CS453 CW2) Search Based Test Data Generation

20170451 Doam Lee

Introduction

This project is about finding test inputs that cover each branches. The basic form of the project is based on GA: initialize population, generate offspring, select next generation and same. But I used some techniques to improve it like utilizing special values for population initialization.

How to use

The program is tested on **Ubuntu 18.04 LTS** and **Python 2.7.17**. And it needs ast module. Some of functions may not work well on python3, so I recommend you to use python2 instead.

To execute the program, place the target file in same directory with ast_helper.py, ga_helper.py and main.py. Then execute **python main.py "file_name"**. The file must be python file that its name ends up with ".py".

It generates files named "**branch_dist_print(Num).py**" which number is the index of current function in body that has revised code, and "**br_dist**" that saves branch distances. And if there is file named same like this or the given code generates, it can be changed.

(Github address: https://github.com/ehdkacjswo/Python_test_generator)

Project overview

The whole project is consisted of three main steps

1. Revise the code to get branch fitness value
2. Find leaf branches
3. Perform GA
    - 3-1. Initialize test cases
    - 3-2. Compute branch fitness value for each leaf branches
    - 3-3. Generate offspring (crossover, mutation)
    - 3-4. Test case evaluation
    - 3-5. Select next generation

On this project, the term leaf branch means the branch that doesn't have any child branch. Then considering only leaf branches reduces computing effort, and at the same time cover parent branches too because they should be passed to reach the leaf branch.

Project component

There are three files that form the project: ast_helper.py, ga_helper.py, main.py

ast_helper.py: Helps the program to walk through the ast

find_num: Find every numbers used (except on return) in current ast. It helps to initialize and mutate the population to special values.

name_len: Find maximum length of names used in current ast. It helps to build name of variable that has been never used.

find_if: It finds out branches and add statement that writes id, operation type and branch distance.

branch: Class that administrates the information of branches (id, line number, presence of child branch, etc…).

ga_helper.py: Helps on genetic algorithm stuffs

mutate: Mutate each elements of the test. If none of them is mutated, call it again.

in_test: Checks whether the new test is in the given list of tests.

add_test: If new test is not in the given list of tests, add it to the list.

main.py: main function

gen_input: Using the special values (1, 0, -1 and values form *find_num*) and random, generate random input.

get_result: Reads the output file of function and get the branch fitness for every leaf index.

test_main: Main function controls everything

Step1: Revise the code to get branch fitness value

First of all we need three variable names, one for file variable (*ff*), one for temporary value (*tt*) and one for function name (*fff*). First one will be used for the file that will contain the branch distances, second will be used to store it, and last will be used as new name of function. *name_len* will find the maximum length of names used in given code and generate the variable names longer than that. In this way, our **new variables won't shadow or be shadowed** by other variables in given code.

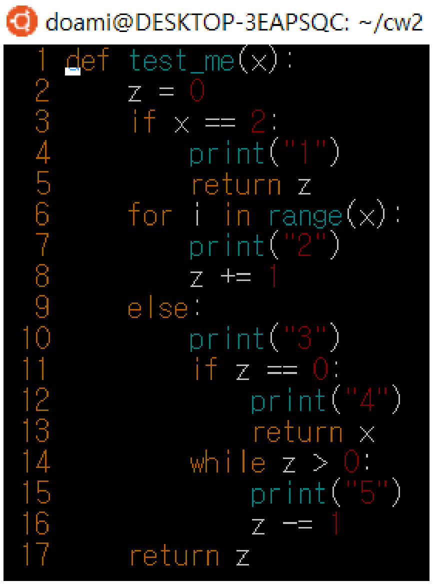
Second with given two variable names *ff* and *tt*, we have to revise the code so that the code can write branch distances. If first adds an argument *ff* to function so that the program can write on it, and analyzes and revises the code at the same time with function *find_if*. It travels down the tree and find the branches (in this case, **if and while statements**) recursively. When it finds it first update the information of branches that contains the number of parent branch, operation type (==, !=, <, <=, >, >=), existence of child branches (updated when child branch is added). And for the branches after return statement, consider them as **"unreachable" branches**. They won't be considered as child braches because they cannot be reached anyway.

After updating branch information, we add two statements that will store the branch distance value and print it. First we save the branch distance on *tt* and write the number of branch, operation type (to check the branch is passed or not) and branch distance on file *ff*. For while statement, it may have to write multiple times, so add it to the end of its body again.

