Evolutionary Generation of Whole Test Suites

Authors: Gordon Fraser and Andrea Arcuri

Paper presentation by Rahul Agarwal

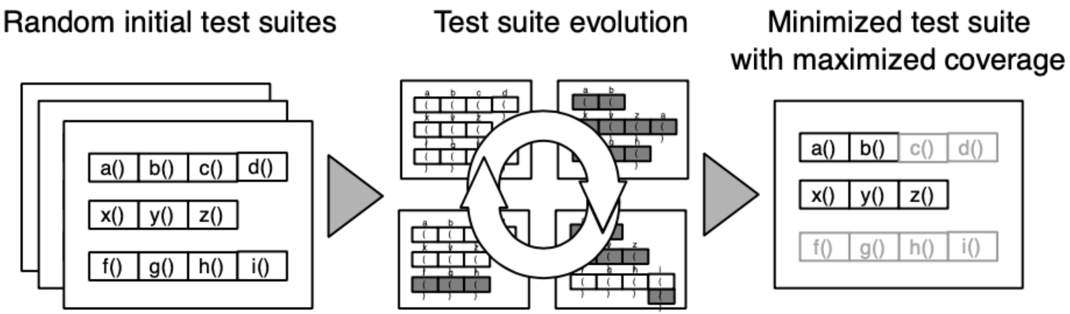
Problem

- Current test generation tools one distinct coverage goal (eg. a program branch) and derive test case.
- But, this approach assumes that all coverage goals are equally **important**, equally **difficult to reach** and **independent** of each other.
- The **order** in which goals are chosen is difficult to predict.
- Thus, the order can impact the coverage quality.

```
public class Stack {
  int[] values = new int[3];
  int size = 0;
  void push(int x) {
    if(size >= values.length)        ⇐ Requires a full stack
      resize();
    if(size < values.length)         ⇐ Else branch is infeasible
      values[size++] = x;
  }
  int pop() {
    if(size > 0)⇐ May imply coverage in push and resize
      return values[size−−];
    else
      throw new EmptyStackException();
  }
  private void resize(){
    int[] tmp = new int[values.length ∗ 2];
    for(int i = 0; i < values.length; i++)
      tmp[i] = values[i];
    values = tmp;
  }
}
```

Solution

- The paper presents a novel tool - EVOSUITE
- Rather than building distinct test cases for distinct coverage goals, EVOSUITE *optimizes the entire test suite at once towards satisfying a coverage criterion.*
- Satisfy the chosen coverage criterion with the smallest possible test suite.
- **Search based testing**
  - Handling dependencies
  - Dynamic test case length
  - Reach private functions



Approach – Test suite optimization

1. Genetic Algorithm (GA)
2. Problem Representation
3. Fitness Function
4. Bloat Control
5. Search Operators

Genetic Algorithm

1. GAs qualify as meta-heuristic search technique
2. A population of chromosomes is evolved until a solution is found that fulfills the coverage criterion
3. In each iteration, a new generation is created using **rank selection**, **crossover**, and **mutation.**

Algorithm 1 The genetic algorithm applied in EvoSuite

$current\_population \leftarrow$ generate random population
repeat
  $Z \leftarrow$ elite of $current\_population$
  while $|Z| \neq |current\_population|$ **do**
    $P_1, P_2 \leftarrow$ select two parents with rank selection
    if crossover probability **then**
      $O_1, O_2 \leftarrow$ crossover P_1, P_2
    else
      $O_1, O_2 \leftarrow P_1, P_2$
    mutate O_1 and O_2
    $f_P = min(fitness(P_1), fitness(P_2))$
    $f_O = min(fitness(O_1), fitness(O_2))$
    $l_P = length(P_1) + length(P_2)$
    $l_O = length(O_1) + length(O_2)$
    $T_B =$ best individual of $current\_population$
    if $f_O < f_P \lor (f_O = f_P \land l_O \leq l_P)$ **then**
      for O in $\{O_1, O_2\}$ **do**
        if $length(O) \leq 2 \times length(T_B)$ **then**
          $Z \leftarrow Z \cup \{O\}$
        else
          $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
    else
      $Z \leftarrow Z \cup \{P_1, P_2\}$
  $current\_population \leftarrow Z$
until solution found or maximum resources spent

