Shrinking Counterexamples in Property-Based Testing with Genetic Algorithms

Fang-Yi Lo
National Chiao Tung University
HsinChu City, TAIWAN
fylo@nclab.tw

Chao-Hong Chen
Indiana University
Bloomington, IN, USA
chen464@indiana.edu

Ying-ping Chen*
National Chiao Tung University
HsinChu City, TAIWAN
ypchen@cs.nctu.edu.tw

***Abstract*—In this paper, genetic algorithms are proposed to shrink counterexamples found by QuickChick, a property-based testing framework for Coq. In order to make the outcome of property-based testing humanly understandable and inspectable, genetic algorithms are brought into the realm of rigorous software development as shrinkers capable of handling a broad range of data structures. In the present study, two showcases, merge sort and insertion of red-black trees, are investigated for illustrative purposes. Due to the lack of relevant results existing in the literature, two baseline methods, random sample and random walk are included in the experiments for comparison with the proposed genetic algorithm. The obtained results indicate that the proposal is effective since the program mistake can be identified with ease by examining the shrunk counterexamples and also that the adopted genetic algorithm statistically significantly outperforms random sample and random walk in both counterexample sizes and running time.**

I. Introduction

As the scale, capability, and performance of computing hardware and facilities have been greatly and rapidly enhanced for the past few decades, the complexity of software and computer programs starts to outgrow the boundary within which things are comprehensible for most programmers and software developers. Guaranteeing computer software to strictly follow its specification and ensuring it to be errors or mistakes free are highly unlikely achievable goals under such circumstances. Software testing, one of the possible countermeasures, has been investigated to address this issue for a long time. Within the field of software testing, certain techniques have been proposed and developed to respond to the urgent need in practice. Among these techniques is property-based testing, nowadays widely used for finding program errors and mistakes, which are so-called *bugs*, in software development [1], [2], [3].

The idea of property-based testing is specifying certain property of a program and using random testing to discover counterexamples, if exist. As an example, consider writing a program for a recursive function `remove` of which the inputs consist of a natural number `x` and a list of natural numbers. The designed functionality is to remove all the elements in the given list that is equal to `x`:

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | y::ys ⇒ if x =? y then ys
              else y :: remove x ys
  end.
```

A list is inductively defined in Coq with two constructors, an empty list and an element attach with a list. This fixpoint matches the input list with two constructors: (a) the input list is empty, return an empty list; (b) the input list is a natural number `y` concatenated with a list `ys`. If `x` is equal to `y`, the list `ys` is returned. If `x` is not equal to `y`, return `y` concatenated with a list formed with removing `x` from `ys`. In this case, the following property `P` must be satisfied:

```
Conjecture P : forall x l, ¬ (In x (remove x l)).
```

`In` is a function which checks whether element `x` is inside a list. By way of explanation, function `remove` should return a list containing no element equal to its first argument. However, function `remove` apparently does not satisfy property `P` because it removes only the first element of the given list equal to the first argument. With the help of random testing, a counterexample may be found as

```
[3 ; 4 ; 7 ; 10 ; 4 ; 16 ; 5 ; 9].
```

The existence of counterexamples indicates that function `remove` is incorrect, while it may be obscure why the code is incorrect and also where in the code the mistake resides from the counterexample. In order to remedy such a situation, another important aspect of property-based testing – shrinking – is brought into play.

Shrinking is a process that tries to make counterexamples as small as possible such that the root cause of an error is easy to be recognized by human. The found counterexample may be shrunk as

```
[4 ; 4].
```

By examining the shrunk counterexample, it is clear why the program errs and how it can be corrected:

```
Fixpoint remove (x : nat) (l : list nat) : list nat :=
  match l with
    | [] ⇒ []
    | y::ys ⇒ if x =? y then remove x ys
              else y :: remove x ys
  end.
```

As this example demonstrates, shrinking is crucial and indispensable for property-based testing in practice. In the present work, QuickChick [4], ported from QuickCheck [5] for Haskell, is adopted. QuickChick is a property-based testing

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

framework for Coq [6], a proof assistant based on *Calculus of Inductive Constructions* [7] currently used to prove theorems [8] and to develop verified software [9], [10], [11].

The essential benefit of using a testing framework based on Coq is that after random testing, the computer program in question may actually be proven to satisfy the specified property in logic. For instance, if counterexamples are found via a quick checking process, there is no need to make attempts to formally prove the property, usually taking a significant amount of time and effort. In the aforementioned example, we can in fact formally prove in Coq that the revised function `remove` does satisfy property `P`. After completing the proof, it can be known that no counterexample for property `P` exists. Hence, we adopt QuickChick in the hope that the outcome of this study may have a positive influence and make contributions also to the field of formal software verification, by enhancing the check mechanism for avoiding formal proof attempts on properties that cannot be satisfied.

In particular, QuickChick is utilized as the testing framework, and genetic algorithms are employed as general shrinkers, capable of handling a broad range of data types and structures, for shrinking counterexamples found by QuickChick. The obtained results demonstrate that shrunk counterexamples enable programmers to pinpoint program mistakes with ease and the adopted genetic algorithm statistically significantly outperforms the two baseline shrinkers in both resultant counterexample sizes and running time. In summary, the proposed method delivers superior performance on shrinking counterexamples in property-based testing, and this study may be considered presenting a step forward helping to build practical verified software.

For further reference, our rudimentary attempt on only integer lists was reported in [12]. The source code, counterexamples, and obtained results with statistical tests have been released on GitHub [13] for the present work.

II. The Problem

The idea of property-based testing is, for a given computer program and a specified property, to find counterexamples as evidence to disprove the property. For the lack of prior knowledge, the counterexamples are generated in a random manner. If counterexamples are found, the property is disproved. However, randomly generated counterexamples are usually too large for human to pinpoint the mistake and accordingly fix the program. Thus, counterexamples which are sufficiently small are in need. This study aims at the post-processing – *shrinking* – in property-based testing. That is, after the testing framework finds a counterexample, genetic algorithms are employed to shrink the counterexample.

In this section, as an illustration, two common problems in programming, merge sort and red-black tree insertion, are chosen as showcases to demonstrate the feasibility of utilizing genetic algorithms as shrinkers. The programs are firstly implemented with certain deliberate errors and then tested by a corresponding axiomatic property. Please note that the properties used in the showcases are merely for illustrative purposes. More criteria are needed in practice, such as that the resultant list is sorted for merge sort and that the resultant tree contains the inserted node as well as all the nodes from the original tree for red-black tree insertion.

A. Merge Sort

The first showcase is merge sort, a divide-and-conquer algorithm. The common implementation separates into two parts: (a) an unsorted list is bisected into sublists recursively; (b) sublists are merged by comparing their elements in order.

