BIC Coursework 3

Investigating ECJ GP parameters

Four parameter categories were given by Coursework. In order to investigate effects of parameters to Genetic programming (GP) performance, 3 parameters were chosen. Each parameter was injected into a ready to run GP application inside ECJ package and run the testing with controlled conditions.

After try to run all application inside ECJ package, I noticed that most of them can find a solution to a problem in less than 50 generations with a small size of initial population such as 50 or 100 so I just cross Initialization category out of my selection and try the rest.

The use of Automatically Defined Functions (ADF) and a little about choice of functions.

In traditional way of coding a computer program, usually programmer tend to write a group actions (functions) that were used repeatedly inside a program to safe time of writing the same thing over and over again. This kind of problem is the same in GP. ADF is a reusable set of functions that are defined before running any GP application and allow to use as sub function set. So it might be a good idea to look into a solution constructing performance of GP with ADF or without ADF.

Lawnmower problem from ECJ was selected to investigate the case. The problem is about mowing every square of a lawn with specific size. The ECJ application has come with a ready to run parameter files with and without ADF also the problem is tunable. You can tune a difficulty of the problem by increase the size of lawn. All parameters inside parameters files are default values, the only size of the lawn were changed to 10 x 10 and 12 x 12. A total number of runs on this investigation is 5 for each lawn size and number of generation is 51 which is also a default value.

Result and Discussion

parameters	Problem solving	Size(Avg.)	Fitness (Avg.)	Fitness (worst)	Fitness (best)
No-ADF(10x10)	4/5	728.44	0.9	0.5	1
ADF(10x10)	5/5	[20.52,40.6,19.65]	1	1	1
No-ADF(12x12)	1/5	992.6	0.37	0.11	1
ADF(12x12)	5/5	[27.86,54.98,32.31]	1	1	1

Figure1: Average performances (ADF vs No-ADF) from 5 runs

Results from both ADF and No-ADF are expected to be the same as in Figure1. An average size of solution (inside bracket is [tree1, tree2, tree3]) from ADF clearly answer reusable ability. All trees inside each individual are really small that also mean ADF use less a computational time for each solution. ADF usually solved a problem within 5 generations and it can find 5 solutions out of 5 runs on 10x10 and 12x12 difficulty. On another hand, No-ADF actually perform badly on Lawnmower problem with both 10x10 and 12x12. An average size of solution is very big, it will be even bigger if you let it evolved more than 51 generation no solution and solution may not be found. If you change difficulty of the problem to 12x12, ADF still be able to find solution on every run but No-ADF now perform much worst as you can see in Figure1.

Furthermore I did try some extra experiment about a choice of functions. There are 2 functions that mow a lawn, one if Frog and one is Mow. Frog will randomly jump within lawn space and maw destination space. Mow is move in some certain direction and then mow the lawn. First I try to remove Frog function from No-ADF parameters, result is unexpected. No-ADF now can solved 12x12 lawn within 51 generation on every run, in fact it can solved even harder lawn if you give it enough generation to evolve. Next I add Frog to 2nd tree of ADF (default have no Frog), I expected that performance should be reduced and it is true. ADF still found 5 solutions from 5 run but it use more generation to obtain a solution. So it mean inappropriate function can greatly reduce a performance of GP application.

Selection mechanisms.

Selection mechanisms are method that operator use to select candidates from population to do Biological evolution such as Mutation and Crossover. The selection processes have played an important role in Evolutionary Computation (EC) from the beginning and it still be the same in GP. The selection did not only select candidates to live but also to die. So it is clear that GP performances will be affected by these parameters. There are a lot of options in ECJ to choose as follow.

1. FirstSelection: Always select first individual from population. Some parameters had been overwritten as follow.

```
gp.koza.mutate.source.0 = ec.select.FirstSelection
gp.koza.xover.source.0 = ec.select.FirstSelection
```

2. RandomSelection: Randomly select individual from population. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.RandomSelection
gp.koza.mutate.source.0 = ec.select.RandomSelection
```

3. FitProportionateSelection: Normal Roulette Selection. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.FitProportionateSelection
gp.koza.mutate.source.0 = ec.select.FitProportionateSelection
```

4. SUSSelection: Work like FitProportionateSelection but with low-variance. High fitness individual are more unlikely to never be selected. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.SUSSelection
gp.koza.mutate.source.0 = ec.select.SUSSelection
```

5. SigmaScalingSelection: Work like FitProportionateSelection but with low-variance (calculate using modified fitness). Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.SigmaScalingSelection
gp.koza.mutate.source.0 = ec.select.SigmaScalingSelection
```

6. BoltzmanSelection: Work like FitProportionateSelection but with a modify fitness until the temperature is dropped below 0.1 then do normal FitProportionateSelection. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.BoltzmanSelection
gp.koza.mutate.source.0 = ec.select.BoltzmanSelection
select.boltzmann.starting-temperature = 1000
select.boltzmann.cooling-rate = 0.97
```

7. GreedyoverSelection: Grouping population into fitter and less fitness and randomly select the group with given probability than perform FitProportionateSelection on selected group. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.GreedyOverselection
gp.koza.mutate.source.0 = ec.select.GreedyOverselection
select.greedy.top = 0.10
select.greedy.gets = 0.80
```

8. BestSelection: Select best fitness individual of size n from population then perform a tournament. Some parameters had been overwritten as follow.

```
gp.koza.xover.source.0 = ec.select.BestSelection
gp.koza.mutate.source.0 = ec.select.BestSelection
select.best.n = 14
select.best.size = 7
```

9. TournamentSelection: Default selection. Randomly select candidates then form a tournament. Winner will send to do evolution.

Default parameters were used to GP application.

