24th May, 2018

Python List Comprehensions Optimization

Group Number : 26

Team :

Abhishek Dwivedi - 01FB15ECS008

Aditya D Ramani - 01FB15ECS017

Akshaykanth D L - 01FB15ECS031

List Comprehensions

Provide a more concise way to create lists, sets and dictionaries. Essentially, comprehensions are just syntactic sugar that Python has.

Example:

```
ExampleList = []
for i in range(100):
     if truthness(i):
     ExampleList.append(func(i))
```

A normal list formation using an append function would be this. List Comprehension converts this into a concise, one line code like the one below :

```
ExampleList  = [ func(i) for i in range(100) if truthness(i) ]
```

There's one large shortcoming of comprehensions.

for loops contain statements (e.g. **x = func(y)**) whereas comprehensions can only contain expressions (e.g. **x + func(y)**) and are in fact expressions themselves.

This issue with list comprehensions becomes an actual issue if func is a expensive function and needs to calculated again and again.

There are several ways to overcome:

1. We could rewrite the list comprehension as a series of **map** and **filter** function calls.
2. We could cache the value with lru_cache model. But still is a cumbersome process.
3. Or we can alter your list comprehensions by appending additional **for** loops where your loop invariant work effectively the same as variables.

Let's Begin Our Optimization:

Intial Knowledge:

Every python code is built from an AST. This AST is then compiled to obtain a compile code. We work on changing this AST to optimize the list comprehensions. For this we will be using "ast" module of python. Given a source code, "ast" module returns us a tree with all nodes and related functions. And the functions we actually use are:

- **NodeVisitor**
- **NodeTransformer**
- **Compile**

Step 1 :

Generate an AST from the source code.

Step 2:

Visit each of the node to find if the calls are duplicate. For simplicity we have considered 2 functions are same if their function dump is the same. We use the NodeVisitor to go through all the nodes and find out the duplicate nodes.

Step 3:

Now we have to calculate the value of only one of all the duplicate calls to a particular function. We store this value in a id and replace all subsequent calls with this id. But we have a problem.

Step 4:

We need to make sure that we transmit the id of a call to all the subsequent calls and this should be unique for a set of calls.

We resort to calculating the hash of the function dump and prepending it with "_ _ " to make it a valid id.

Eg: __512226.

Step 5:

Replace the rest of the calls in place by the id calculated as above.

Step 6:

Transform the AST node to be able to show the above changes.

AST for a list comprehensions

<Target function , iters , ifs >

So here we move the function call to new comprehension we defined.

```
expr = ...
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
       ...
     | Call(expr func, expr* args, keyword* keywords,
            expr? starargs, expr? kwargs)

comprehension = (expr target, expr iter, expr* ifs)
```

Fig 1. AST node for elements in python

Node type	Attribute	Value
Add	left	left operand
	right	right operand
And	nodes	list of operands
AssAttr		*attribute as target of assignment*
	expr	expression on the left-hand side of the dot
	attrname	the attribute name, a string
	flags	XXX
AssList	nodes	list of list elements being assigned to
AssName	name	name being assigned to
	flags	XXX
AssTuple	nodes	list of tuple elements being assigned to
Assert	test	the expression to be tested
	fail	the value of the AssertionError
Assign	nodes	a list of assignment targets, one per equal sign
	expr	the value being assigned
AugAssign	node	
	op	
	expr	