```
Fixpoint merge l1 l2 :=
  let fix merge_aux l2 :=
    match l1, l2 with
      | [], _ ⇒ l2
      | _, [] ⇒ l1
      | a1::l1', a2::l2' ⇒
        if a1 <? a2 then a1 :: merge l1' l2
        else if a2 <? a1 then a2 :: merge_aux l2'
             else a1 :: merge l1' l2'
  end
in merge_aux l2.
```

The source code shown above is the second part for merging. In line 3, two sublists are matched for three conditions. In lines 4 and 5, If either `l1` or `l2` is an empty list, the other sublist is returned. If both sublists are non-empty, the first elements of two sublists are compared, and the smaller is extracted and merged recursively. The property to test in this showcase is specified as

```
Conjecture LengthPreserved :
  forall l, length l = length (sort l).
```

The list length should remain unchanged after sorting. Here, as aforementioned, an implementation error is introduced on purpose. By using the generator and checker of QuickChick, a counterexample as a list of 748 integers [13] can be obtained as the first initial counterexample given in the supplemental material. It is obvious that although the counterexample proves the program wrong, programmers are unlikely able to extract useful information to make the program right.

B. Red-Black Tree Insertion

The other showcase is red-black tree insertion. A red-black tree is inductively defined as (a) an empty black leaf node `E`; (b) some tree `T c a k b`. `c` and `k` are the color and key value of the root, `a` and `b` are the left and right subtrees. The insertion consists of three parts:

1) `ins`: The first step for insertion is to find where to insert the new key. Since a red-black tree is also a binary search tree, three conditions may be encountered: (a) $insertkey < currentkey$ – insert into the left subtree; (b) $insertkey > currentkey$ – insert into the right subtree; (c) $insertkey = currentkey$ – do nothing.

```
Fixpoint ins x s :=
  match s with
    | E ⇒ T Red E x E
    | T c a y b ⇒
      if x <? y then balance (T c (ins x a) y b)
      else if y <? x then balance (T c a y (ins x b))
           else T c a x b
  end.
```

2) `balance`: A red-black tree is a self-balancing binary search tree. Function `ins` calls function `balance` to balance a tree from a newly inserted node up to the root. While it may seem laborious and inefficient to balance the tree by enumerating all possibilities, the implementation is done in this fashion to facilitate debugging.

```
Definition balance t :=
  match t with
    (*case 0*)
    | T Black (T Red (T Red E a E) b E) c E
    ⇒ T Black (T Red E a E) b (T Red E c E)
    | T Black (T Red E a (T Red E b E)) c E
    ⇒ T Black (T Red E a E) b (T Red E c E)
    | T Black E a (T Red (T Red E b E) c E)
    ⇒ T Black (T Red E a E) b (T Red E c E)
    | T Black E a (T Red E b (T Red E c E))
    ⇒ T Black (T Red E a E) b (T Red E c E)
    (*case 1*)
    | T Black (T Red (T Red t1 a t2) b t3) c (T Red t4 d t5)
    ⇒ T Red (T Black (T Red t1 a t2) b t3) c (T Black t4 d t5)
    | T Black (T Red t1 a (T Red t2 b t3)) c (T Red t4 d t5)
    ⇒ T Red (T Black t1 a (T Red t2 b t3)) c (T Black t4 d t5)
    | T Black (T Red t1 a t2) b (T Red (T Red t3 c t4) d t5)
    ⇒ T Red (T Black t1 a t2) b (T Black (T Red t3 c t4) d t5)
    | T Black (T Red t1 a t2) b (T Red t3 c (T Red t4 d t5))
    ⇒ T Red (T Black t1 a t2) b (T Black t3 c (T Red t4 d t5))
    (*case 2*)
    | T Black (T Red t1 a (T Red t2 b t3)) c (T Black t4 d t5)
    ⇒ T Black (T Red t1 a t2) b (T Red t3 c (T Black t4 d t5))
    | T Black (T Black t1 a t2) b (T Red (T Red t3 c t4) d t5)
    ⇒ T Black (T Red (T Red t1 a t2) b t3) c (T Red t4 d t5)
    (*case 3*)
    | T Black (T Red (T Red t1 a t2) b t3) c (T Black t4 d t5)
    ⇒ T Black (T Red t1 a t2) b (T Red t3 c (T Black t4 d t5))
    | T Black (T Black t1 a t2) b (T Red t3 c (T Red t4 d t5))
    ⇒ T Black (T Red (T Black t1 a t2) b t3) c (T Red t4 d t5)
    (*case 4*)
    | _ ⇒ t
  end.
```

3) `makeBlack`: Function `makeBlack` is defined to re-color the root if it becomes red after balancing.

```
Definition makeBlack t :=
  match t with
    | E ⇒ E
    | T _ a x b ⇒ T Black a x b
  end.
```

Then, function `insert` can be defined as

```
Definition insert x t := makeBlack (ins x t).
```

An intuitive property to test against `insert` may be specified as

```
Conjecture InsertIsBalance : forall x t,
  IsRBTree (insert x t) = true.
```

The property represents one fundamental indicator of the correctness for insertion: a red-black tree is still a red-black tree after inserting a node with key value `x`. `IsRBTree` is a checker that examines whether or not an instance satisfies the four red-black tree constraints as

```
Definition IsRBTree (t : tree) : bool :=
  (RootIsBlack t) && (NoConsecutiveRed t) &&
  (AllPathSameBlack t) && (IsBST t).
```

1) The root node is black.

```
Definition RootIsBlack (t : tree) : bool :=
  match t with
    | E ⇒ true
    | T Black l k r ⇒ true
    | _ ⇒ false
  end.
```

2) There are no consecutive red nodes, i.e., a red node cannot have red child nodes. Function `NoConsecutiveRed` recursively checks if a red node has red child nodes.

```
Fixpoint NoConsecutiveRed (t : tree) : bool :=
  match t with
    | E ⇒ true
    | T Red (T Red _ _ _) _ _
    | T Red _ _ (T Red _ _ _) ⇒ false
    | T _ l _ r
      ⇒ andb (NoConsecutiveRed l) (NoConsecutiveRed r)
  end.
```

3) Each path from the root to leaves contains the same number of black nodes. Function `HeightB` recursively calculates the height of the left and right subtrees considering only black nodes. When the left and right subtrees have different heights, `HeightB` returns `None`. Consequently, we can verify the constraint by testing whether `HeightB` returns `Some nat` or not, as function `AllPathSameBlack` does.

```
Fixpoint HeightB (t : tree) : option nat :=
  match t with
    | E ⇒ Some 1
    | T c l k r ⇒
      let lh := HeightB l in
      let rh := HeightB r in
      match optioneq lh rh with
        | false ⇒ None
        | true ⇒
          match c with
            | Black ⇒ optionplus lh (Some 1)
            | Red ⇒ lh
          end
      end
  end.
Definition AllPathSameBlack (t : tree) : bool :=
  match HeightB t with
    | Some n ⇒ true
    | None ⇒ false
  end.
```

4) A red-black tree is a binary search tree. Function `IsBST` examines if the current key is greater than all keys in the left subtree and less than all keys in the right subtree.

