

Genetic Algorithms for Test Suite Generation

EvoSuite

Announcements

- Lab4 due last night
- HW2 due next Wednesday (3/5)
 - Before spring break
- Lab5 today

Overview

- Genetic Algorithms
 - Employee scheduling problem
- EvoSuite - an **E**volutionary algorithm to generate test **S**uites

Lab 4 recap

How was Randoop?

- Did it find Chart 1?
- Did it have high coverage?
- Were the tests readable?

Search Based Software Testing

Idea: A test exists in a finite space of all possible Java tests. We just need to find the one which triggers the bug.

This requires two things

- 1) Prefix
- 2) Assertion

Genetic Algorithms

- A search and optimization technique inspired by natural selection and genetics
- Effective for complex search spaces
- NO CONVERGENCE GUARANTEES
- Termination: set number of iterations, time limit, or reached a “good enough” solution

Basic Genetic Algorithm

- Start with a large “population” of randomly generated “attempted solutions” to a problem
- Repeatedly do the following:
 - Evaluate each of the attempted solutions
 - (probabilistically) keep a subset of the best solutions
 - Use these solutions to generate a new population
- Quit when you have a satisfactory solution (or you run out of time)

Example: Employee Scheduling Problem

Imagine you're managing a small team of **3 employees**, and you need to create an optimal weekly schedule.

Each employee can either work a shift (1) or have the day off (0).

The objective is to create a schedule that balances **coverage** (you want as many people as possible working each day) and **minimizing overtime** (no one employee should work more than 4 days)

Genetic Algorithms

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

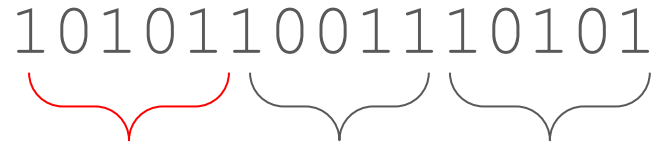
1. What is a chromosome / individual in the population?
 - a. This is a potential solution
2. How are individuals selected for breeding?
3. What happens during crossover?
4. What happens during mutation?
5. How is fitness evaluated?
 - a. How do we measure how “good” a chromosome / individual is?
 - b. Called the *fitness function*

Example: Employee Scheduling Problem

Encoding: a binary string of length 15

Each employee's schedule is represented by a binary string of length 5

1 0 1 0 1 1 0 0 1 1 1 0 1 0 1



Employee 1 works M, W, F

Employee 3 works M, W, F

Employee 2 works M, Th, F

Employee Schedule Fitness

Function that maps an individual (a potential solution) to a scalar value (fitness score).
The higher the score, the better the solution.

1. **Maximize coverage:** For each day, if at least 2 employees are scheduled, the day contributes positively to the fitness.
2. **Minimize overtime:** For each employee, penalize if they work more than 4 days.

$f(i) = \# \text{ of days with at least 2 employees on each day} - \# \text{ of employees that work more than 4 days}$

Selection

How are individuals from the population selected for reproduction?

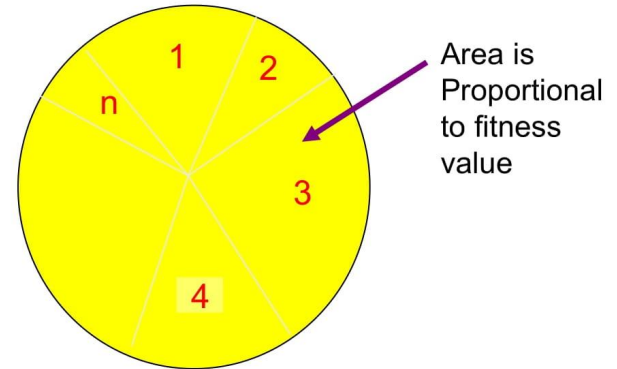
Step 1: Selection

We randomly (using a biased coin) select a subset of the individuals based on their fitness:

S1 = 110101011011011
S2 = 101100101011011
S3 = 011110011100101
S4 = 111000111010100
S5 = 000001010011001

...

