Setup for running

1. Llvm must be installed

```
$ clang –-version
Debian clang version 10.0.1-++20200529024432+a634a80615b-
1~exp1~20200529005028.166
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

10.0.1 → it needs version 9 at least
/usr/bin → instalation location is important to know!

Make it sure that the following executables exists:

```
$ clang --version
$ opt --version
$ llc --version
```

All of them must be installed in the same folder (in my case */usr/*bin/). If some of those executables has a different name (clang-9 or llc-9 for example) the easiest way is to define some symbolic links:

```
/usr/bin$ ln –s clang-9 clang
/usr/bin$ ln –s opt-9 opt
/usr/bin$ ln –s llc-9 llc
```

Now you have clang, opt and llc located in the same folder
Note: *intallation folder may be different in your computer. Mine is /usr/bin/*

2. En llvmMultiobjetiveProblem.py, line 16:

```
self.llvm = LlvmUtils(
    llvmpath='/usr/bin/',
    source="polybench_small/polybench_small_original.bc" ,
    jobid='llvm_multiobjetive',
    useperf=False)
```

/usr/bin/ → Installation folder of llvm, it must end with / character
polybench_small/polybench_small_original.bc → Be sure that file exists in that folder

3. Run genetic algorithm:

```
$ pyhton3 main_llvm_multiobjetive.py
```

It must work as long you have llvm 9+ installed and the installation folder correctly configured as shows step 2.

Add new measures for GA

1. Add to LlvmUtils.py a new function that measures some feature of .ll or .bc optimized file. For this case it consists in a dummy function that always returns 1.

```
    def get_test(self):
        return 1;
```

2. At llvmMultiobjetiveProblem.py

Locate the following lines:

```
20      self.obj_directions = [self.MINIMIZE, self.MAXIMIZE]
21      self.obj_labels = ['runtime', 'codelines']
22      self.number_of_objectives = 2
```

Add a new objetive to the GA modifying those lines:

```
20      self.obj_directions = [self.MINIMIZE, self.MAXIMIZE, self.MAXIMIZE]
21      self.obj_labels = ['runtime', 'codelines', 'test']
22      self.number_of_objectives = 3
```

In this case I added a new MAXIMIZATION objetive (as codelines objetive) named ‘test’. It could be MINIMIZATION (as runtime objetive) and of course freely named.

Notice that number_of_objetives increased to 3.

3. En llvmMultiobjetiveProblem.py

Locate the following lines:

```
60      solution.objectives[0] = self.llvm.get_codelines(passes=passes)
61      solution.objectives[1] = self.llvm.get_runtime(passes=passes)
62      self.dictionary.update({key: solution.objectives})
63  else:
64      solution.objectives[0] = value[0]
65      solution.objectives[1] = value[1]
```

Add third objetive value gathering:

```
60      solution.objectives[0] = self.llvm.get_codelines(passes=passes)
61      solution.objectives[1] = self.llvm.get_runtime(passes=passes)
62      solution.objectives[2] = self.llvm.get_test() # Function of step 1
63      self.dictionary.update({key: solution.objectives})
64  else:
65      solution.objectives[0] = value[0]
66      solution.objectives[1] = value[1]
67      solution.objectives[2] = value[2]
```

4. Launch main

```
$ pyhton3 main_llvm_multiobjetive.py
```

```
evaluated solution 1 from epoch 1 : variables=[4,46,17,54,10,10,43,62,58,52],fitness=[0.1320042610168457,6002,1]'
evaluated solution 2 from epoch 1 : variables=[63,84,53,84,36,46,52,9,80,42],fitness=[0.15546941757202148,5270,1]'
evaluated solution 3 from epoch 1 : variables=[64,4,78,41,73,81,12,18,56,10],fitness=[0.1372377872467041,5167,1]'
…
Settings:
     Algorithm: NSGAII
     Problem: Llvm Multiobjetive Problem
     Computing time: 9.094732522964478 seconds
     Max evaluations: 20
     Population size: 10
     Offspring population size: 10
     Probability mutation: 0.1
     Probability crossover: 0.3
     Solution length: 10
     Opt executed one by one: 0 times

Results:
          [48, 30, 84, 81, 21, 3, 32, 23, 48, 47]      [0.14612269401550293, 5262, 1]
          [35, 73, 25, 24, 49, 71, 71, 9, 77, 21]      [0.156358003616333, 5005, 1]
          [72, 41, 2, 6, 45, 67, 22, 45, 60, 42]       [0.14722251892089844, 5231, 1]
          [77, 71, 6, 73, 50, 18, 81, 4, 13, 3]        [0.14569497108459473, 5263, 1]
          [77, 71, 8, 20, 50, 72, 81, 7, 47, 4]        [0.15433907508850098, 5067, 1]
```

The last number ‘1’ is the value returned from *get_test* step 1 function. Of course get_test function isn’t measuring any obfuscation at all, it’s just a dummy function that always returns 1.

Checkout get_runtime and get_codelines functions at LlvmUtils.py for inspiration.

The number of final results changes at each execution: it’s completely normal.