```
Fixpoint IsBST (t : tree) : bool :=
  match t with
    | E ⇒ true
    | T _ l k r ⇒
      match l,r with
        | T _ ll kl lr, T _ rl kr rr
          ⇒ (kl<?k)&&(k<?kr)&&(IsBST l)&&(IsBST r)
        | T _ ll kl lr, E
          ⇒ (kl<?k)&&(IsBST l)
        | E, T _ rl kr rr
          ⇒ (k<?kr)&&(IsBST r)
        | _, _ ⇒ true
      end
  end.
```

Testing this implementation against the specified property, a counterexample, composed of a red-black tree and a key value to insert, can be found by using QuickChick. For the first counterexample given in [13], after inserting 1024 into the tree consisting of 878 nodes, the resultant tree is not a valid red-black tree.

As in the case for merge sort, the counterexample found by QuickChick indicates that program mistakes exist. Nevertheless, the counterexample size prevents the program mistake from being easily identified or located by human inspection. Hence, a shrinker able to make counterexamples human readable is imperatively in need.

III. Shrinkers

As preliminaries, the applicable scope of this study is firstly defined. The proposed framework works for each data structure supporting implementation of SIZE, DELETE, SEARCH, and RANDOMDELETE satisfying the following conditions:

1) SIZE: $I \to \mathbb{N}$, where I is the set of all instances of the data structure. This function provides a way to measure the size of a given instance.
2) DELETE: $I \times K \to I$ such that

$$\forall i \in I,\ \forall k \in K,\ \mathsf{SIZE}(\mathsf{DELETE}(i,k)) \leq \mathsf{SIZE}(i)\ ,$$

where K is any suitable set. This function provides a way to create a smaller instance from the given one.
3) SEARCH: $I \to (K \to \texttt{Bool})$ such that $\forall i \in I,\ \forall k \in K$,

$$\mathsf{SEARCH}(i)(k) = \texttt{True} \text{ iff } \mathsf{SIZE}(\mathsf{DELETE}(i,k)) < \mathsf{SIZE}(i)\ .$$

And $\forall i \in I$, the inverse can be efficiently enumerated:

$$\mathsf{SEARCH}(i)^{-1}(\texttt{True}) = \{k \in K \mid \mathsf{SEARCH}(i)(k) = \texttt{True}\}$$

4) RANDOMDELETE: $I \times \mathbb{N} \to I$ is a random function such that $\forall i, i' \in I,\ \forall n \in \mathbb{N}, \exists 0 \leq m \leq n\ (i \mapsto^m i')$ iff $Pr[\mathsf{RANDOMDELETE}(i,n) = i'] > 0$, where $\mapsto^n$: $I \times I$ is a relation defined inductively as

$$i \mapsto^0 i' \text{ iff } i = i'$$
$$i \mapsto^{n+1} i' \text{ iff } \exists i'' \in I,\ \exists k \in K, \begin{pmatrix} (i \mapsto^n i'') \\ \wedge(\mathsf{SEARCH}(i'')(k)=\texttt{True}) \\ \wedge(\mathsf{DELETE}(i'',k)=i') \end{pmatrix}$$

RANDOMDELETE provides a way to randomly sample a smaller instance from the given one with the guarantee that each instance obtainable from successive DELETE operations may be an output.

For the two showcases used in this study:

1) Merge Sort: The data structure in this case is a list of natural numbers. Function SIZE simply reports the list length. Function DELETE: `list nat` $\times$ `nat` $\to$ `list nat` deletes the k-th element from list l when $\mathsf{DELETE}(l,k)$ is called. SEARCH checks whether a given number is less than the list length, i.e., $\mathsf{SEARCH}(l)(k) := k <? \mathsf{SIZE}(l)$, and the inverse image of `True` can be efficiently enumerated because it is just $\{0, \ldots, \mathsf{SIZE}(l) - 1\}$. Given these three functions, one possible implementation of RANDOMDELETE is shown in Algorithm 1.

Algorithm 1 RANDOMDELETE for integer lists

```
1: procedure RANDOMDELETE(l, n)
2:   d ← RANDOM(0, n)
3:   for i = 0 to d-1 do
4:     r ← RANDOM(0, SIZE(l) − 1)
5:     l ← DELETE(l, r)
6:   return l
```

Algorithm 2 RANDOMDELETE for red-black trees

```
1: procedure RANDOMDELETE(t, n)
2:   elements ← TRAVERSE(t)
3:   dindex ← PERMUTATION(0, SIZE(elements) − 1)
4:   d ← RANDOM(0, n)
5:   for i = 0 to d-1 do
6:     t ← DELETE(t, elements[dindex[i]])
7:   return t
```

2) Red-Black Tree: Function SIZE simply counts the number of tree nodes. Function DELETE in this case is the deletion for red-black tree as DELETE: `tree` $\times$ `nat` $\to$ `tree` which deletes the key k from tree t when $\mathsf{DELETE}(t,k)$ is called. SEARCH checks the presence of a given key, i.e., $\mathsf{SEARCH}(t)(k) := \mathsf{LOOKUP}(t,k)$, and the inverse image of `True` can be efficiently enumerated by TRAVERSE: `tree` $\to$ `list nat` returning a list contains all the keys in the tree. A possible implementation of RANDOMDELETE is shown in Algorithm 2.

As described, the constraints are actually quite general and include a wide variety of data structures. Based on the definition of the applicable scope, the three shrinking methods involved in this study are introduced in what follows.

A. Shrinker based on Random Sample

The pseudocode for the shrinker based on random sample is shown in Algorithm 3. The idea for using random sample as a shrinker is to sample a random instance smaller than the original given counterexample. As per the aforementioned definition, the sampling is simply done by using the functionality of RANDOMDELETE of the corresponding data structure. In every iteration of the while loop (from line 3 to line 9), a smaller instance is sampled directly from the original counterexample. Function ISCE at line 6 checks whether instance t is a counterexample. The if-else statement from line 8 to line 9 stores the current minimal counterexample size. $EvalMax$ is the allowed total number of instances, counterexamples or not, sampled and examined. In the present work, $EvalMax$ is set to 500 for merge sort and 50,000 for red-black tree insertion for all the three shrinkers.

B. Shrinker based on Random Walk

The shrinker based on random walk shown in Algorithm 4 is a straightforwd extension to the random sample shrinker. Instead of generating each new instance directly from the original counterexample, the random walk shrinker continues to sample next smaller instance from the current instance until a non-counterexample is sampled. Then, the shrinker restarts the procedure to sample from the original counterexample.

Algorithm 3 Random Sample Shrinker

```
1: ce ← {OriginalCounterexample}
2: mince ← SIZE(ce); eval ← 0; cecount ← 0
3: while (eval < EvalMax) do {
4:   t ← RANDOMDELETE(ce, SIZE(ce))
5:   eval ← eval + 1
6:   if ISCE(t) then
7:     cecount ← cecount + 1
8:     if SIZE(t) < mince then
9:       mince ← SIZE(t) }
```

Algorithm 4 Random Walk Shrinker