Individual i will have a probability to be chosen $\frac{f(i)}{\sum_i f(i)}$



Step 1: Selection

We randomly (using a biased coin) select a subset of the individuals based on their fitness:

Individual i will have a probability to be chosen $\frac{f(i)}{\sum_i f(i)}$

S1 = 110101001011111

$f(1) = \text{mon} + \text{tues} + \text{thurs} - e3 = 2$

S2 = 100101100000001

$f(2) = \text{mon} = 1$

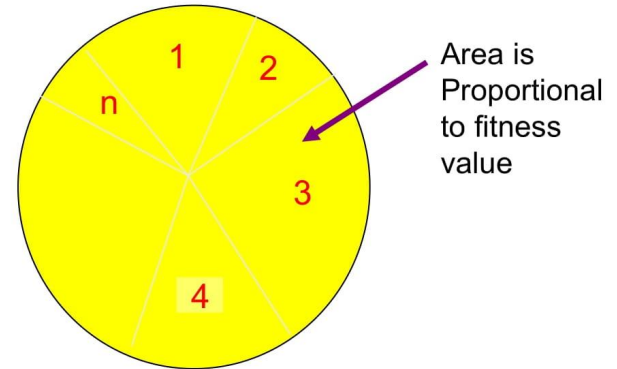
S3 = 001110101111100

$f(3) = \text{tues} + \text{wed} + \text{thurs} + \text{fri} = 4$

S4 = 111000101010000

$f(4) = \text{mon} + \text{tues} = 2$

...



This is called **roulette wheel selection**

Step 2: Crossover

Combines two parent solutions (individuals) to produce offspring solutions. The idea is to combine the good features of both parents to create one or more new solutions that may be better than the parents

Usually performed with some probability

Crossover for Employee Scheduling

Schedule1 x Schedule2 => creates two children

1. First 10 digits from Schedule1 and last 5 digits from Schedule2
 - a. Take schedule for employees A and B from schedule 1 and C from schedule
2. First 10 digits from Schedule2 and last 5 digits from Schedule1

Step 3: Mutations

Mutation introduces small random changes to candidate solutions to maintain diversity

Performed on the offspring

Performed with some probability

Mutations for Employee Scheduling

1. Randomly select an index
2. Flip bit for that index

Basic Genetic Algorithm

- Start with a large “population” of randomly generated “attempted solutions” to a problem
- Repeatedly do the following:
 - Evaluate each of the attempted solutions
 - (probabilistically) keep a subset of the best solutions
 - Use these solutions to generate a new population
- Quit when you have a satisfactory solution (or you run out of time)

GA Handout

Use parameters we just went over.

Other parameters of our algorithm:

- Initial population size of 10
- Offspring are *added* to the population
- Parents are never removed
- Population only grows
- Termination occurs after 3 iterations

Step 0: Initialization

Generate a large random “population”

S0 = 110101011001010

S1 = 101100101011011

S2 = 010110011100101

S3 = 111000111010100

S4 = 001101010111001

S5 = 110011010001111

S6 = 011101001010110

S7 = 100101111000011

S8 = 001011100110101

S9 = 111010010101101

Selection script

Fitness Proportionate Selection: Each individual's chance of being selected is proportional to its fitness. Higher fitness means a higher likelihood of being chosen.

Normalized Fitness: First, we compute normalized fitness by dividing each individual's fitness by the total fitness. This gives us relative probabilities for each individual. For example, if an individual has a fitness of 10 and the total fitness is 55, the probability of selecting that individual is $10/55$

Cumulative Probabilities: Let's say we have 4 individuals with normalized fitness values: $[0.2, 0.3, 0.4, 0.1]$

Their **cumulative probabilities** would be calculated as:
 $[0.2, 0.2+0.3=0.5, 0.5+0.4=0.9, 0.9+0.1=1.0]$

So, the cumulative probabilities are:
 $[0.2, 0.5, 0.9, 1.0]$

This means:

- Individual 1 occupies the range $[0.0, 0.2]$
- Individual 2 occupies the range $(0.2, 0.5]$
- Individual 3 occupies the range $(0.5, 0.9]$
- Individual 4 occupies the range $(0.9, 1.0]$

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 = Total number of branches / Number of branches covered

Are some of these conflicting?