After that, update the testing part of branch to use *tt* for branch testing. The reason why we store the branch distance and reuse it to branch test is because **testing twice may affect the program** (For example, updating global variable). To add these statements, we assume that every branches are given in a list. (It seems to but there may be counterexample)

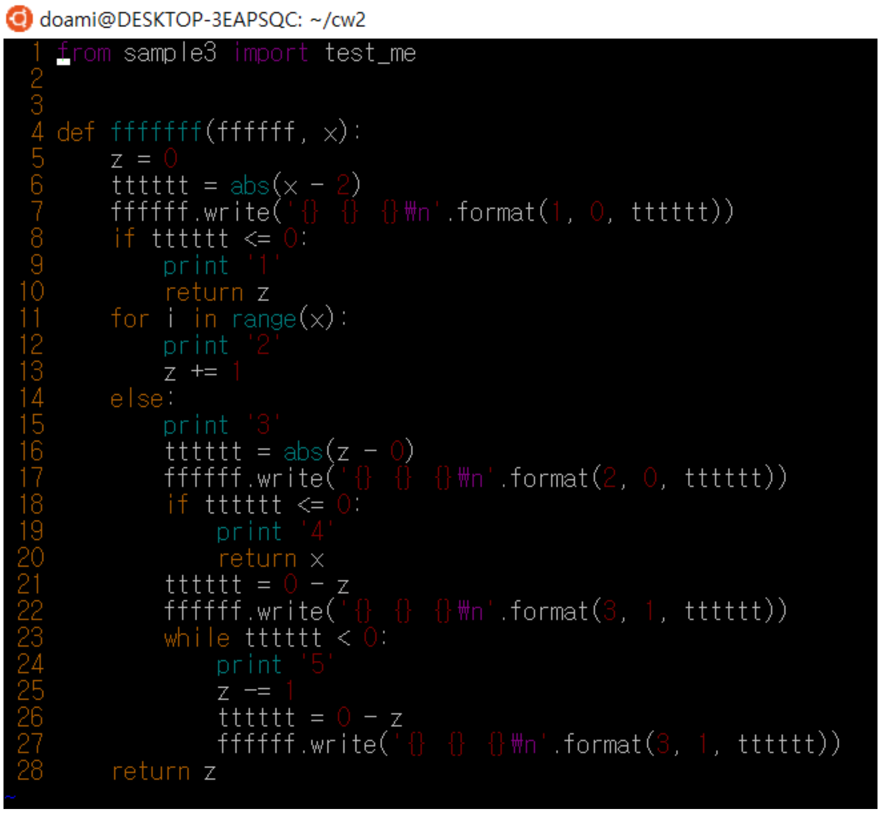
The potential problem of this technique is that the revised function has one more argument now, so when the recursive call happens, it cannot execute the original function.

And our goal is to consider branch of only “current call” (don’t consider recursive calls), so change the name of revised function and import the original one from given file. Now it can **deal with recursive functions** too, because branch distance will be printed on only the first (original) call of the function. Re-importing file with same name but different content doesn’t work well, so we name the file of supervised function differently every time that the program **can handle multiple functions**.



doami@DESKTOP-3EAPSQC: ~/cw2

```
def test_me(x):
    z = 0
    if x == 2:
        print("1")
        return z
    for i in range(x):
        print("2")
        z += 1
    else:
        print("3")
        if z == 0:
            print("4")
            return x
        while z > 0:
            print("5")
            z -= 1
    return z
```



doami@DESKTOP-3EAPSQC: ~/cw2

```
from sample3 import test_me


def fffffff(ffffff, x):
    z = 0
    tttttt = abs(x - 2)
    ffffff.write('{} {} {}₩n'.format(1, 0, tttttt))
    if tttttt <= 0:
        print '1'
        return z
    for i in range(x):
        print '2'
        z += 1
    else:
        print '3'
        tttttt = abs(z - 0)
        ffffff.write('{} {} {}₩n'.format(2, 0, tttttt))
        if tttttt <= 0:
            print '4'
            return x
        tttttt = 0 - z
        ffffff.write('{} {} {}₩n'.format(3, 1, tttttt))
        while tttttt < 0:
            print '5'
            z -= 1
            tttttt = 0 - z
            ffffff.write('{} {} {}₩n'.format(3, 1, tttttt))
    return z
~
```

Example) sample3.py

Step2: Find leaf branches

In this step, we find leaf branches and approach level of its parents. According to the branch information induced by previous section, we can find leaf branches that doesn't have child branch. Additionally, unreachable branches from step1 won't considered as leaf branch too so that the program **won't spend the time dealing with the branches that cannot be reached**. And for each of leaf indexes, the program travel through the parents and add them to the dictionary with their approach level. This information will be very helpful when calculating branch fitness value.

