SAARLAND UNIVERSITY
COMPUTER SCIENCE

Universität des Saarlandes
Software Engineering Chair
Prof. Andreas Zeller
Gordon Fraser
David Schuler

4. Project: Search Based Testing

The task of this project is to write a search based test input generator.

1 Get the project

You can check out the project from our SVN repository. Your user name is your matriculation number, and the password was sent to you by e-mail. The location of the repository is:
`https://prog2.cs.uni-saarland.de/debugging/students/project4/<MATR>`
At the end of the project (15th July, at 23:59) the version in the repository will be graded.

For this project you will need a Java version ≥ 1.6.

2 Overview

The *InputGenerator* receives a method for which it should generate a set of inputs that achieves full *branch coverage*. The inputs are generated using a *genetic algorithm* that uses a *fitness function* that involves instrumenting the method.

3 Restrictions

The methods that are passed to the *InputGenerator* fulfill the following properties:

- The method is public and static.
- All parameters are integers.
- There is no return statement besides the last statement.
- No exceptions are thrown inside the method.
- There are no continue or break statements, or finally blocks inside the method.
- There are no endless loops.
- The only control flow affecting statements are if-statements, while, do-while, and for loops, but no for-each loops.
- All statements inside conditions are side effect free.

4 Genetic Algorithm

A genetic algorithm, as described in the lecture on *Search-based Testing*, should be used to drive the input generation. The genetic algorithm should be able to generate input such that a specific program point is reached. Thus, to achieve full branch coverage the genetic algorithm has to be run several times.

The genetic algorithm starts with a randomly generated start population, that represents different inputs. Then it evolves this generation using crossover and mutation until a specific program point or a step limit is reached.

If you want to skip the joy of implementing the genetic algorithm, you can use the genetic algorithm framework from the commons-math library.

4.1 Fitness Function

In order to asses the quality of an input, a fitness function is used. This fitness function is defined as the sum of the *approach level* and the *normalized branch distance*, which are defined below. To compute the exact value of the fitness function the code has to be instrumented and the input has to be run on the instrumented version.

4.2 Approach Level

The approach level gives the number of control dependencies between goal and chosen path. It is defined as the number of dependent nodes minus the number of executed nodes.

The number of control dependent nodes was defined in the lecture, and in more detail described in the paper *The Program Dependence Graph and Its Use in Optimization*.

Note: As there are several restrictions on the methods that are used for input generation, there is a simple way to calculate the dependent nodes from the AST. Every node in the then-part of an if-statement is control dependent on the condition evaluating to true, and vice versa every node in the else-part of an if-statement is control dependent on the condition evaluating to false. For loops, every statement in the body is control dependent on the condition evaluating to true. For nested statements, theses rules are applied transitively.

4.3 Branch Distance

The branch distance measures the distance to the critical branch being true or false, where the critical branch is the branch where control flow diverged from reaching the target. The distance metrics are defined for different constructs and are summarized in Table 1. Note that we use a value of 1 for k.

The branch distance for compound statements can be computed by combining the metrics for the different constructs. If a statement is executed multiple times, the branch distance is the minimum over all executions.

The normalized branch distance is defined as follows: If x is the branch distance the normalized branch distance is $\frac{x}{x+1}$.

Table 1: Distance metrics

Construct	Metric
$a = b$	$a - b = 0 \; ? \; 0 : abs(a - b) + k$
$a \neq b$	$a - b \neq 0 \; ? \; 0 : k$
$a < b$	$a - b < 0 \; ? \; 0 : (a - b) + k$
$a \leq b$	$a - b \leq 0 \; ? \; 0 : (a - b) + k$
$a > b$	$b - a < 0 \; ? \; 0 : (b - a) + k$
$a \geq b$	$b - a \leq 0 \; ? \; 0 : (b - a) + k$
boolean	$true ? \; 0 : k$
$a \; \&\& \; b$	$distance(a) + distance(b)$
$a \; \|\| \; b$	$min(distance(a), distance(b))$
$!a$	Move inward and propagate, e.g $!(a > b)$ becomes $a \leq b$ and $!(a \; \&\& \; b)$ becomes $!a \; \|\| \; !b$.

4.4 Chromosome

A chromosome represents an input to the method. It consists of a bit representation of the input and a fitness value, which is computed by executing the input on the instrumented version.

The bit representation of chromosome, which is used in the crossover and mutation steps, is a concatenation of the bit representation of the input values.

4.5 Population

A population consists of different chromosomes and is evolved during the run of the genetic algorithm. Here it should be taken care that no chromosomes that represent the same input are in one population.

4.6 Selection

The genetic algorithm should use a tournament selection as described in the lecture.

4.7 Crossover

A crossover between two chromosomes a and b is defined as follows: Take the bit representation of a and b, and randomly choose a point where to split them. This gives two parts for every bit representation: a_1, a_2 and b_1, b_2, where $length(a_x) = length(b_x), x \in 1, 2$. The two new chromosomes n and k are then $n = concatenate(a_1, b_2)$ and $k = concatenate(b_1, a_2)$.

4.8 Mutation

For the mutation step one random bit in the bit representation of the chromosome is flipped.

4.9 Stopping Condition

The genetic algorithm should stop when a limit of steps is reached (specified by a system property) or an input represented by a chromosome covers the target branch, e.g. its fitness is 0.

4.10 Output file

The generated test inputs are finally stored in a CSV file. Each line represents an input, different values are divided by a comma, and no blanks are allowed.

4.11 Parameters

For the genetic algorithm several parameters have to be specified.

Crossover Rate The probability at which a crossover should take place.

Mutation Rate The probability at which the a chromosome should be mutated.

Tournament Size Size of the tournament for the tournament selection.

Starting Population Size of the starting population.

Elitism Rate The percentage of chromosomes with the best fitness that are passed unchanged to the next generation.

Choose these parameters in such a way that the time needed to execute the test suite is minimized. For comparison, a run of *InputGeneratorTest* on the reference implementation takes 6 seconds. Your implementation should not take more than 1 minute on our test server.

5 Hints

- For crossover bit arithmetic might be necessary, depending on your chromosome representation. Take care that the value of the highest bit (indicating whether an int is negative) is handled correctly ($>>$ vs. $>>>$).
- When combining the different branch distance metrics, take care that the right order of operations is used, it might also be necessary to insert additional braces. For example, $a < b - 49$, results in a difference metric $a - (b - 49) < 0 \ldots$.
- Take care of overflows. For example, when $a < b$ an overflow can cause $a - b$ to be > 0.
- When using the genetic algorithm from the commons-math library, you have to take care that the population size stays constant. This can be done in the *nextGeneration()* method by taking only the x fittest chromosomes, where x is the initial size of the population.
- The commons-math algorithm optimizes towards a higher fitness. In this case you have to adapt the fitness function.

Enjoy!