We will discuss how to properly balance multiple objectives in GA next class

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 n_1 statements and test2 has n_2 statements
 - a. $k = \text{floor}(r * n_1)$, $j = \text{floor}(r * n_2)$
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

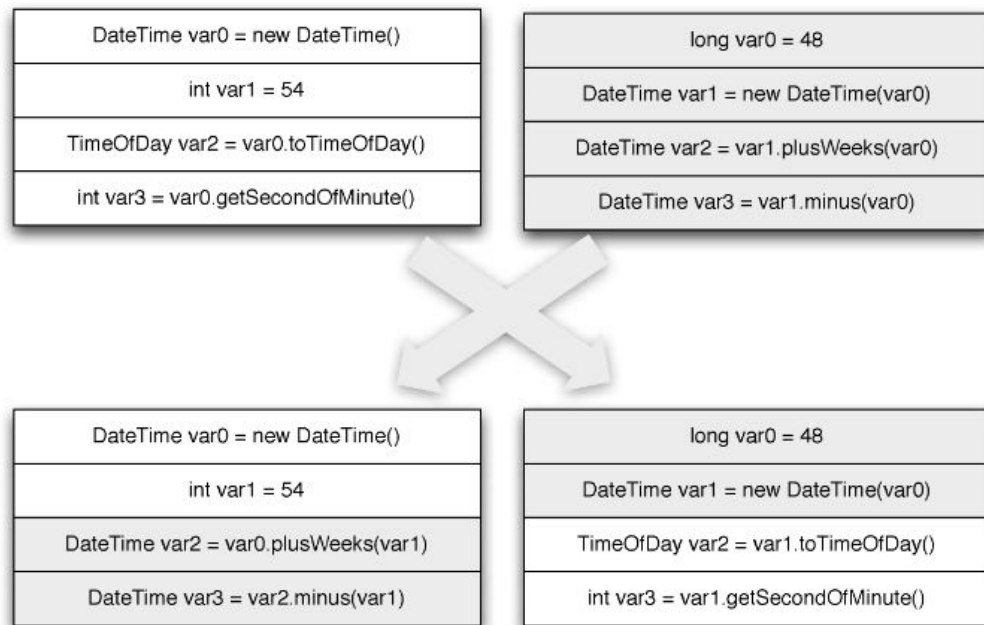


Fig. 4. Crossover between two test cases.

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

First a bunch of mutants of the original program are generated (at they bytecode level)

Then, execute each test case against the mutant and record “*traces*” with necessary information to derive the assertion

After execution, the traces are analyzed for difference between runs. An assertion is added for each difference.

Generating Regression Oracles

Assertions are generated based on the output type of the Method Under Test

Primitive Assertions: `assertEquals(var, value);`

Where `var` is the output of MUT and `value` is the observed value on the non-mutated trace

Comparison Assertions: `assertTrue/False(var2.equals(var));`

Where `var` is the output of MUT and `var2` is an observed value of the same type on the non-mutated trace.

These assertions would fail if executed on a mutant

Generating Regression Oracles

Assertions are generated based on the output type of the Method Under Test

Inspector Assertions: `assertEquals(var2, value);`

Where `var2` is the output of **an observer method** and `value` is the observed value on the non-mutated trace

Field Assertions: `assertEquals(obj.VAR, value);`

Where `VAR` is a field on the object under test and `value` is an observed value of `VAR` on the non-mutated trace.

These assertions would fail if executed on a mutant

Generating Regression Oracles

Assertions are generated based on the output type of the Method Under Test

String Assertions: `assertEquals(var.toString(), value);`

Where `var` is the output of the Method Under Test and `value` is the observed value on the non-mutated trace

Not always useful as not all classes implement `toString`

Generating Regression Oracles

Given a unit under test P ,

Execute P and record values of program variables

$M = \text{genMutants}(P)$

For each mutant in M :

 Execute mutant and record values of program variables

 Add assertion which kills mutant based on type of MUT

Summary

- Randoop generates low quality tests (both prefix and assertion)
- EvoSuite guides search based on coverage fitness value
- We traced a GA on employee scheduling
- We adapted GA to test suite generation by defining various aspects:
 - Population, selection, crossover, mutation ...