TEAM 14

OPTIMIZING THE JIT PARAMETERS OF PYPY USING AMORTIZED OPTIMIZATION

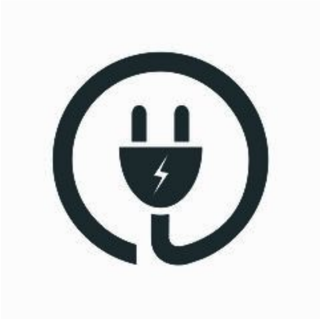
Nguyen Duc Khanh Thi, Hoang Trung Dung, Le Thanh Tu, Nguyen Tien Dat

Introduction

Introduction

SBSE has **high potential** for optimising non-functional properties:

Execution time



power consumption

etc.

Introduction

However, many non-functional properties are dependent on

- The software system under consideration
- The environment that surrounds the system

Offline optimization: **2 issues**

- Avoid sampling bias: difficult
- Offline optimisation is satisfied, production environment may **change** (degrades behaviour of deployed system)

We need a support for online, in situ optimization.

Amortized Optimization

Amortized Optimization

Idea

- Optimization algorithms perform fitness evaluations:
  - one by one (if it is a local search)
  - as a group (a population-based algorithm).
- With amortized optimisation, fitness evaluation drives the optimization algorithm.
- Whenever the System Under Metaheuristic Optimisation (SUMO) is executed, measure **one fitness value** out of it, and **drive optimization forward by a single step.**
- **One iteration** consist of **multiple executions of SUMO**.

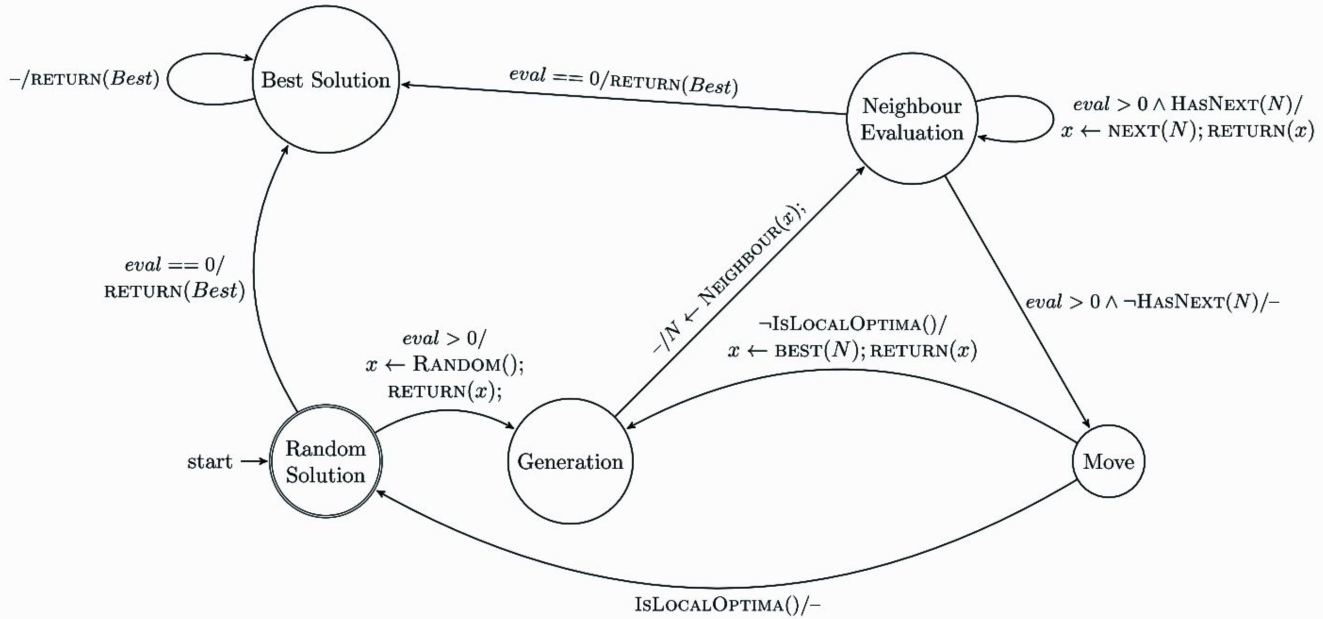
Amortized Optimization

Amortized Steepest Hill Climbing

Whenever the SUMO is executed:

1. Ask for a candidate solution x from the amortised optimisation.
2. Amortised hill climbing algorithm first **retrieves** its **current status from the persistence layer**
3. Executes transitions until it makes a return(x) call.

Amortized Optimization



State-based model of amortised hill climbing algorithm: *return*(x) decreases the remaining number of fitness evaluations, *eval*, by 1

Experiment: Optimizing JIT Parameters for PyPy

Experiment

PyPy

PyPy is an alternative implementation of Python that focuses on Just In Time (JIT) compilation

Advantages:

- JIT compilation has produced efficient implementation (meta-tracing)
- Tracing JIT profiles the code to identify frequently "hot" loops.
- To ensure correctness, pypy inserts guards in the translated code.

Disadvantages:

In some case, JIT compilation performs *worse* than normal compilation (cost of tracing)

Experiment

Set of controlled parameter

- **Function threshold** (default 1619): number of times a function has to be executed before it is traced from the beginning.
- **Loop threshold** (default 1039): number of times a loop has to be executed before it is identified as a hot loop.
- **Trace eagerness** (default 200): number of times a guard (usually unpredicted branching) has to 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

Experiment

Benchmark user scripts

- **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

Experiment

Piacin

- Based on work by professor Shin Yoo
- Amortized Steepest Hill Climbing
- Amortized Simulated Annealing: initial temperature = 1 & alpha = 0.02
- Amortized Genetic Algorithm:
    - Initial Population: 10
    - Selection: Tournament Selection
    - Crossover: Swapping parameters between two parents
    - Mutation: randomly mutate one of six controlled parameter with probability 1/6
    - Generational Selection: elitism

Experiment

Experimental setup

1. **Controlled Parameter:**

- Loop threshold parameter is replaced by threshold ratio parameter.

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

- Neighborhood solutions for Amortized Steepest Hill Climbing and Amortized Simulated Annealing: +/- predefined step values to each of the parameters: Function threshold: 20, Trace eagerness: 10, Threshold ratio: 0.05, Disable unrolling: 10, Decay: 10, Max retrace guard: 5

Experiment

Experimental setup

1. **Controlled Parameter:**

Newly generated candidate solution has any parameter **outside** predefined range: **wrap parameter value 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]

Experiment

Experimental setup

2. Benchmark user scripts

To overcome the **inherent randomness** in measuring execution times:

modify scripts to repeat main test functions **50 times** with **each execution**.

Experiment

Experimental setup

3. 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.

Experimental result

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

Table 1: The average execution time over 20 executions

Result Analysis

Experiment

Result analysis

- **Remarkable improvement** on the first two scripts: up to **18% decrease** in the execution time of the first script with Amortized Genetic Algorithm.
- Among the 3 amortized algorithm, the **Genetic Algorithm** version has the **best** result.

Experiment

Discussion

- The first and the second script regularly use the regexp operations (worse perform on PyPy).
- Result of third script is even worse than the default option
    - Possible reason: the compilation of the script is already optimized by the default choice of parameter.

Thank you
for your attention