-oriented concepts such as inheritance, overriding, and polymorphism. We also used several well-known design patterns such as the strategy design pattern and the template method [7]. Another key design approach is the use of an XML-based configuration to specify various parameters of the genetic algorithm, as well as new components implementing genetic operators.

The PWiseGen implementation consists of several program modules such as initialization, fitness calculation, and evolution, and these modules are separated from the main algorithm module through well-defined interfaces. Each module comes with abstract classes and a collection of concrete classes that implement the interfaces. For example, the `Crossover` interface of the evolution module specifies the protocol for implementing the crossover genetic operator. An abstract class, `CrossoverStrategy`, provides a template for implementing a crossover strategy. It uses the template method design pattern and includes methods for determining crossover points and performing the actual crossover operation. This abstract class has several concrete subclasses that implement various crossover strategies (see Section III-C).

The use of interfaces makes the tool extensible, and the use of an XML-based configuration makes it plug-and-playable. For example, one can easily introduce a new crossover strategy by defining a new crossover class, say `MyCrossover`; it can be a subclass of the abstract class `CrossoverStrategy` or directly implement the interface `Crossover`. Once this is done, one only needs to change the XML configuration file to specify this new class as the crossover strategy (see Figure 4).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE properties (View Source for full doctype...)>
<properties version="1.0">
 <entry key="IsParamsFile">true</entry>
 <entry key="PrintEveryX">1</entry>
 <entry key="PopulationSize">30</entry>
 <entry key="TestSetSize">12</entry>
 <entry key="MaxGenerations">1000000</entry>
 <entry key="FitnessFunction">DifferentPairsFitness</entry>
 <entry key="ParentSelector">RouletteWheelSelector</entry>
 <entry key="CrossoverStrategy">MyCrossover</entry>
 ...
 <entry key="PairOccurrenceMutationValue">0.01</entry>
 <entry key="ThresholdPairOcurrences">8</entry>
</properties>
```

Fig. 4. XML-based configuration



	Problem 1	Problem 2	Problem 3	Problem 4	Problem 5	Problem 6
AETG	9	15	41	28	10	180
IPO	9	17	34	26	15	212
Tconfig	9	15	40	30	14	231
CTS	9	15	39	29	10	210
GAPTS	9	15	35	27	10	196
PWiseGen	9	15	34	26	10	220

Fig. 5. Efficiencies of pairwise testing tools. A problem of size x^y means that it takes x parameters, each with y distinct values.

V. Evaluation

We conducted a series of experiments to evaluate the effectiveness of our genetic algorithm. The Pairwise Testing website lists quite a few tools for generating pairwise test sets, some with their efficiency measures [4]. The efficiency of a tool was measured in terms of the numbers of test cases produced by the tool for several benchmark problems. The efficiencies of these tools were compared using six benchmark problems of different sizes: 3^4, 3^{13}, $4^{15}3^{17}2^{29}$, $4^{1}3^{39}2^{35}$, 2^{100}, and 10^{20}, where the notation x^y means a problem with x input parameters, each with y distinct values. Figure 5 shows the efficiency measures of these tool, along with that of PWiseGen[1]. Among these tools, only GAPTS uses a genetic algorithm [5], and the rest use some sort of deterministic algorithms; AETG is a commercial web service [1], IPO uses the in-parameter-order algorithm [2], and TConfig [8] and CTS [9] use orthogonal arrays (see Section VI). As shown, PWiseGen shows competitive results. It shows equal or better efficiencies on all the benchmark problems except for the last one. It even outperformed a genetic algorithm-based GAPTS tool on two benchmark problems; however, it was outperformed on the last problem.

[1]Unfortunately, we was not able to compare time and space efficiencies, as these measures were not available for the existing tools.

As mentioned earlier, crossover and mutation are two most important genetic operations that greatly affect the performance of a genetic algorithm. For the crossover operation, we used well-known strategies such as single and multiple crossover points. However, for the mutation operation, we introduced the notion of smart mutations that use heuristics specific for the generation of pairwise test sets, such as comparing similarity of test cases and counting duplicate values or pairs (see Section III-C). We observed that the smart mutation is on average 1.35 times faster than the random mutation in terms of the number of generations needed to find a solution. We also learned that the value occurrences mutation and the pair occurrences mutation are 1.41 and 1.43 times faster than the similarity mutation.

