Automatic Test Case Generation of TSTL based on Modified Random Test Algorithm

He Zhang

zhangh7@oregonstate.edu

1 Introduction

1.1 Automatic Test Case Generation

Automatic test case generation is a method which producing test case automatically based on test requirement and test target. Rather than designing test case manually, automatic test case generation is sampler, more efficient and more directed. Based on different techniques, now there are four mainly algorithms: random test algorithms, genetic algorithms, ant colony optimization algorithms and particle swarm optimization algorithms [1].

1.2 TSTL

The Template Scripting Testing Language (TSTL) [2], is a domain-specific language for writhing test harness. TSTL provides an easy way to creating SUT.py file, which can be used by automatic test case generator.

2 Algorithms Design

2.1 Random Test Algorithm

Random test is a very popular test method. It randomly choose the input sequence (or called action sequence) through all the reasonable input space. Because it almost does not cost time on choosing next action, it is proved to be an efficient test method. Other method, as breadth-first-search method, may be not good enough to be more efficient than random test.

However, pure random test also has some disadvantage. Since it randomly choose action for each state, there is little we can do to force it to some interesting branches. Therefore, after running for some time, the coverage of random test changes little. For a deep nested branch, it is very unlikely to be covered.

2.2 Modified Random Test(MRT) with Sparse BFS algorithm

Considering the disadvantage of pure random test,

If we regard high coverage as our test goal, a better test strategy is when we find a new statement or branches when the coverage is stable, we should spend more time on searching in this new “zone”. This algorithm is based on the obvious reason: if a branch

is covered at first time, when continue digging into this test case, it is more likely to find new nested branches compared to starting a pure random test.

If we target on finding more faults, we can consider a practical software engineering rule: normally 80% defects are in 20% part of the software [3]. When find a bug somehow at the first time, test its "neighbors". These neighbors normally refer to neighbors in source code. But since it is not easy to track these neighbors, I use neighboring states and their actions instead since they also reflect the structure and logic of source code, and they are easy to get when using TSTL.

BFS is a common algorithm for searching. However, BFS is very slow and can only detect shallow depth. Sparse BFS is an improved BFS algorithm, which limits the number of nodes in a level. For instance, now we just search ten enabled actions in a state, and just add these ten following states (produced by ten relative actions) into the search queue. Before using, we need to shuffle the new queue. The improved BFS is very fast to reach the maximum depth compared to the traditional BFS. And it may test more combination cases. When the test sequence, which length is proportional to search depth, is longer, it is more likely to find a bug. So the improved BFS has a better performance.

Sparse BFS algorithm cam be described as:

```
sparseBFS(queue):
{
    Get s from queue
    Search n actions of state s
    Get new states [s1,…, sn]
    Add [s1,…, sn] to frontier
    Shuffle frontier
    Queue ← frontier
}
```

2.3 Further Improvement with Genetic Algorithm

When test timeout is long (relative to the size of software under test), modified random test with sparse BFS (s-BFS) is still not enough. One improving way is spending some time on genetic algorithm. If no new coverage changed for a setting time, it means normal paths has been covered already, and the left time need to find new branches (or new statements). So when running modified random test, the coverage information of each test case is recorded. After that, using this information to find high coverage test cases and mutate them with genetic algorithm. Same as modified random test phase, if some new branches or statements are found, start sparse BFS from this state.

Combining modified random test and genetic algorithm, the modified algorithm can be described briefly below. rtTimePara is a parameter determining a test time (time rate)

spending on modified random test. newCovTime is a valuable recording how long it lasts since last new branch or statement is covered. covTimePara is a parameter determining when to step into next test phase.

```
MainAlgorithm
{
    While time < timeout/rtTimePara or newCovTime > covTimePara:
    {
        Do random test
        If find a bug (or new branch/statement) when state is s:
            While BFS layer <= Target and time <= timeout/rtTimePara:
                Do sparseBFS test starting from s (or its neighbor)
    }
    While time <= timeout:
        Do genetic algorithm test
        If find a bug (or new branch/statement) when state is s':
            While BFS layer <= Target and time <= timeout/rtTimePara:
                Do sparseBFS test starting from s' (or its neighbor)
}
```

2.4 Strategy Selection based on Input Parameters

Some test results shows that genetic algorithm only improves tester performance when test timeout is long (more than 30 seconds for small SUT). So mytester.py contains a function. When timeout is less than 30 seconds, it will not start genetic algorithm test.