Step3: GA

It's the main step of the project. It's consisted of 4 main steps: population initialization, crossover, mutation and selection. I used two main strategies in this part: 1) **Remove overlapping tests**, 2) **Separated population**.

First, avoiding overlap is very important because the population converges very fast and most of tests become same. It makes the sequence slower but more stable.

Second, separated population means that each leaf branches has their own population and evolve together. It requires more computation and resource for every generations, but in my case it worked a lot faster than simple single, multiple objective optimization for complicated cases because it finds solutions faster than them.

Step3-1: Population initialization

find_num **finds the numbers used in the given code** to utilize it for population initialization. *gen_input* gets the length of input and apply random integer or one of the special values to each of the arguments. And by using *add_test*, it generates number of tests that we need without overlapping.

Step3-2: Crossover

For crossover, I used **single point crossover** and **secant method**. Single point crossover is simple: select crossover point and change the chromosomes (arguments) after it. And secant method assumes that the fitness function is linear equation of test to find test that makes the fitness value negative (-1).

To apply crossover, first select two distinct tests by binary tournament (Select two, then select better one) and then apply crossover. Mutation will be applied randomly or if the offspring is same with one of its parents.

Step3-3: Mutation

For mutation, we use special values from *gen_input* again. Following certain probability, it randomly change the test argument to one of special values or add random number based on **gamma distribution**. By adjusting the parameters of distribution, we can control exploration of mutation. If none of the arguments is mutated, do it again.

Crossover and mutation will be applied on every distinct population for leaves and the result tests will be summed up into one list of tests.

Step3-5: Evaluation

For given list of tests, we have to calculate branch fitness for each leaves. For revised code from step2 and tests, executing the code will make a file that contains branch distance of each branches visited. *get_result* function deals with it. This step is **exception safe**, because it uses try-exception. But we assume that the writing distance is done correctly. Elsewise, *get_result* won't work properly. And also, it **suppresses print of the revised code** for user's good.

First, it reads the file and get the (minimum) fitness values of all branches. If the branch is passed, save the branch distance as negative (-1). And then, using this fitness values and approach level obtained from step2, calculate branch fitness for every leaf braches. When branch distance is positive (not pass the branch), normalize it with (br_dist + 1) / (br_dist + 2) so that it can be poisitive.

If the solution for leaf branch is discovered (negative fitness value), add the solution and delete the branch from list of leaf branches so that it won't be considered for further computation. When the list becomes empty, it means that all the solutions are discovered.

When the parent branch is passed but the program doesn't visit its child (unreachable), we define the fitness for leaf as the best (minimum) approach level of its visited ancestors. So that when the leaf is not reachable, ceiling of best fitness value for it is the approach level of deepest visited ancestor. Using this, **even if the leaf is unreachable, we can find solution for ancestors that are reachable**.

Step3-4: Selection

From the result of evaluation, for each leaf branches we select best population for them. But to avoid early convergence, **leave some of original tests**.

After all these steps, it prints out the solution for each populations (not writing on the file). The branches that the solution is not found prints out '-'

Hyper parameters

Number of population (for each leaf): 100

Number of population saved for next generation: 10% of population

Number of generation: 1000

Mutation probability: 20%

Gamma distribution: alpha=1, beta=1

These parameters can be adjusted by arguments. Check them with **python main.py –h**.

Limitation and Future direction

As described on step1 part, branch may not be in the list. Then the program cannot detect that branch.

Functions with randomness may not work well, because it makes hard to reproduce the same branch result.

It can detect "syntactically" unreachable branches earlier and avoid them, but cannot detect "logically" unreachable branches (For example, if b>0: if b<0). Applying constraint solver may help.

In this project, we consider only if and while statements and integers. Considering ternary operator, for statements and float, string arguments could make it more useful.

Review

I spent a lot of time on this project testing many techniques. And the techniques that affect the performance most separated solution, special value and gamma distribution based mutation. Surprisingly, the mutation was the most helpful technique to pass sample4.py and prevent early convergence. I didn't really care much about mutation before, but now I know the importance of mutation.

And also, I ran into and solved lots of corner case that affect the performance like recursive call, or even "don't closing the file properly ends up with wrong fitness values" and "reimporting module during runtime doesn't work well". It was fun though, but very challenging. Maybe I would improve it to cover more cases later.