Fig 2. AST node for elements in python

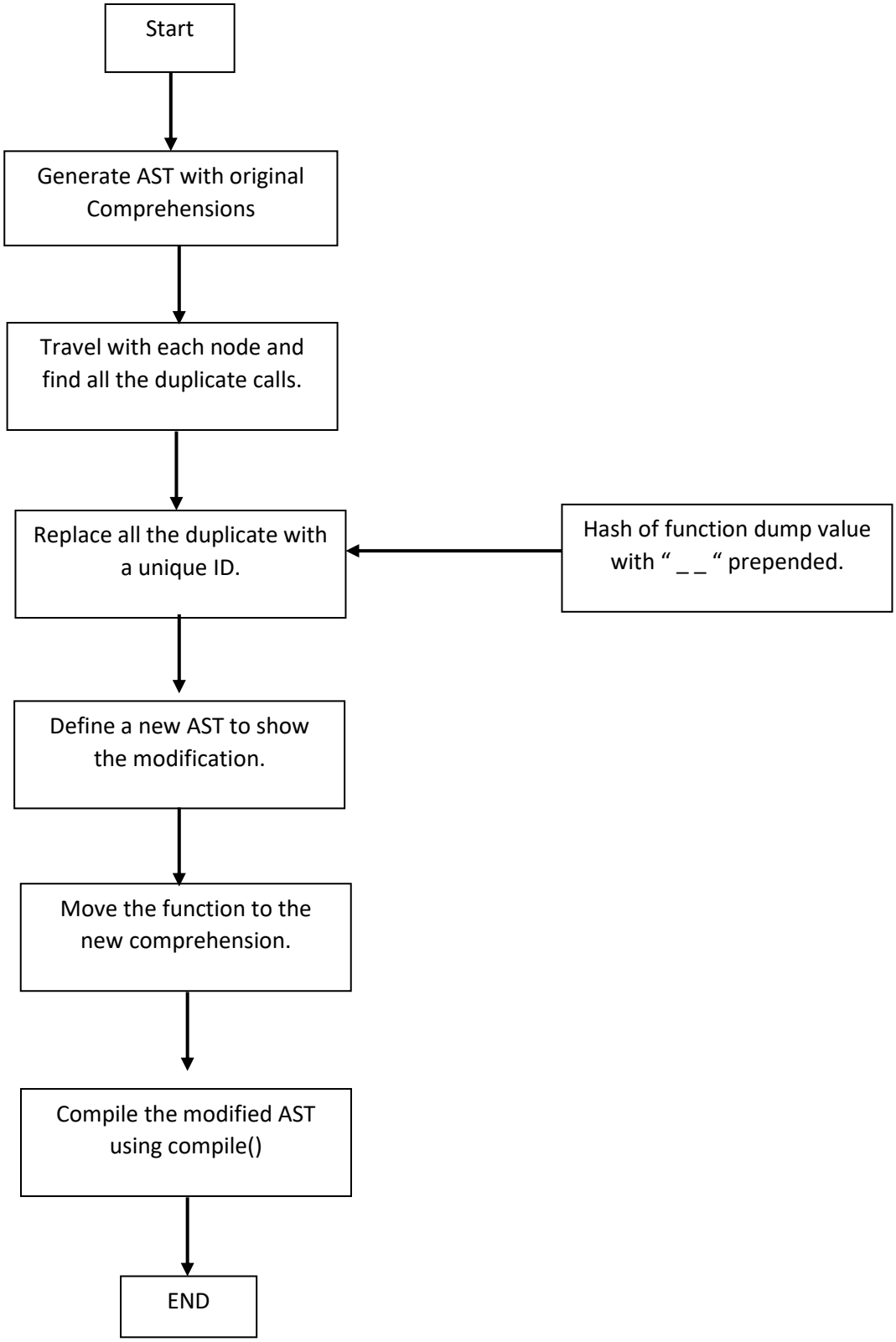

Fig 3. Flow chart of the process

Results :

To compare the results we went with 3 case functions. Plotted graphs for both the modified and the non-modified Comprehensions.

Three functions we considered was :

1. Identity function x.
2. x ** x for x in range(900,1000)
3. matrix multiplication of i X i square matrix.

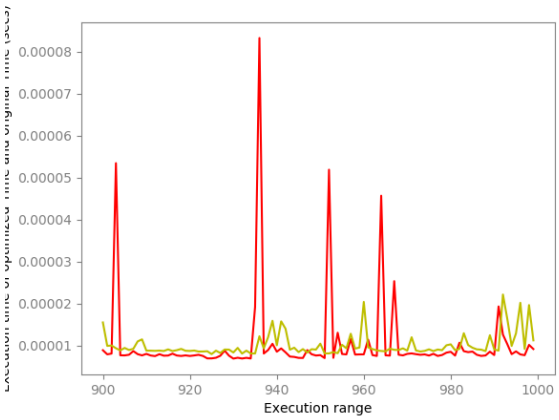

Fig 4. Graph for X

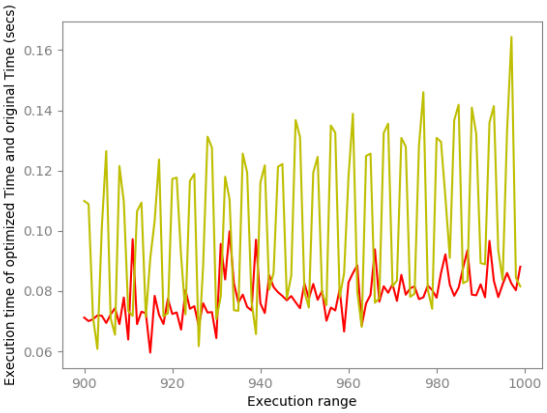

Fig 5. Graph for x**x

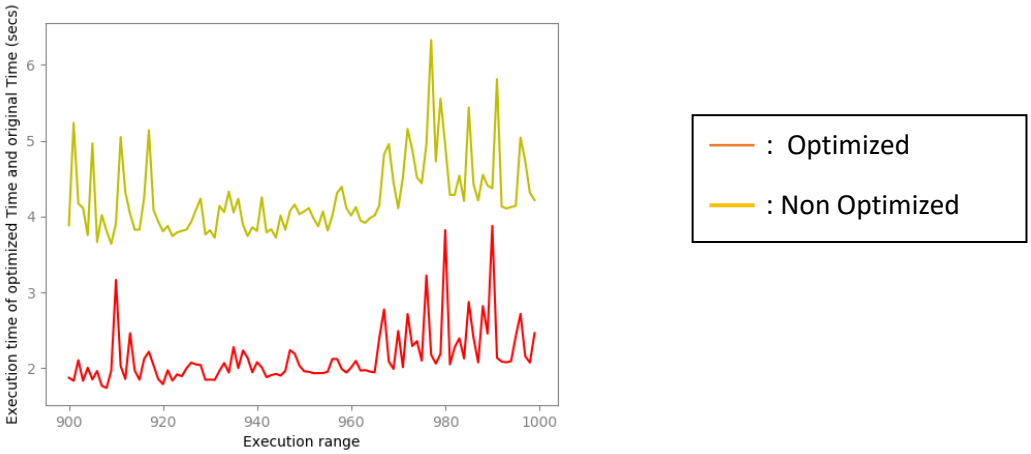

Fig 5. Graph for matrix multiplication

Conclusion:

We provide at an average very less computation time as the ones in the actual python built-in comprehensions. This inference is evident from the graphs above.

This particular model is a vanilla model without considering some obvious setbacks. Our intension was to show that we can produce a much better compiler w.r.t List comprehensions and we think its now just.

Further Improvements and scope:

- Considering better ID selection as hashing might lead to collisions.
- We determine duplicate functions from a function dump, which does not always turn to be good option. Leading to error codes.

Refernces:

- https://wiki.python.org/moin/PythonSpeed/PerformanceTips
- https://stackoverflow.com/questions/13274110/list-comprehension-optimization
- https://docs.python.org/2/library/ast.html
- https://www.journaldev.com/19243/python-ast-abstract-syntax-tree