Problem Representation

1. Test Suite is represented as T which consists of set of test cases t_i
2. A test case is a sequence of statements of length l $t = \langle s_1, s_2, \ldots, s_l \rangle$
3. The length of a test suite is defined as the sum of length of it's test cases length(T) = $\sum_{t \in T} l_t$
4. Each statement in a test case represents one value which belongs one of the below type:
    1. Primitive statements: numeric variables (e.g., int var0 = 54)
    2. Constructor statements: instance of a new class (e.g., Stack var1 = new Stack())
    3. Field statements: access public members of an object (e.g., int var2 = var1.size)
    4. Method statements: invoke methods on statements (int var3 = var1.pop())
5. Constraint -> $n \in [0, N]$ and $l \in [0, L]$

Fitness Function

- In order to guide the selection of parents for offspring generation, fitness function is used that rewards better coverage.
- If two test suites have the same coverage, the test suite with less statements is selected.
- In this paper, *branch coverage* is used as test criterion.

$$d(b,T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

Branch distance

$$\text{fitness}(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k,T)$$

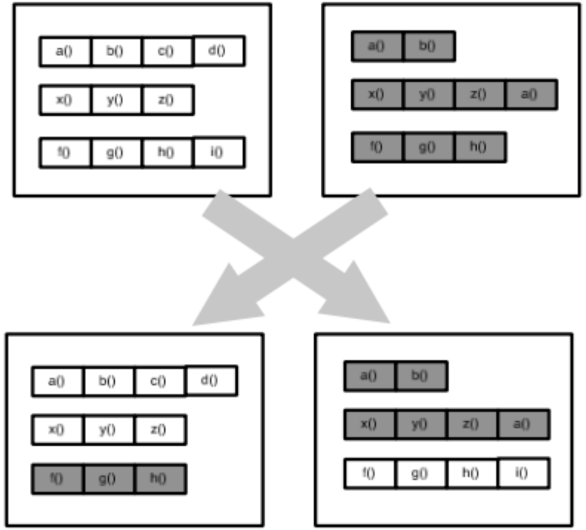
Bloat Control

- A very common problem in GA because of the variable size representation.
- After each generation, the test cases can become longer, until all memory is consumed.
- Bloat control methods employed:
    1. Limit N on max number of test cases and limit L for maximum length of each test case.
    2. Offspring with non-better coverage are never accepted in new generations

if $\boxed{f_O < f_P \vee (f_O = f_P \wedge l_O \leq l_P)}$ **then**
    for O **in** $\{O_1, O_2\}$ **do**
        if $\boxed{length(O) \leq 2 \times length(T_B)}$ **then**
            $Z \leftarrow Z \cup \{O\}$
        else
            $Z \leftarrow Z \cup \{P_1 \text{ or } P_2\}$
else
    $Z \leftarrow Z \cup \{P_1, P_2\}$

Search Operators - Crossover

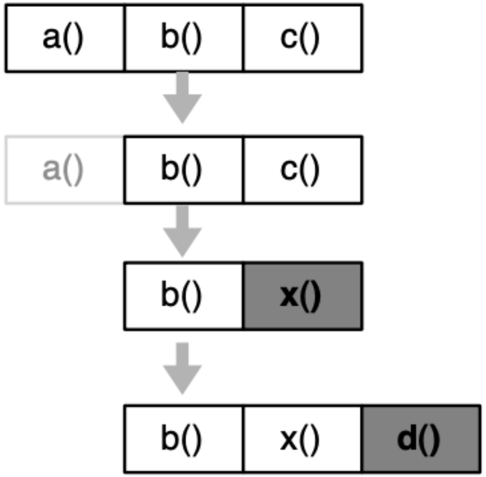
- Generates two offsprings O_1 and O_2 from parent test suites P_1 and P_2.
- O_1 will contain the first $\alpha|P_1|$ test cases followed by last $(1-\alpha)\,|P_2|$. Similarly, for O_2.
- Since the test cases are independent, the crossover yields valid offsprings.
- It also decreases the difference in the number of test cases between test suites, i.e., $abs(|O_1|-|O_2|) \leq abs(|P_1|-|P_2|)$



(a) Crossover

Search Operators – Mutation

- When a test suite T is mutated, each of it's test case is mutated with probability $1/|T|$.
- Test cases are added with a probability σ^i till the limit N.
- When a test case is mutated, then three types of operation are applied in order with probability of 1/3:
    1. **Remove**: each statement s_i is deleted with probability 1/n. If the test case needs to be repaired, then another statement is replaced of same type. If not, then s_i is deleted recursively.



