Genetic Algorithms for Test Suite Generation

EvoSuite

Announcements

- HW2 due next Wednesday (3/5)
  - Before spring break
- Lab5 deadline extended
  - Fix is to use java version 11.0.11
  - Lab instructions updated
  - Due after spring break

Overview

- EvoSuite - an **E**volutionary algorithm to generate test **S**uites
- Multiple Objective GAs
- CodaMosa - Escaping Coverage Plateaus with LLMs

Genetic Algorithms
in
Test Suite Generation

EvoSuite

Genetic Algorithms for Test Suite Generation

To adapt GA to any task we need to define the following:

1. What is a chromosome / individual in the population?
   a. A test case!
2. How is fitness evaluated?
   a. Coverage!
3. How are individuals selected for breeding?
   a. “Rank Selection”
4. What happens during crossover?
   a. Take part of test1 and part of test2
5. What happens during mutation?
   a. Remove a statement, swap out a call, replace a primitive, etc....

Initial Population

How do we generate a random initial population of test cases?

- Select a random length n
- Randomly insert n statements
- Many are invalid!
- Many are redundant

We need to do this in a more principled way... Java has rules

How did Randoop do this?

Initial Population

When inserting random statement either:

1. Insert a call to the unit under test
   a. constructor or method
   b. Parameters: re-use existing values in the test or create a new variable

2. Insert a statement which can later be used as a parameter to a call to the unit under test

Fitness

Fitness = Number of branches covered / total branches

Are some of these conflicting?

We will discuss how to properly balance multiple objectives in GA later

Rank Selection

A method for choosing individuals to be parents for the next generation where

1. the population is first ranked based on their fitness, and then
2. selection probabilities are assigned based on their rank

This means the best individuals (with the highest rank) have a higher chance of being selected, while the worst individuals have a lower chance, regardless of the exact difference in their fitness values

How is this different from roulette? Pros / Cons?

Rank Selection

1. Calculate fitness
2. Rank the population
    a. Sort the individuals based on their fitness values, assigning a rank to each individual (e.g., 1 for the best, N for the worst, where N is the population size)
3. Assign selection probabilities
    a. Based on the assigned ranks, determine the selection probability for each individual.
    - **Linear ranking:** A simple approach where the selection probability is directly proportional to the rank (best individual has the highest probability).

4. Select Parents
    a. Randomly select individuals for reproduction based on their assigned selection probabilities.

Evosuite avoids scaling and cumulative probability by approximating the rank

Crossover

How do we create an offspring of two test cases?

1. Generate a random number between 0 and 1. This is the % of statements from each test to keep. If test1 has n1 statements and test2 has n2 statements
   a. `k = floor(r * n1) , j = floor(r * n2)`

2. New test1 = the first k statements from test1 and the last k statements from test2.
3. New test2 = the first j statements from test2 and the last j statements from test1.