```
1: ce ← {OriginalCounterexample}
2: mince ← SIZE(ce); eval ← 0; cecount ← 0
3: while (eval < EvalMax) do {
4:   ceflag ← False; c ← ce
5:   do {
6:     t ← RANDOMDELETE(c, SIZE(c))
7:     eval ← eval + 1
8:     if ISCE(t) then
9:       cecount ← cecount + 1
10:      ceflag ← True; c ← t
11:      if SIZE(t) < mince then
12:        mince ← SIZE(t) }
13:  while (ceflag & (eval < EvalMax)) } }
```

C. Shrinker based on Genetic Algorithms

Algorithm 5 presents the pseudo code for the shrinker based on genetic algorithms proposed in this study. The overall algorithmic flow is shown, followed by the description on each major component. Please note that this study is a proof-of-principle study, and we are making an attempt to demonstrate the viability of using genetic algorithms as general shrinkers. Hence, we simply choose a reasonable, working parameter setting, instead of focusing on parameter tuning, which is surely an item of our future work.

1) Initialization: In INITIALIZATION, $psize$ instances are initially sampled from the original counterexample by using the same method previously described except for the maximum number of elements to delete is bounded by $idmax$. The other half space of $P[]$ is kept as the working space for CROSSOVER. For merge sort, $psize$ is set to 10, and $idmax$ is set to the half size of the original counterexample. For red-black tree insertion, $psize$ is set to 100, and $idmax$ is set the 1/10 size of the original counterexample.

2) Crossover: Parent candidates PA and PB are selected at random from the population for CROSSOVER. $child$ is firstly generated by making a copy of PA and then by deleting all the elements that do not exist in PB. When the elements of PA are listed and being searched in PB, a permutation is applied such that these elements may be deleted in various orders. Functions DELETE and SEARCH are defined as the operations supported by the data structure.

3) Mutation: Mutation rates are set separately depending on whether the input instance is a counterexample. One of the reasons is that if a non-counterexample selected as PB in CROSSOVER is too small in size, it might never produce a counterexample. Consequently, in order to ensure that non-counterexample instances in the population do not shrink rapidly, different mutation rates are necessary. The mutation operator employed in the present work is simply RANDOMDELETE with a bound of *mdmax*, which is set to 20 for merge sort and 5 for red-black tree insertion.

Algorithm 5 Genetic Algorithm Shrinker

```
1: ce ← {OriginalCounterexample}
2: psize ← {PopulationSize}
3: idmax ← {MaxElementsToDeleteInInit}
4: mdmax ← {MaxElementsToDeleteInMutation}
5: cemr ← {CounterexampleMutationRate}
6: ncemr ← {NonCounterexampleMutationRate}
7: survive ← {IndividualsGuaranteeToSurvive}
8: penalty ← SIZE(ce); eval ← 0
9: P[psize * 2] ← INITIALIZATION(ce, idmax)
10: while (eval < EvalMax) do {
11:   for i = psize to psize * 2 - 1 do {
12:     PA ← RANDOMSELECT(P[0], P[psize − 1])
13:     PB ← RANDOMSELECT(P[0], P[psize − 1])
14:     P[i] ← CROSSOVER(PA, PB) }
15:   for i = 0 to psize * 2 - 1 do
16:     P[i] ← MUTATION(P[i], mdmax, cemr, ncemr)
17:   P ← SELECTSURVIVOR(P, survive) }
```

4) Survivor Selection: The objective value is defined as

$$obj(t) = \text{SIZE}(t) + penalty * b \tag{1}$$

for instance t, where $penalty$ is set to the size of the original counterexample, i.e., mostly ce in the pseudo code. b is set to 0 if t is a counterexample and otherwise 1. When the objective value is computed for an individual, if the individual needs to determine whether it is a counterexample by calling ISCE, the $eval$ count will be increased by one for bookkeeping. According to the definition of objective values, an instance with a smaller objective value is considered superior. Survivor selection is then carried out by sorting the population with respect to the objective values in the increasing order and keeping the first $survive$ instances unchanged, 8 for merge sort and 40 for red-black tree insertion. The empty slots are filled by the remaining individuals selected at random.

IV. Results

In this section, the performance comparison and results indicating effectiveness will be presented. In the experiments, for both merge sort and red-black tree insertion, 50 different counterexamples found by QuickChick were shrunk with random sample, random walk, and the proposed method, respectively. The experiment on each combination of shrinkers and counterexamples was conducted for 30 independent trials for observing the performance in a statistical manner.

A. Performance Comparison on Shrinkers

Tables I and III show the average size and standard deviation of the minimal counterexamples generated by the three shrinkers over 30 independent runs on 50 different counterexamples for merge sort and red-black tree insertion,

Algorithm 6 INITIALIZATION(ce, idmax)

1: **for** i = 0 to $psize$ - 1 **do** {
2: $P[i] \leftarrow$ RANDOMDELETE$(ce, idmax)$
3: $P[i + psize] \leftarrow \emptyset$ }

Algorithm 7 CROSSOVER(PA, PB)

1: $child \leftarrow PA$
2: **for** $element \in$ SEARCH$(PA)^{-1}(True)$ **do**
3: **if** SEARCH$(PB, element) = False$ **then**
4: DELETE$(child, element)$
5: **return** *child*

respectively. The second column characterizes the initial counterexamples. The last three columns give the results of the Mann-Whitney U test. '+' indicates that the p-value < 0.005; '-' indicates otherwise. T_1, T_2, and T_3 represents tests for (RS, RW), (RS, GA), and (RW, GA), respectively. Tables II and IV show the results for running time (in milliseconds), organized similarly to that of Tables I and III.

According to Table I, random walk performs better than random sample does. Random walk is able to shrink a counterexample to nearly half of the size compared with what random sample can do, while the performance of random walk seems to deteriorate when the data structure gets more sophisticated. Table III shows that only on 15 counterexamples, random walk statistically significantly outperforms random sample. The proposed GA-based shrinker apparently outperforms random sample and random walk, used as a baseline for the lack of existing relevant results, on all items for both merge sort and red-black-tree insertion. Moreover, although the two baseline methods are able to shrink given counterexamples to a certain extent, only the proposed method can stably deliver shrunk counterexamples equal to or very close to the minimum counterexample.

B. Merge Sort

While the shrinking performance presented in Tables I, II, III, and IV indicates the effectiveness and efficiency of the GA-based shrinker, the significance has also to be demonstrated on how the shrunk counterexample can assist to identify and to correct the program mistake, which is actually the main goal of this study.

Starting from the counterexample as a list consisting of 748 integers, the minimum counterexample discovered by the proposed shrinker is an integer list of length 2:

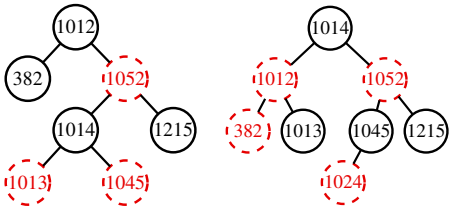
[64184177 ; 64184177].

By examining the counterexample, it is reasonable to speculate that the program mistake is the inability to handle duplicate elements. Thus, the fix can be done with ease as

