Team 14 - CS454

Final project report

Hoang Trung Dung, Nguyen Thanh Tu, Nguyen Tien Dat, Duc Khanh Thi Nguyen
Github link: https://github.com/VietDunghacker/CS454-team14

I. Introduction

Search-based software engineering (SBSE) has high potential for optimizing non-functional properties such as execution time or power consumption, etc. that is in the offline optimization domain. However, many non-functional properties are dependent not only on the software system under consideration but also the environment that surrounds the system. This leads to the two main problems of offline optimization. The first problem is that sampling bias is hard to avoid. Moreover, even if the offline optimization is already satisfied, the production environment may change toward the direction that degrades the behaviour of the deployed system, making it problematic. In order to solve these issues, we need support for online, in situ optimization that are able to remove the subjective from their environment hence direct and focusing optimization in the specific environment.

II. Amortized Optimization

1. Idea

Usually, optimization algorithms perform fitness evaluations either one by one in case of a local search or as a group when it comes to a population-based algorithm. With the new concept of amortized optimization, optimization algorithms are heavily affected by fitness evaluation. Everytime executing System Under Metaheuristic Optimisation (SUMO), one fitness value was measured. This fitness value measured when running SUMO affects that optimization for the one next step. During optimization, in each iteration, SUMO will be executed multiple times.

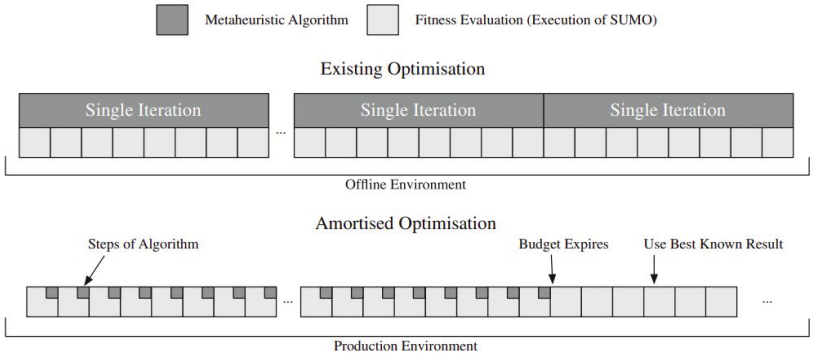

Fig.1 Amortised optimisation interleaves executions of SUMO. After it passes the point that the budget expires, SUMO runs using the best known result so far.

2. Example: Amortized Steepest Hill Climbing

Everytime SUMO is executed, SUMO asks amortised optimization for a candidate solution x. So that, first, the amortised hill climbing algorithm has to know its status. This status can be received from the persistence layer. After having its current status, it will start transitions. This process will continue until it reaches the call RETURN(x), meaning the made solution is returned.

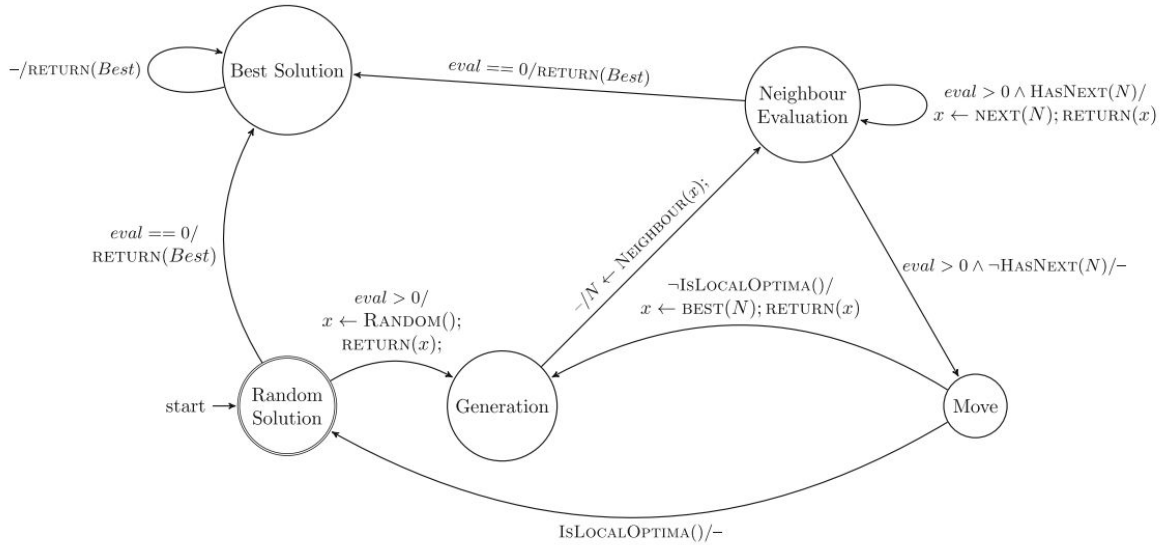

Fig.2 Amortised hill climbing algorithm's state-based model

III. Experiment: Optimizing JIT Parameters for PyPy

1. PyPy

PyPy is an alternative implementation of Python that mostly focuses on Just In Time (JIT) compilation. With the advantage that the JIT compilation has produced efficient implementation of programs by deploying a technique known as meta-tracing. In addition, tracing JIT profiles the code to identify frequently "hot" loops, records the history of all operations executed during a single iteration of a hot loop, and then translates them into machine code. Moreover, to ensure correctness, pypy inserts guards in the translated code. When guards fail (such as an unpredicted branching direction), tracing JIT falls back to interpreting the loop. If guard failure happens above a certain threshold, tracing JIT attempts to translate the sub-path from the point of guard failure to the end of the loop (also called a bridge). However, JIT also has some disadvantages that should be carefully considered because in some cases JIT compilation performs worse than normal compilation, as the cost of tracing will increase the more aggressive PyPy tries to JIT compile.