4. Clean up test by removing unused variables

Crossover

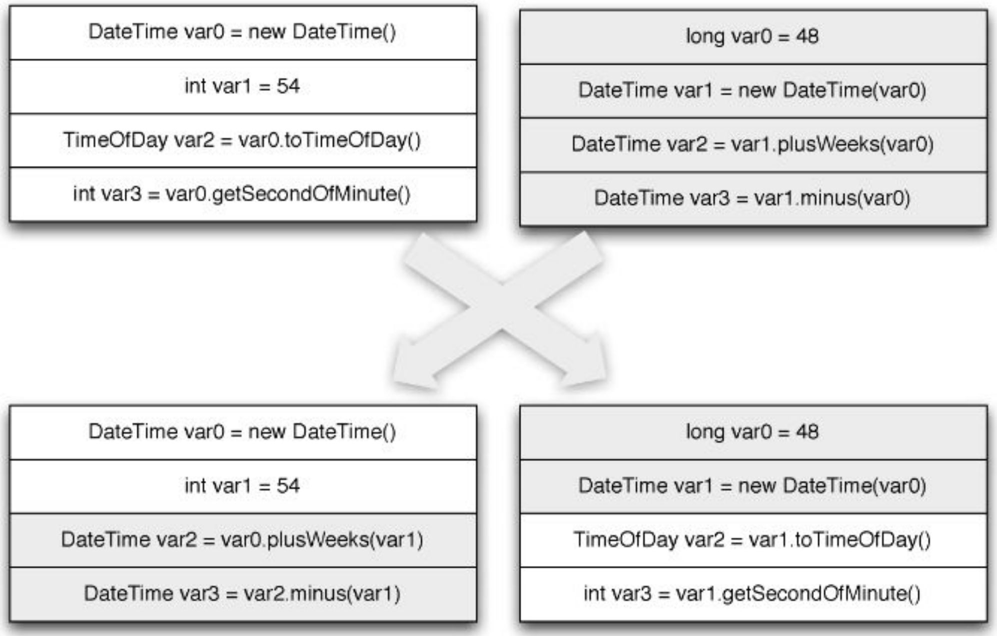

DateTime var0 = new DateTime()
int var1 = 54
TimeOfDay var2 = var0.toTimeOfDay()
int var3 = var0.getSecondOfMinute()
long var0 = 48
DateTime var1 = new DateTime(var0)
DateTime var2 = var1.plusWeeks(var0)
DateTime var3 = var1.minus(var0)
DateTime var0 = new DateTime()
int var1 = 54
DateTime var2 = var0.plusWeeks(var1)
DateTime var3 = var2.minus(var1)
long var0 = 48
DateTime var1 = new DateTime(var0)
TimeOfDay var2 = var1.toTimeOfDay()
int var3 = var1.getSecondOfMinute()

Fig. 4. Crossover between two test cases.

Mutation

1. Delete a statement
2. Insert a method call
3. Modify an existing statement
    a. Change callee (target object)
    b. Change params
    c. Change method / constructor - replace with one of the same return type
    d. Change field
    e. Change primitive

How do you think these program transformations are implemented?

Assertions

Test suites contain a prefix and assertion

The GA we defined only generates a prefix. How does it find the assertion?

EvoSuite generates ***regression*** tests.

Generating Regression Oracles

Given a unit under test `P`,

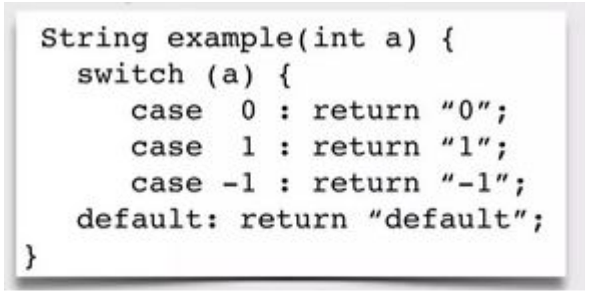
```
Execute P and record values of program variables

M = genMutants(P)

For each mutant in M:

    Execute mutant and record values of program variables

    Add assertion which kills mutant based on type of MUT
```

Multiple Objective Optimization

Genetic Algorithms with Multiple Objectives

Often times, our fitness function includes multiple objectives which may be conflicting.

- Meal Plan: Maximize nutritional value, minimize cost
- Employee Scheduling: Maximize coverage, minimize overtime
- Test Suite Generation: Maximize branch coverage



```
String example(int a) {
  switch (a) {
     case  0 : return "0";
     case  1 : return "1";
     case -1 : return "-1";
  default: return "default";
}
```

Genetic Algorithms with Multiple Objectives

Fitness functions for multiple objectives:

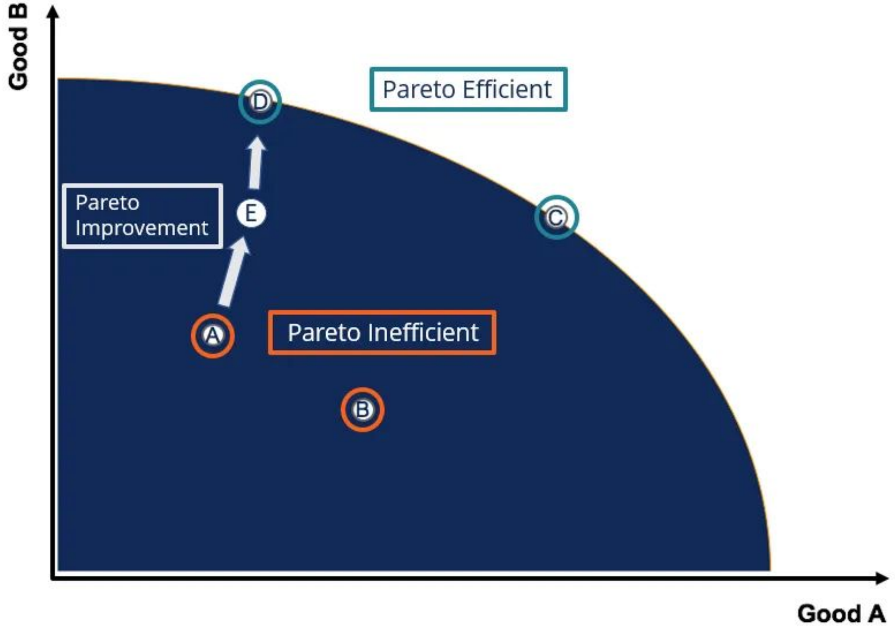
1. Weighted linear combination
   a. Nutritional Value + .5 * cost
   b. This becomes less effective with many goals
   c. Programs often have 100s of branches...

Pareto Fronts

A **Pareto improvement** formalizes the idea of an outcome being "better in every possible way".

A change is called a **Pareto improvement** if it leaves at least one person in society better-off without leaving anyone else worse off than they were before.

A situation is called **Pareto efficient** or **Pareto optimal** if all possible Pareto improvements have already been made; in other words, there are no longer any ways left to make one person better-off, without making some other person worse-off



Pareto Dominance

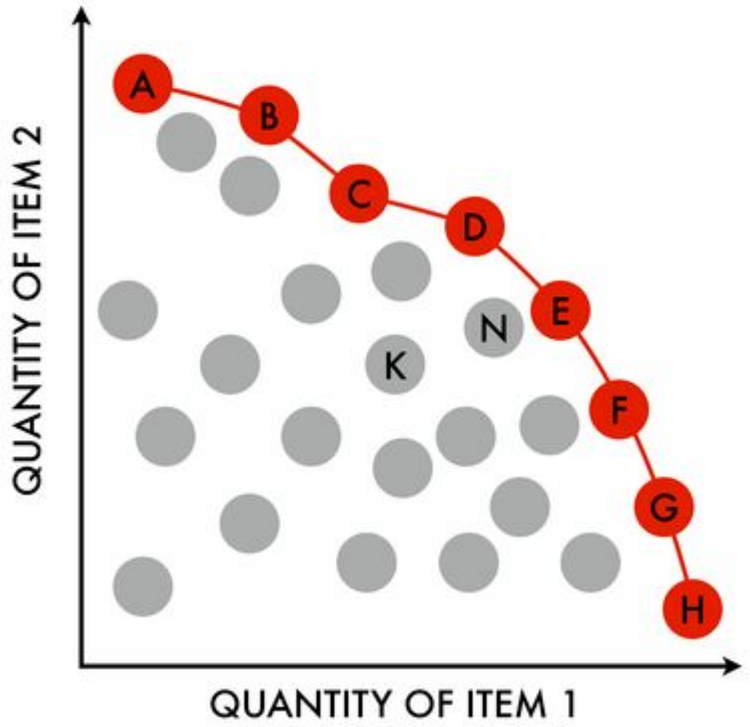
Pareto Dominance: A solution A dominates solution B if A is at least as good as B in all objectives and strictly better in at least one.