```
Fixpoint merge l1 l2 :=
  ......
      else if a2 <? a1 then a2 :: merge_aux l2'
          else a1 :: a2 :: merge l1' l2'
  end
in merge_aux l2.
```

Algorithm 8 MUTATION(P[i], mdmax, cemr, ncemr)

1: $m \leftarrow$ RANDOM$(0, 100)$
2: $ceflag \leftarrow$ ISCE$(P[i])$
3: $eval \leftarrow eval + 1$
4: **if** $ceflag = True \ \& \ m < cemr$ **then**
5: **return** RANDOMDELETE$(P[i], mdmax)$
6: **if** $ceflag = False \ \& \ m < ncemr$ **then**
7: **return** RANDOMDELETE$(P[i], mdmax)$
8: **return** $P[i]$



(a) Before insertion. (b) After. Not an RBT.

Fig. 1: Minimum counterexample for red-black tree insertion generated by the proposed GA-based shrinker.

C. Red-Black Tree

For the showcase of red-black tree insertion, the proposed shrinker also successfully accomplishes the task. The minimum generated counterexample consists of a red-black tree with 7 nodes as shown in Fig. 1(a) and key value 1024 to insert. It is slightly more complicated than the case for merge sort, but manually tracing the code of function `balance` is still practical. Only case 1-3 and case 2-2 are executed during the process of insertion. As a consequence, the code can then be corrected as