3 Test Result and Algorithm Evaluation

3.1 Test Result

Test Tools: Using avlbugs.py as SUT, avlblocks.tstl as tstl file.

Test Conditions: Timeout: 20/40/60 seconds, Seed: 46/19, Depth: 100, Width: 5

Test Target: Branch Coverage

timeout	seed	PRT	MRT-sBFS		MRT-GA	MRT-sBFS-GA	
			CT	FT		CT	FT
20	46	212	214	212	227	229	223
	19	211	211	213	228	226	227
60	46	216	216	213	229	229	223
	19	214	214	213	227	229	228
120	46	216	216	214	229	229	230
	19	216	216	215	228	229	228

Abbreviation in the form: PRT: Pure Random Test, MRT: Modified Random Test, s-BFS: sparse BFS, GA: Genetic Algorithm, CT: Coverage Trigger, FT: Fault Trigger

3.2 Algorithm Evaluation

Algorithm Efficiency Comparison

From the test result, the efficient of four algorithm is:

MRT-sBFS-GA = MRT-GA > MRT-sBFS = PRT

sBFS does not improve the coverage significantly. The reason may be because sBFS is not an efficiency algorithm, even when it combines with other algorithm (PRT and GA). It is also possible that parameter for sBFS, as width, is not optimized, or this algorithm is not suitable for the SUT. It needs more test experiments to figure out.

CT vs FT

Coverage Trigger and Fault Trigger does not show obvious difference on branch coverage. The main reason, in my opinion, is that sparse BFS does not improve branch coverage, no matter BFS is triggered by coverage or fault.

Since number of faults is not easy to measure (hard to separate unique faults), the test does not record faults found in each test. In practice, FT function should find more faults, especially for low quality SUT.

Timeout Influence

Test time (timeout) does not show significant influence on test result. For PRT, branch coverage increases from 211/212 to 216 when timeout change from 20 to 120. For MRT-GA, branch coverage increases less. It means MRT-GA needs less time to reach saturation.

4 Interface Design

In order to make the tester more flexible and easy to use, mytester.py contains more input parameters as list below:

abbreviation	name	type	default	meaning
-d	--depth	int	100	Maximum search depth
-t	--timeout	int	60	Timeout in seconds
-s	--seed	int	None	Random seed
-R	--reduce	Bool	False	Reduce -- report reduced failing test
-f	--faults	Bool	False	save a test case for each discovered failure
-N	--normalize	Bool	False	Normalize/simplify after reduction
-w	-width	int	5	maximum memory/BFS queue/other parameter that is basically a search width

-r	--running	Bool	False	Produce running branch coverage report
-c	--coverage	Bool	False	Produce a coverage report at the end
-b	--bfs	Bool	False	Using BFS when find new statements or branches
-m	--ifmutate	Bool	False	Using genetic testing when coverage does not change for a setting time
-a	--auto	Bool	False	According to input timeout, choosing test strategy by tester itself
-T	--TargetOnBug	Bool	False	This mode target on finding bugs

5 References

[1] NIE Peng, GENG Ji, QIN Zhi-guang. Survey on automatic test case generation algorithms for software testing [J]. Application Research of Computers: Vol. 29 No.2, pp.401-413.

[2] Joise Holmes, Alex Groce, Jervis Pinto, Pranjal Mittal, Pooria Azimi, Kevin Kellar, James O'Brien. TSTL: The Template Scripting Testing Language [J]. International Journal on Software Tools for Technology Transfer manuscript.

[3] Pankaj Jalote. Software Project Management in Practice[M]. Tsinghua University Press.