Pareto Front: The set of non-dominated solutions representing the trade-offs between conflicting objectives.

Production Possibility Frontier

The red line is an example of a Pareto-efficient frontier, where the frontier and the area left and below it are a continuous set of choices.

The red points on the frontier are examples of Pareto-optimal choices of production. Points off the frontier, such as N and K, are not Pareto-efficient, **since there exist points on the frontier which Pareto-dominate them.**

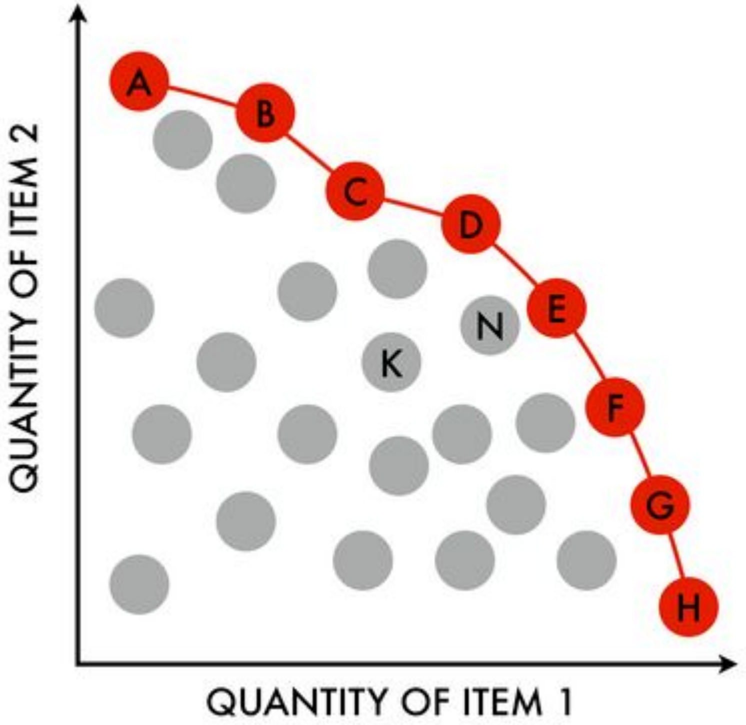


Production Possibility Frontier

Points off the frontier, such as N and K, are not Pareto-efficient, **since there exist points on the frontier which Pareto-dominate them.**

a solution A dominates solution B if A is at least as good as B in all objectives and strictly better in at least one.

What points dominate K and N?



Back to EvoSuite

EvoSuite uses these concepts in its **selection function**

Calculates pareto optimal solutions

a. Tests which are not “dominated”
b. Dominance is when one test has better fitness on some branch and the other is not better on any other branch
c. If T1 coverages branch 4F, but T2 covers branch 4T, and are the same on all other branches, neither domains the other

Branch Coverage Dominance Example

```
public void unlock(String pin) {
        //check if account holder is admin
        if (pin.equals(decode(admin))) { //B1
                locked = false;
                return;
        }
}
public boolean withdraw(double amount) {
        if (locked) return false; //B2

        if (amount > 0 && balance >= amount) { //B3
                balance -= amount;
                return true;
        }
        return false;
}
```

Dominance is when one test has better fitness on some branch and the other is not better on any other branch

Test Case	Branches Covered
T1	B2 T
T2	B1 F, B2 T
T3	B1 T, B2 F, B3 F

T2 dominates T1

T3 and T2 do not dominate eachother

Making progress on B3 would make progress on B2 F worse

EvoSuite Selection and Fitness

Last class, we simplified fitness: `total branches covered / total branches`

It’s actually sorted by dominance ! All tests on the pareto front are equal

Higher branch coverage doesn’t mean the test is necessarily better! We want our test suite to include both cases:

- Call to withdraw with locked
- Call to withdraw with unlocked

DynaMosa

We want both T2 and T3 in our test suite, but if we keep B1F as a goal, we will never make progress toward covering B3T

DynaMosa dynamically adapts which goals are included in our ranking

Add T2 to the test suite and remove B1F from our goals

DynaMosa

1. Generate initial population

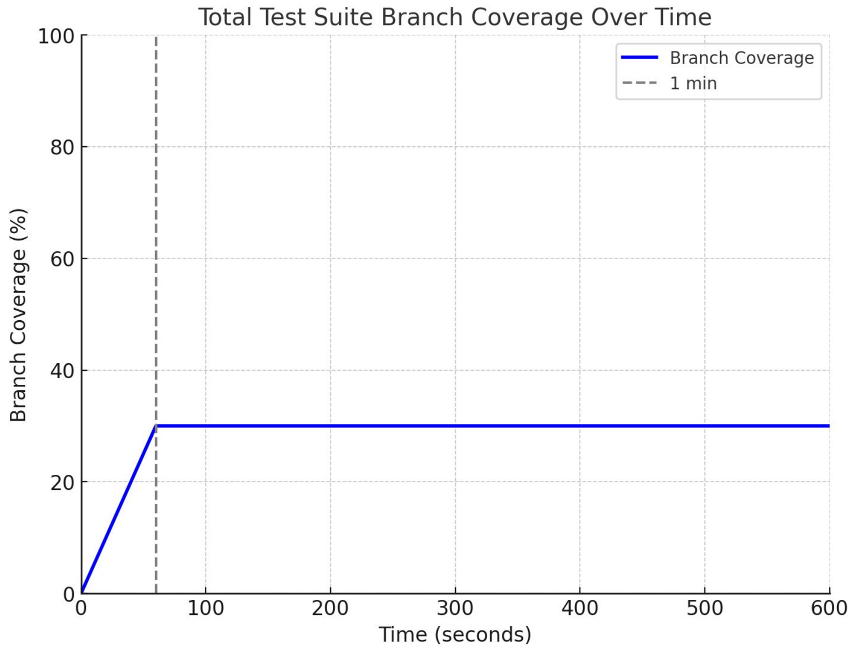
2. Until timeout:
    a. Select Parents
        i. **Requires evaluating population and ranking**
        ii. Rank based selection
    b. Perform Crossover
    c. Perform mutation with some probability
    d. **Dynamically adapt goals**

3. Output test suite

CodaMosa: Escaping Coverage Plateaus with LLMs

Coverage Plateaus

- SBST works well when mutation / crossover has a non-negligible likelihood of increasing fitness
- Imagine our unlock method...
    - What are the odds we would call unlock with the “magic pin” ?
- Fitness plateaus are common problem in test generation