```
Definition balance t :=
  match t with
    ......
    | T Black (T Black t1 a t2) b (T Red (T Red t3 c t4) d t5)
    ⇒ T Black (T Red (T Black t1 a t2) b t3) c (T Red t4 d t5)
    (*case 3*)
    ......
  end.
```

V. Conclusions

This paper proposed the use of genetic algorithms as general shrinkers capable of being easily adapted to shrink counterexamples in property-based testing for a broad range of data types and structures of which the scope was precisely defined. Merge sort and red-black tree insertion were used as showcases to demonstrate the effectiveness and efficiency of GA-based shrinkers. Two other shrinkers, random sample and random walk, were included in the study as baselines for comparison due to the lack of existing relevant results. The obtained results indicated that the performance of random walk was slightly better than that of random sample while deteriorated for more complicated data structures and also that the genetic algorithm outperformed both random sample and random walk without performance deterioration.

The successful attempt presented in this paper utilizing genetic algorithms as general shrinkers may be considered an important step forward helping to build practical verified

TABLE I: The mean and standard deviation of the minimal sizes over 30 trials of merge sort for random sample (RS), random walk (RW), and genetic algorithm (GA).

#	Length	RS	RW	GA	T_1	T_2	T_3
1	748	130.73 ± 44.51	57.77 ± 27.06	2.13 ± 0.72	+	+	+
2	944	124.43 ± 47.08	54.27 ± 29.47	2.20 ± 1.08	+	+	+
3	855	104.40 ± 36.25	48.60 ± 27.71	2.00 ± 0.00	+	+	+
4	721	96.50 ± 31.58	45.90 ± 26.67	2.00 ± 0.00	+	+	+
5	855	108.70 ± 36.18	60.10 ± 32.19	2.03 ± 0.18	+	+	+
6	880	115.13 ± 40.26	55.13 ± 34.65	2.00 ± 0.00	+	+	+
7	665	88.93 ± 34.28	40.90 ± 17.30	2.00 ± 0.00	+	+	+
8	859	121.07 ± 42.50	63.70 ± 27.86	2.00 ± 0.00	+	+	+
9	838	111.67 ± 40.89	53.53 ± 28.14	2.00 ± 0.00	+	+	+
10	719	91.67 ± 40.50	41.50 ± 20.57	16.93 ± 80.42	+	+	+
11	599	75.83 ± 22.98	40.73 ± 16.40	2.00 ± 0.00	+	+	+
12	629	110.97 ± 41.14	55.93 ± 24.48	2.03 ± 0.18	+	+	+
13	508	58.20 ± 23.07	35.27 ± 16.62	2.00 ± 0.00	+	+	+
14	858	123.70 ± 39.25	49.17 ± 29.21	28.90 ± 142.83	+	+	+
15	442	55.23 ± 19.17	27.07 ± 13.04	2.00 ± 0.00	+	+	+
16	432	69.30 ± 26.58	38.60 ± 21.54	2.00 ± 0.00	+	+	+
17	666	83.87 ± 27.37	35.57 ± 19.71	2.00 ± 0.00	+	+	+
18	966	125.77 ± 51.37	59.67 ± 26.05	2.00 ± 0.00	+	+	+
19	784	96.43 ± 31.71	54.70 ± 26.48	2.00 ± 0.00	+	+	+
20	494	76.17 ± 17.63	39.87 ± 17.89	2.23 ± 1.26	+	+	+
21	673	108.53 ± 33.61	52.87 ± 34.91	2.03 ± 0.18	+	+	+
22	794	116.83 ± 37.66	50.70 ± 22.66	2.00 ± 0.00	+	+	+
23	931	113.17 ± 40.92	62.50 ± 29.99	2.00 ± 0.00	+	+	+
24	924	121.40 ± 42.45	58.67 ± 31.96	2.03 ± 0.18	+	+	+
25	984	131.27 ± 43.97	61.23 ± 31.23	27.20 ± 135.71	+	+	+
26	833	143.20 ± 48.21	80.97 ± 38.57	2.17 ± 0.90	+	+	+
27	793	102.27 ± 32.80	47.83 ± 16.71	2.00 ± 0.00	+	+	+
28	519	78.53 ± 30.54	37.33 ± 17.84	2.07 ± 0.36	+	+	+
29	259	43.07 ± 15.34	18.07 ± 9.45	2.00 ± 0.00	+	+	+
30	858	116.00 ± 42.24	56.37 ± 25.07	2.03 ± 0.18	+	+	+
31	762	133.97 ± 50.42	66.20 ± 36.33	2.23 ± 0.88	+	+	+
32	805	104.93 ± 35.96	52.50 ± 23.50	2.00 ± 0.00	+	+	+
33	980	171.87 ± 52.00	88.43 ± 39.32	2.00 ± 0.00	+	+	+
34	705	111.90 ± 39.02	62.00 ± 31.26	2.03 ± 0.18	+	+	+
35	600	77.53 ± 28.92	39.90 ± 21.87	2.00 ± 0.00	+	+	+
36	450	64.97 ± 16.66	29.47 ± 16.66	2.00 ± 0.00	+	+	+
37	848	114.47 ± 35.14	48.63 ± 24.05	2.00 ± 0.00	+	+	+
38	685	119.23 ± 39.66	60.30 ± 23.28	2.00 ± 0.00	+	+	+
39	478	80.53 ± 30.09	41.00 ± 20.42	2.03 ± 0.18	+	+	+
40	588	72.70 ± 23.84	35.93 ± 17.86	2.07 ± 0.36	+	+	+
41	383	67.13 ± 23.33	32.73 ± 18.02	2.00 ± 0.00	+	+	+
42	945	128.07 ± 46.71	64.83 ± 36.01	2.03 ± 0.18	+	+	+
43	767	126.53 ± 49.13	57.53 ± 23.85	2.00 ± 0.00	+	+	+
44	355	48.20 ± 16.83	21.63 ± 11.44	2.07 ± 0.36	+	+	+
45	424	75.10 ± 27.00	31.30 ± 17.57	2.00 ± 0.00	+	+	+
46	874	159.77 ± 48.26	72.30 ± 31.90	2.00 ± 0.00	+	+	+
47	545	83.73 ± 28.75	49.07 ± 22.07	2.03 ± 0.18	+	+	+
48	732	122.57 ± 36.33	61.07 ± 38.30	2.07 ± 0.36	+	+	+
49	900	118.00 ± 34.24	62.27 ± 25.57	2.00 ± 0.00	+	+	+
50	881	154.17 ± 49.09	68.63 ± 40.11	2.03 ± 0.18	+	+	+

TABLE II: The mean and standard deviation of the running time in milliseconds over 30 trials of merge sort for random sample (RS), random walk (RW), and genetic algorithm (GA).

#	Length	RS	RW	GA	T_1	T_2	T_3
1	748	22.47 ± 1.09	21.07 ± 1.65	13.13 ± 0.76	+	+	+
2	944	29.63 ± 2.29	24.80 ± 1.30	17.30 ± 1.37	+	+	+
3	855	27.70 ± 2.04	23.17 ± 1.13	15.30 ± 0.97	+	+	+
4	721	23.33 ± 6.05	18.37 ± 0.66	12.90 ± 0.94	+	+	+
5	855	26.97 ± 1.40	22.83 ± 1.13	15.37 ± 1.20	+	+	+
6	880	27.40 ± 1.87	22.73 ± 0.93	15.83 ± 1.13	+	+	+
7	665	21.70 ± 7.08	16.90 ± 0.75	12.03 ± 0.95	+	+	+
8	859	26.43 ± 1.12	22.17 ± 0.73	16.20 ± 3.53	+	+	+
9	838	26.47 ± 1.65	22.13 ± 1.41	15.27 ± 1.09	+	+	+
10	719	22.87 ± 2.72	19.50 ± 1.20	12.90 ± 0.91	+	+	+
11	599	18.40 ± 0.61	15.87 ± 1.28	10.53 ± 0.72	+	+	+
12	629	20.47 ± 5.61	17.57 ± 1.12	11.10 ± 1.11	+	+	+
13	508	16.93 ± 5.07	13.23 ± 0.67	9.00 ± 0.82	+	+	+
14	858	26.90 ± 1.60	23.33 ± 1.96	15.23 ± 1.15	+	+	+
15	442	13.33 ± 1.51	11.53 ± 0.72	8.27 ± 1.15	+	+	+
16	432	13.63 ± 5.72	11.47 ± 0.96	7.77 ± 0.92	+	+	+
17	666	21.23 ± 5.86	17.60 ± 1.72	13.17 ± 3.10	+	+	+
18	966	31.80 ± 6.82	26.30 ± 1.04	17.47 ± 1.63	+	+	+
19	784	25.97 ± 7.97	20.77 ± 1.02	14.10 ± 1.19	+	+	+