(b) Mutation

Search Operator – mutation contd.

- **Change**: each statement s_i is changed with probability 1/n. If s_i is a primitive statement, then. Numeric value is changed by a random value in $\pm[0, \Delta]$. If s_i is not a primitive statement, then a method, field or constructor of same type is chosen out of the test cluster.
- **Insert**: A new statement is added at a random position in the test case with a probability σ'.

Algorithm 1 The genetic algorithm applied in EvoSuite

$current\_population \leftarrow$ generate random population
repeat
  $Z \leftarrow$ elite of $current\_population$
  while $|Z| \neq |current\_population|$ **do**
    $P_1,P_2 \leftarrow$ select two parents with rank selection
    if crossover probability **then**
      $O_1,O_2 \leftarrow$ crossover P_1,P_2
    else
      $O_1,O_2 \leftarrow P_1,P_2$
    mutate O_1 and O_2
    $f_P = min(fitness(P_1),fitness(P_2))$
    $f_O = min(fitness(O_1),fitness(O_2))$
    $l_P = length(P_1) + length(P_2)$
    $l_O = length(O_1) + length(O_2)$
    $T_B =$ best individual of $current\_population$
    if $f_O < f_P \lor (f_O = f_P \land l_O \leq l_P)$ **then**
      for O in $\{O_1,O_2\}$ **do**
        if $length(O) \leq 2 \times length(T_B)$ **then**
          $Z \leftarrow Z \cup \{O\}$
        else
          $Z \leftarrow Z \cup \{P_1 \text{ or } P_2\}$
    else
      $Z \leftarrow Z \cup \{P_1,P_2\}$
  $current\_population \leftarrow Z$
until solution found or maximum resources spent

Experiments – Implementation detail

- EVOSUITE is implemented in Java and generates Junit test suites.
- To execute the tests during the search, EVOSUITE uses Java Reflection.
- Test suites are minimized using simple minimization algorithm [2]
  - Remove each statement one at a time till remaining statements contribute to coverage
  - Reduces both the number of test cases as well as their length.
- Test case execution can be slow, and in particular when generating test cases randomly, infinite recursion can occur. Thus, a timeout of **5 seconds** is chosen for test case execution.

Experiment setup

- 5 open-source libraries and a subset of an industrial case study project.
- 727 public classes.
- Crossover probability = 3/4
- Probability of test case insertion $\sigma = 0.1$
- Probability of statement insertion σ' = 0.5
- L = 80 | N = 100
- Search is performed until 100% branch coverage or till k = 1,000,000 statements executed
- Each experiment was repeated 100 times with different seeds.

Results

- Vergha-Delaney $\hat{A}_{12}$ effect size is used to estimate the probability of EVOSUITE performing better than traditional single branch method.
- $\hat{A}_{12}$ = 0.5 means the performance of the two randomized algorithms is the same
- $\hat{A}_{12}$ = 1 means that in all 100 runs of EVOSUITE performed better than single branch strategy

Results - 1

- The coverage improvement of EVOSUITE is up to 18 times better than single branch strategy.
- “Whole test suite generation achieves **higher coverage** than single branch test case generation. “

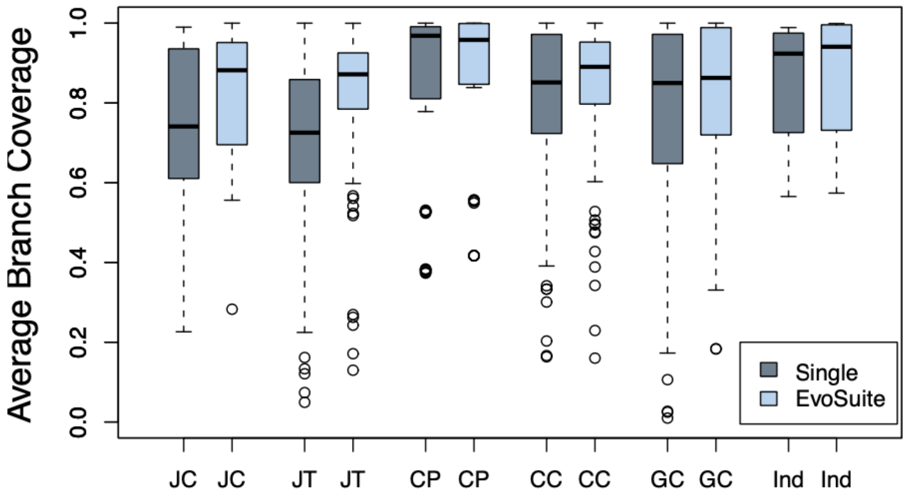


Fig. 5. Average branch coverage: Even with an evolution limit of 1,000,000 statements, EvoSuite achieves higher coverage.