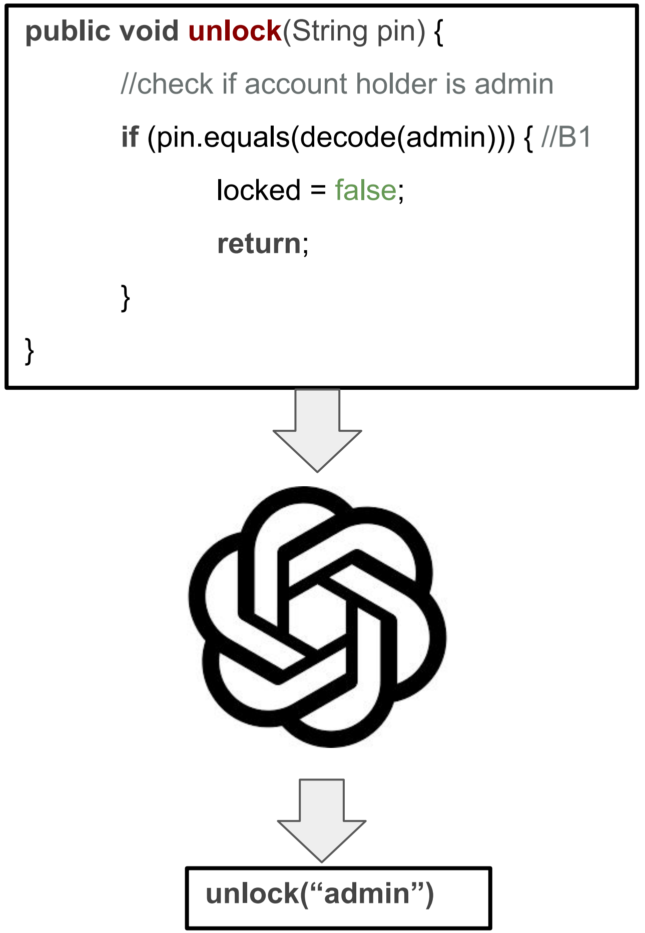


CodaMosa

CodaMOSA, starts SBST and monitors its coverage progress.

When it notices a stall in coverage, it identifies methods that have low coverage

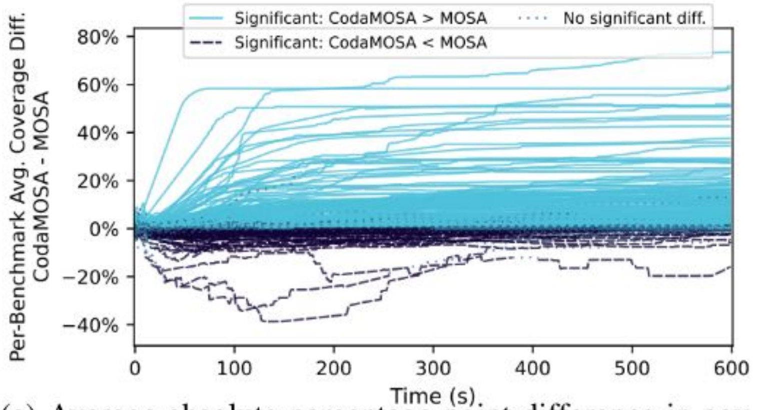
Then, it queries Codex to generate tests for these methods and adds them to the population



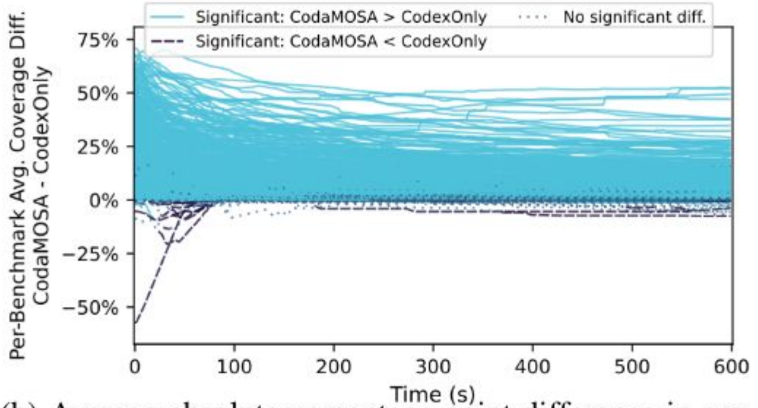
```
public void unlock(String pin) {
        //check if account holder is admin
        if (pin.equals(decode(admin))) { //B1
                locked = false;
                return;
        }
}
```

```
unlock("admin")
```

CodaMosa Evaluation



(a) Average absolute percentage point difference in coverage between CODAMOSA and MOSA.



(b) Average absolute percentage point difference in coverage between CODAMOSA and CodexOnly.

Summary

- EvoSuite adapts GA for test suite generation
  - Interesting tradeoff for branch coverage
  - We want tests which execute all cases!
  - Solves this with dominance criteria for selection and dynamic adjustment of goals
- Coverage plateaus are common!
  - Querying an LLM to add tests to the population can be beneficial
- Next week we will talk about non OOP automated test generation