20	494	15.10 ± 3.78	13.63 ± 1.08	8.47 ± 0.56	+	+	+
21	673	20.70 ± 2.13	18.87 ± 1.28	12.50 ± 1.06	+	+	+
22	794	24.40 ± 0.99	20.73 ± 0.85	13.90 ± 0.91	+	+	+
23	931	29.03 ± 1.47	25.30 ± 1.29	17.67 ± 3.31	+	+	+
24	924	31.60 ± 5.39	24.33 ± 1.27	16.67 ± 1.40	+	+	+
25	984	31.37 ± 3.56	26.23 ± 1.12	17.93 ± 1.65	+	+	+
26	833	26.23 ± 2.54	22.63 ± 1.05	14.90 ± 1.85	+	+	+
27	793	25.07 ± 6.26	20.20 ± 0.83	13.93 ± 1.00	+	+	+
28	519	16.87 ± 3.16	13.10 ± 0.75	9.17 ± 0.86	+	+	+
29	259	7.67 ± 2.05	6.97 ± 0.80	4.70 ± 0.64	-	+	+
30	858	29.33 ± 10.52	23.17 ± 3.40	15.13 ± 1.12	+	+	+
31	762	24.47 ± 1.41	21.03 ± 1.05	13.77 ± 1.41	+	+	+
32	805	25.80 ± 3.76	20.73 ± 0.96	14.23 ± 1.20	+	+	+
33	980	32.13 ± 4.08	27.70 ± 1.07	17.97 ± 1.70	+	+	+
34	705	21.63 ± 4.07	19.07 ± 1.03	12.80 ± 2.21	+	+	+
35	600	17.97 ± 0.98	16.10 ± 1.47	11.03 ± 0.80	+	+	+
36	450	14.23 ± 4.84	11.37 ± 0.66	8.13 ± 0.67	+	+	+
37	848	28.43 ± 7.52	22.37 ± 1.33	15.67 ± 1.40	+	+	+
38	685	22.70 ± 4.81	19.07 ± 1.46	12.30 ± 1.00	+	+	+
39	478	14.37 ± 0.91	13.10 ± 0.94	8.80 ± 0.79	+	+	+
40	588	18.60 ± 1.82	14.80 ± 0.87	10.70 ± 0.78	+	+	+
41	383	12.83 ± 4.78	9.83 ± 0.78	6.67 ± 0.60	+	+	+
42	945	29.50 ± 1.36	26.57 ± 6.12	17.23 ± 1.02	+	+	+
43	767	24.23 ± 2.06	21.20 ± 1.08	13.47 ± 0.81	+	+	+
44	355	10.50 ± 0.85	9.00 ± 0.68	6.70 ± 0.64	+	+	+
45	424	12.90 ± 0.94	11.33 ± 0.75	7.50 ± 0.56	+	+	+
46	874	27.67 ± 1.40	25.13 ± 1.33	15.90 ± 1.35	+	+	+
47	545	17.77 ± 2.12	16.37 ± 1.43	10.30 ± 1.10	-	+	+
48	732	23.67 ± 1.85	20.13 ± 1.09	13.37 ± 0.98	+	+	+
49	900	28.73 ± 1.53	24.17 ± 1.88	16.47 ± 1.28	+	+	+
50	881	29.03 ± 1.70	26.23 ± 2.63	16.67 ± 1.45	+	+	+

software and toward broadly incorporating methodologies in the field of evolutionary computation into the realm of rigorous software development. Along this line of research are promising directions for future work, including immediately integrating the present work into QuickChick as a general shrinker, demonstrating the feasibility and flexibility of using GA-based shrinkers on data structures of greater complexity in the short term, substituting genetic algorithms for the current counterexample searcher in QuickChick in the medium term, and finally, completing or generating formal proofs of specified properties for a given computer program by using methodologies in evolutionary computation in the long term.

ACKNOWLEDGMENTS

The work was supported in part by the Ministry of Science and Technology of Taiwan under Grant MOST 108-2221-E-009-077. The authors are grateful to the National Center for High-performance Computing for computer time and facilities.

REFERENCES

[1] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with QuickCheck!" in *Proc. of the 11th International Workshop on Automation of Software Test*, ser. AST '16. New York, NY, USA: ACM, 2016, pp. 71–77. [Online]. Available: http://doi.acm.org/10.1145/2896921.2896928

[2] B. K. Aichernig and R. Schumi, "Property-based testing with FsCheck by deriving properties from business rule models," in *Proc. of the 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2016, pp. 219–228.

[3] C. Benac Earle, L.-Å. Fredlund, and J. Hughes, "Automatic grading of programming exercises using property-based testing," in *Proc. of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '16. New York, NY, USA: ACM, 2016, pp. 47–52. [Online]. Available: http://doi.acm.org/10.1145/2899415.2899443

[4] M. Dénès, C. Hriţcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce, "QuickChick: Property-based testing for Coq," in *Coq Workshop*, 2014.

[5] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279.

TABLE III: The mean and standard deviation of the minimal sizes over 30 trials of RBT insertion for random sample (RS), random walk (RW), and genetic algorithm (GA).

#	(Size,Key)	RS	RW	GA	T_1	T_2	T_3
1	(878,1024)	255.5 ± 65.6	217.1 ± 52.8	15.5 ± 13.4	-	+	+
2	(810,1443)	257.8 ± 33.7	209.3 ± 45.0	9.6 ± 2.3	+	+	+
3	(844,1056)	243.5 ± 41.7	223.7 ± 29.7	10.8 ± 3.6	-	+	+
4	(864,1355)	239.0 ± 32.9	219.0 ± 37.1	10.3 ± 5.0	-	+	+
5	(605,1171)	191.5 ± 38.9	196.0 ± 25.8	12.8 ± 10.0	-	+	+
6	(793,1088)	279.9 ± 27.9	264.2 ± 24.8	14.3 ± 9.1	-	+	+
7	(725,683)	199.7 ± 38.1	178.9 ± 26.5	10.0 ± 2.9	-	+	+
8	(587,514)	162.2 ± 29.3	145.5 ± 19.9	9.8 ± 1.8	+	+	+
9	(372,1259)	116.1 ± 13.1	101.9 ± 14.7	8.1 ± 2.4	+	+	+
10	(562,897)	166.7 ± 25.8	160.2 ± 20.4	12.9 ± 6.1	-	+	+
11	(697,736)	205.6 ± 45.2	187.8 ± 30.6	11.0 ± 1.5	-	+	+
12	(526,593)	210.5 ± 26.5	197.1 ± 22.9	11.1 ± 1.9	-	+	+
13	(851,641)	254.5 ± 37.9	229.0 ± 40.0	10.3 ± 1.8	-	+	+
14	(840,670)	274.1 ± 36.3	260.9 ± 38.2	18.9 ± 21.2	-	+	+
15	(1132,1444)	421.8 ± 50.2	393.6 ± 49.7	22.6 ± 36.0	-	+	+
16	(680,924)	216.4 ± 46.6	188.0 ± 38.1	9.5 ± 2.3	-	+	+
17	(588,998)	201.7 ± 25.7	174.6 ± 25.5	9.9 ± 2.6	+	+	+
18	(539,1155)	155.0 ± 18.3	138.8 ± 31.7	12.1 ± 11.9	-	+	+
19	(772,707)	241.5 ± 42.7	215.9 ± 39.0	13.8 ± 8.8	-	+	+
20	(753,1381)	235.3 ± 29.6	218.4 ± 29.7	9.1 ± 4.1	-	+	+
21	(894,1368)	240.7 ± 39.2	221.8 ± 45.1	17.6 ± 23.2	-	+	+
22	(762,1333)	282.7 ± 38.8	247.3 ± 44.0	8.4 ± 2.6	+	+	+
23	(712,1294)	204.8 ± 35.1	175.5 ± 32.1	10.7 ± 5.7	+	+	+
24	(949,1277)	285.5 ± 43.4	242.9 ± 46.4	23.0 ± 35.6	+	+	+
25	(794,861)	254.1 ± 37.5	232.7 ± 41.1	9.4 ± 2.4	-	+	+
26	(997,1235)	331.4 ± 54.3	303.6 ± 49.4	10.1 ± 4.1	-	+	+
27	(712,815)	213.9 ± 25.1	200.3 ± 24.5	9.9 ± 3.0	-	+	+