One practical problem of using genetic algorithms is that there are too many variables that may affect the performance of the algorithms. For example, the PWiseGen tool has about 30 different configurable parameters; some parameters such as mutation strategy have a few discrete values, but others such as mutation rate have continuous values (see Figure 4 in Section IV). The user is perplexed by the large number of parameters and their possible values, and it's a daunting task to find an optimal configuration for a given problem. To alleviate this problem, the PwiseGen tool provide a dozen of predefined configurations. We performed a series of experiments to compare these configurations using the six benchmark problems. The results are varying for different problems, but in general C3—that uses multiple crossover with 2 crossover points—is best for small number of input parameters and C6—that uses multiple random crossover with 5 crossover points—is best for large number of input parameters.

VI. Related Work

We know of few publications on applying genetic algorithms to the problem of generating pairwise test sets. Ghazi and Ahmed suggested the use of genetic algorithms to maximize pairwise coverage in testing the interaction between software components [10]. However, their work focused on proving the feasibility of using genetic algorithms without showing much details of the algorithm itself. Their experimental results didn't include efficiencies measured on the benchmark problems available from the Pairwise Testing website [4]. McCaffrey described his genetic algorithm and the GAPTS tool in general terms including the genetic operators used [5]. He also showed experimental results obtained with the benchmark problems (see Section V). However, his source code is not available, and this in fact inspired our work. McCaffrey used only standard genetic operators and didn't make use of problem-specific heuristics.

There are several deterministic algorithms capable of generating test cases for pairwise testing [4]. However, it remains uncertain if the generated test cases are optimal in terms of the number of test cases. One such a pairwise test set generation strategy is called *in-parameter-order (IPO)* that constructs a test set by considering one parameter at a time [2]. Another strategy is to use *orthogonal arrays* [8]. An orthogonal array is a two-dimensional array of numbers that have a particular value distribution—any two columns have the same number of different value pairs. The AETG system uses yet another strategy [1]. It generates a pairwise test set incrementally by starting with an empty set and adding one test case at a time Each time a new test case is needed, it produces a certain number of candidate test cases based on a greedy algorithm, and selects the one with the largest number of pairs that have not been captured yet.

VII. Conclusion

We formulated the problem of pairwise testing as a search problem and applied genetic algorithms to find pairwise test sets. We also developed a support tool called PWiseGen that could be used as a framework for generating pairwise test sets using genetic algorithms. Our approach showed competitive results compared with existing approaches and tools for pairwise testing. The key contributions of our work include (a) a genetic algorithm-based approach for generating pairwise test sets, (b) an open-source framework called PWiseGen, and (c) sample configurations and guidelines for using, adapting, and extending the framework. To our knowledge, our study is few work that applies genetic algorithms to pairwise testing, and PWiseGen is the only open-source tool for generating pairwise test sets using genetic algorithms; the tool is available from `http://code.google.com/p/pwisegen/`.

Acknowledgment

Cheon's work was supported in part by NSF grants CNS-0707874 and DUE-0837567.

References

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, Jul. 1997.

[2] Y. Lei and K. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium, November 12-14, 1998, Washington, DC*. IEEE Computer Society, 1998, pp. 254–261.

[3] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, 1999.

[4] J. Czerwonka. (2010, Dec.) Pairwise testing, combinatorial test case generation. [Online]. Available: http://www.pairwise.org

[5] J. D. McCaffrey, "An empirical study of pairwise test set generation using a genetic algorithm," in *ITNG 2010: 6th International Conference on Information Technology: New Generations, April 12-14, 2010, Las Vegas, NV*. IEEE Computer Society, 2010, pp. 992–997.

[6] D. M. Tate and A. E. Smith, "Expected allele coverage and the role of mutation in genetic algorithms," in *Proceedings of the 5th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 31–37.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[8] A. Williams and R. Probert, "A practical strategy for testing pairwise coverage of network interfaces," *Software Reliability Engineering, International Symposium on*, pp. 246–254, 1996.

[9] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149–156, Jul. 2004.

[10] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *The 2003 Congress on Evolutionary Computation, Volume 2, December 8-12, 2003, Canberra, Australia*. IEEE Computer Society, 2003, pp. 1420–1423.