Results - 2

- For cases where $\hat{A}_{12}$ = 0.5, the obtained test suite size was 44% smaller for EVOSUITE than the single branch strategy.
- “Whole test suite generation produces **smaller test suites** than single branch test case generation. “

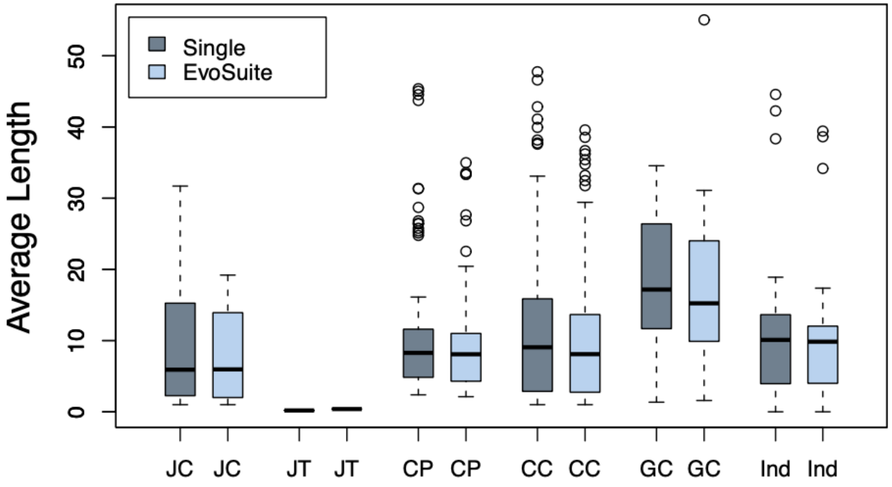


Fig. 7. Average length values: Even after minimization, EVOSUITE test suites tend to be smaller than those created with a single branch strategy (shown for cases with identical coverage).

Conclusion

- optimizing whole test suites towards a coverage criterion is superior to the traditional approach of targeting one coverage goal at a time.
- Even though branch coverage is used, other test criteria can also be utilized similarly.
- EVOSUITE can also be applied to procedural software.

Related work

1. A similar genetic algorithm is used to automatically generate unit test cases for classes.
    - This paper is being used for comparing the performance of EVOSUITE
    - They built a tool called ***eToc*** [1] for the Java language.
2. PathCrawler [2]
    1. Instead of heuristic function minimization, it used constraint logic programming

Discussion Questions

- Can EVOSUITE be applied to procedural software as well?
  - Procedural programming uses recursion
  - Flow control is performed using function calls instead of conditional statements.
- Does Automated White-Box Test Generation Really Help Software Testers? [3]
  - Once we have generated the test data, how should developers use it?
- Are the test cases generated by EVOSUITE easy to understand by the developers?
  - What if they want to repair certain test cases?
  - Or understand the test suite and manually add more test cases to further improve the coverage?

Citations

1. P. Tonella, “Evolutionary testing of classes,” in *ISSTA’04: Proceedings of the ACM International Symposium on Software Testing and Analysis*. ACM, 2004, pp. 119–128.
2. N. Williams, B. Marre, P. Mouy, and M. Roger, “PathCrawler: automatic generation of path tests by combining static and dynamic analysis,” in *EDCC’05: Proceedings ot the 5th European Dependable Computing Conference*, ser. LNCS, vol. 3463. Springer, 2005, pp. 281–292.
3. Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2013. Does automated white-box test generation really help software testers? In <i>Proceedings of the 2013 International Symposium on Software Testing and Analysis</i> (<i>ISSTA 2013</i>). Association for Computing Machinery, New York, NY, USA, 291–301. DOI:https://doi.org/10.1145/2483760.2483774

Thank You