28	(869,958)	211.0 ± 50.6	217.9 ± 59.0	12.5 ± 4.6	-	+	+
29	(1028,912)	294.2 ± 39.0	271.4 ± 46.9	10.4 ± 3.7	-	+	+
30	(673,779)	233.2 ± 36.9	211.7 ± 23.4	9.8 ± 2.4	-	+	+
31	(782,655)	274.1 ± 53.7	269.5 ± 40.1	8.7 ± 2.1	-	+	+
32	(772,1129)	238.8 ± 32.1	211.8 ± 25.3	9.9 ± 3.5	+	+	+
33	(422,860)	163.7 ± 16.2	150.0 ± 19.2	7.7 ± 1.2	+	+	+
34	(606,589)	167.6 ± 21.5	134.8 ± 29.6	10.2 ± 1.6	+	+	+
35	(580,1021)	191.2 ± 21.5	161.7 ± 30.0	9.8 ± 2.5	+	+	+
36	(483,927)	147.5 ± 21.3	152.0 ± 16.0	8.3 ± 1.9	-	+	+
37	(1231,940)	435.8 ± 70.8	416.8 ± 47.8	13.4 ± 13.9	-	+	+
38	(642,1356)	216.8 ± 29.0	205.8 ± 27.2	10.1 ± 3.4	-	+	+
39	(791,1514)	239.1 ± 32.0	219.1 ± 22.5	10.8 ± 3.8	+	+	+
40	(485,694)	158.1 ± 24.9	147.8 ± 23.0	8.3 ± 2.1	-	+	+
41	(380,603)	120.8 ± 19.8	114.0 ± 19.0	10.2 ± 2.4	-	+	+
42	(400,1366)	119.3 ± 17.0	103.0 ± 17.2	9.6 ± 3.3	+	+	+
43	(561,978)	184.0 ± 13.5	159.8 ± 22.1	9.7 ± 3.6	+	+	+
44	(872,1484)	247.3 ± 48.5	231.4 ± 36.7	9.9 ± 3.9	-	+	+
45	(962,831)	440.4 ± 34.1	431.1 ± 42.7	50.6 ± 55.4	-	+	+
46	(906,1026)	266.6 ± 45.0	260.0 ± 32.7	12.1 ± 6.6	-	+	+
47	(986,957)	289.2 ± 44.9	281.7 ± 46.4	41.1 ± 61.4	-	+	+
48	(574,1111)	171.5 ± 24.4	149.8 ± 27.1	8.9 ± 2.0	+	+	+
49	(716,886)	218.1 ± 46.8	228.8 ± 24.4	8.7 ± 2.0	-	+	+
50	(892,1234)	261.1 ± 38.2	240.4 ± 45.3	12.5 ± 8.4	-	+	+

TABLE IV: The mean and standard deviation of the running time in milliseconds over 30 trials of RBT insertion for random sample (RS), random walk (RW), and genetic algorithm (GA).

#	(Size,Key)	RS	RW	GA	T_1	T_2	T_3
1	(878,1024)	10377 ± 63	12003 ± 67	1102 ± 125	+	+	+
2	(810,1443)	9535 ± 70	10913 ± 110	818 ± 75	+	+	+
3	(844,1056)	10040 ± 83	11233 ± 50	879 ± 83	+	+	+
4	(864,1355)	10235 ± 59	11495 ± 76	920 ± 87	+	+	+
5	(605,1171)	7075 ± 58	8174 ± 40	787 ± 97	+	+	+
6	(793,1088)	9458 ± 166	10690 ± 68	1005 ± 157	+	+	+
7	(725,683)	8578 ± 76	9652 ± 63	736 ± 75	+	+	+
8	(587,514)	6942 ± 50	7827 ± 54	617 ± 53	+	+	+
9	(372,1259)	4266 ± 38	4890 ± 64	417 ± 38	+	+	+
10	(562,897)	6549 ± 41	7392 ± 36	633 ± 87	+	+	+
11	(697,736)	8249 ± 122	9337 ± 60	694 ± 62	+	+	+
12	(526,593)	6120 ± 46	7070 ± 41	594 ± 74	+	+	+
13	(851,641)	10098 ± 107	11447 ± 63	859 ± 72	+	+	+
14	(840,670)	9969 ± 75	11228 ± 86	1036 ± 137	+	+	+
15	(1132,1444)	13604 ± 64	15367 ± 33	1418 ± 190	+	+	+
16	(680,924)	8021 ± 48	9184 ± 36	692 ± 82	+	+	+
17	(588,998)	6901 ± 29	7878 ± 31	630 ± 58	+	+	+
18	(539,1155)	6262 ± 60	7203 ± 41	630 ± 93	+	+	+
19	(772,707)	9029 ± 61	10204 ± 27	834 ± 428	+	+	+
20	(753,1381)	8953 ± 139	10162 ± 155	822 ± 72	+	+	+
21	(894,1368)	10660 ± 110	11903 ± 46	1007 ± 145	+	+	+
22	(762,1333)	9043 ± 60	10496 ± 82	769 ± 54	+	+	+
23	(712,1294)	8346 ± 60	9430 ± 56	785 ± 79	+	+	+
24	(949,1277)	11257 ± 116	12897 ± 209	1192 ± 175	+	+	+
25	(794,861)	9512 ± 217	10784 ± 35	848 ± 71	+	+	+
26	(997,1235)	12351 ± 409	13535 ± 78	1066 ± 103	+	+	+
27	(712,815)	8354 ± 42	9390 ± 69	744 ± 66	+	+	+
28	(869,958)	10466 ± 135	12021 ± 120	1116 ± 105	+	+	+
29	(1028,912)	12673 ± 53	14243 ± 81	1108 ± 88	+	+	+
30	(673,779)	7929 ± 121	8942 ± 36	696 ± 57	+	+	+
31	(782,655)	9416 ± 312	10828 ± 146	825 ± 84	+	+	+
32	(772,1129)	9127 ± 38	10384 ± 56	820 ± 84	+	+	+
33	(422,860)	4869 ± 37	5617 ± 15	493 ± 54	+	+	+
34	(606,589)	6994 ± 28	7908 ± 87	603 ± 54	+	+	+
35	(580,1021)	6723 ± 32	7674 ± 45	564 ± 26	+	+	+
36	(483,927)	5553 ± 38	6378 ± 42	488 ± 36	+	+	+
37	(1231,940)	14893 ± 81	16904 ± 41	1323 ± 134	+	+	+
38	(642,1356)	7528 ± 32	8628 ± 58	674 ± 66	+	+	+
39	(791,1514)	9225 ± 125	10431 ± 80	833 ± 89	+	+	+
40	(485,694)	5651 ± 58	6473 ± 65	481 ± 39	+	+	+
41	(380,603)	4337 ± 30	4895 ± 21	429 ± 57	+	+	+
42	(400,1366)	4607 ± 41	5314 ± 27	422 ± 48	+	+	+
43	(561,978)	6573 ± 104	7418 ± 111	655 ± 72	+	+	+
44	(872,1484)	10617 ± 63	11985 ± 43	912 ± 69	+	+	+
45	(962,831)	11897 ± 76	13748 ± 48	1342 ± 186	+	+	+
46	(906,1026)	11073 ± 136	12339 ± 240	994 ± 141	+	+	+
47	(986,957)	12364 ± 257	13835 ± 88	1366 ± 198	+	+	+
48	(574,1111)	7038 ± 231	8283 ± 133	703 ± 83	+	+	+
49	(716,886)	9235 ± 64	10663 ± 154	832 ± 61	+	+	+
50	(892,1234)	11496 ± 249	13099 ± 112	1115 ± 141	+	+	+

[6] T. Coq Development Team, *The Coq Reference Manual, version 8.10.1*, 2019, https://coq.inria.fr/documentation.

[7] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, no. 2, pp. 95–120, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0890540188900053

[8] G. Gonthier, "Formal proof–the four-color theorem," *Notices of The American Mathematical Society*, vol. 55, no. 11, pp. 1382–1393, 2008.

[9] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *Proc. of the 33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. [Online]. Available: http://xavierleroy.org/publi/compiler-certif.pdf

[10] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proc. of the ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[11] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent os kernels," in *Proc. of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 653–669. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026928

[12] F.-Y. Lo, C.-H. Chen, and Y.-p. Chen, "Genetic algorithms as shrinkers in property-based testing," in *Proc. of 2019 ACM SIGEVO Genetic and Evolutionary Computation Conference Companion (GECCO'19 Companion)*, 2019, pp. 291–292.

[13] ——, "GitHub repo: GA.Shrinker," https://github.com/nclab/ga.shrinker.