2. Set of controlled parameter

The parameter controls the behavior of pypy JIT. In this project we put extra attention on 6 parameters that mentions below:

- Function threshold (default 1619): identify how many times a function has to be called before it is traced from the beginning.
- Loop threshold (default 1039): before a loop becomes a hot loop, loop threshold is the number of times a loop has to be executed before that.
- Trace eagerness (default 200): identify how many times a guard (usually unpredicted ranching) fail before pypy compiles the bridge
- Disable unrolling (default 200): after how many operations we should not unroll
- Decay (default 40): amount to regularly decay counters by (0=none, 1000=max)
- Max retrace guards (default 15): number of extra guards a retrace can cause

3. Benchmark user scripts

In our projects, we have 5 users benchmark to work on, these are:

- bm_regex_v8.py: Perform regexp (regular expression) operations on most 50 famous pages on the web
- bm_nltk_intensifier.py: Find all intensifiers in the dictionary
- bm_sort.py: Sort a list of objects
- bm_image_resize.py: Resize all images in a folder
- bm_nonDecInt.py: Find the number of 1000-digit non decreasing integers.

4. Piacin

Piacin is the packet that contains the amortised optimization for JIT parameters. Based on work by professor Shin Yoo. We choose to work experimenting on Amortized Steepest Hill Climbing, Amortized Simulated Annealing with the initial temperature = 1 & alpha = 0.02 and Amortized Genetic Algorithm with initial Population is 10, selection is Tournament Selection, crossover is Swapping parameters between two parents, mutation is randomly mutate one of six controlled parameter with probability ⅛ and generational selection is elitism.

5. Experimental setup

Controlled Parameter

It is recommended from the paper that the loop threshold should be smaller than the function threshold, so the loop threshold parameter will be replaced by the threshold ratio parameter.

$$threshold\ ratio\ = \frac{loop\ threshold}{function\ threshold}$$

Neighborhood solutions for Amortized Steepest Hill Climbing and Amortized Simulated Annealing are generated by adding or subtracting predefined step values to each of the parameters like below:

- Function threshold: 20
- Trace eagerness: 10
- Threshold ratio: 0.05
- Disable unrolling: 10

- Decay: 10
- Max retrace guard: 5

When the newly generated candidate solution has any parameter outside the predefined range, the parameter value is wrapped around the range.

- Function threshold: [100, 5000]
- Trace eagerness: [1, 1000]
- Threshold ratio: [0.01, 0.99]
- Disable unrolling: [1, 1000]
- Decay: [1, 1000]
- Max retrace guard: [1, 100]

Benchmark user scripts

To overcome the inherent randomness in measuring execution times, the scripts are modified to repeat the main test functions 50 times with each execution.

Control group vs. Treatment group

Control group: 20 un-optimized executions

Treatment group: 100 executions (80 for optimization, 20 for test)

Both groups are executed with PyPy version 3.7.1 on Ubuntu OS 18.04.5 LTS, using Intel(R) Core(TM) i7-3520M 2.9gHz CPU with 4 cores and 8GB of RAM.

6. Experimental result

	Default	Hill Climbing	Simulated Annealing	Genetic Algorithm
bm_regex_v8.py	1.85s	1.79s	1.78s	1.52s
bm_nltk_intensifier.py	21.04s	20.66s	20.50s	19.82s
bm_sort.py	12.67s	12.79s	12.86s	12.79s
bm_image_resize.py	17.21s	17.10s	16.92s	16.79s
bm_nonDecInt.py	3.04s	3.00s	3.04s	3.01s

Table 1: The average execution time over 20 executions

7. Result analysis

There is a remarkable improvement in execution time in the three scripts bm_regex_v8.py, bm_nltk_intensifier.py and bm_image_resize.py. The Amortized Genetic Algorithm has the best performance compared to the other two algorithms, with up to 18% reduction in execution time in the case of the script bm_regex_v8.py.

Discussion

The script bm_regex_v8.py and bm_nltk_intensifier.py regularly use the regexp operations, while the script bm_image_resize.py regularly loads the image file from the folder, both of which are known to have lower performance on PyPy. Moreover, the obtained result of the

script bm_sort.py and bm_nonDecInt.py show no improvement in execution time, possibly because the compilation of the script is already optimized by the default choice of parameter.

8. References

Shin Yoo, *Amortised Optimisation of Non-functional Properties in Production Environments*
https://coinse.kaist.ac.kr/publications/pdfs/Yoo2015aa.pdf