10. MultiSelection: Support combination of multiple selection parameters

I did not use this parameter in this Coursework because a combination of parameter required way too much than 5 page.

NOTE: Default ECJ operator are actually not the same pair as in ECJ document. So we have to change it from Reproduction to Mutation. Some parameters had been overwritten as follow.

```
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.MutationPipeline
```

Royaltree tree problem was selected to investigate these parameters. Runnable solution from GP application usually have a good tree structure (correct terminal sets and function sets). This benchmark calculate a fitness base on how close the solution is to a target tree. The difficulty of the problem can be tuned by adding more alphabet (terminal) into a problem. You can find a detail of this benchmark in Goodman et al (1996) paper. The fitness (score) of this problem will be higher than 100 if a perfect solution was founded.

Without proper combination of parameters most of Benchmark application from ECJ produced unstable fitness if you change only a selection parameter. In one type of selection, resulted fitness can be like 0.25 – 1. So I choose Royaltree because the results is easier to evaluate selection parameters. All other parameters in this investigation are default parameters of this problem. No other parameters were changed.

Result and Discussion

Parameters	Generation(Avg.)	Fitness(Avg.)
FirstSelection	0	17.75
RandomSelection	0	18.27
FitProportionateSelection	484	42.79
SUSSelection	481.9	44.9
SigmaScalingSelection	499	70.5
BoltzmanSelection	0	17.75

GreedyoverSelection	390.3	75.34
BestSelection	87.7	71.84
TournamentSelection	281	72.38

Figure2: Average fitness and generation from 20 runs

In this section, default generation was set to 500 and population at 3500. Most of default values are unchanged except on GreedyoverSelection and BoltzmanSelection. First I try to imitate Over-selection in Goodman et al (1996) paper by setting select.greedy.top = 0.10 (best fit 10% of population) and select.greedy.gets = 0.80 (probability to select from best fit). An average of GreedyoverSelection showed better result but it is not much different than Tournament and Best selection. But if you consider an average generation that best fit solutions were found. BestSelection and TournamentSelection are better in term of computation time to find best fit solution. SigmaScalingSelection have the most interesting performance. An average fitness is above 70 but when you consider and average generation, it is 499 which mean it still evolving. If you provided more generation to this selection results fitness should be much better but computation cost is also very high. For BoltzmanSelection, ECJ manual stated that in order to effectively run this parameter a temperature should reduce to 0.1 before the all generation are ended so I set select.boltzman.cooling-rate = 0.97 but overall result is very poor. For all 0 value in Figure2 mean that fitness in each generation have no direction, it just randomly evolve slightly better or worse fitness than an initial population.

Crossover operators.

Operators always play a crucial role in EC – GP, without them you cannot evolve your population. So it quite Obvious almost every one choose to investigate these parameters.

There are 3 Crossover operators and 8 Mutation operators in GP and Korza package that came along with ECJ. I did try all Mutation operator on high difficulty level of several Benchmark application but did not see much difference in performance with other default parameters. But Crossover operators clearly show effects of different operators. Also again, result of 8 mutation operators is a bit too much for 5 page. The list of Crossover operator are as follow

1. InternalCrossoverPipeline: Select 2 node in the same individual, node can be in different tree and then swap subtrees. Some parameters had been overwritten as follow.

```
pop.subpop.0.species.pipe.source.0 = ec.gp.breed.InternalCrossoverPipeline
gp.breed.internal-xover.source.0 = ec.select.TournamentSelection
gp.breed.internal-xover.source.1 = same
gp.breed.internal-xover.ns.0 = ec.gp.koza.KozaNodeSelector
gp.breed.internal-xover.ns.1 = same
gp.breed.internal-xover.maxdepth = 17
gp.breed.internal-xover.tries = 1
```

2. SizeFairCrossoverPipline: You can see a detail of how this operator work in The ECJ Owner's Manual (2014). Some parameters had been overwritten as follow.

```
pop.subpop.0.species.pipe.source.0 = ec.gp.breed.SizeFairCrossoverPipeline
gp.breed.size-fair.source.0 = ec.select.TournamentSelection
gp.breed.size-fair.source.1 = same
gp.breed.size-fair.ns.0 = ec.gp.koza.KozaNodeSelector
gp.breed.size-fair.ns.1 = same
gp.breed.size-fair.maxdepth = 17
gp.breed.size-fair.tries = 1
gp.breed.size-fair.homologous = false
```

3. CrossoverPipline: Default Crossover operator, select 2 individuals then select node inside each individual and then swap subtrees.

NOTE: Default ECJ operators are actually not the same pair as in ECJ document. So we have to change it from Reproduction to Mutation. Some parameters had been overwritten as follow.

```
pop.subpop.0.species.pipe.source.1 = ec.gp.koza.MutationPipeline
```

Parity problem was selected to investigate these parameters. Main reason that I chose this problem because it have ready to run ADF parameter so Internal Crossover operator may have some benefit from multiple trees inside each individual in population. No other parameters were changed.

Result and Discussion

Operators	Times (Avg.)	Problem solving	Fitness (Avg.)	Fitness (worst)	Fitness (Best)
InternalCrossoverPipeline	394.1	0/20	0.17	0.12	0.33
SizeFairCrossoverPipline	-	-	-	-	-
CrossoverPipline	727.2	2/20	0.22	0.12	1

Figure3: Average performances from 20 runs

First A problem with SizeFairCrossoverPipline. SizeFairCrossoverPipline will only run a few generation and then throw out a Null pointer exception on every run. I am quite sure that this is a bug of this pipeline. In this investigation, I expected a result from Internal Crossover should be better than the result in Figure3. Because, normally when using this operator with normal GP application that have only 1 tree inside an individual, it will act like Mutation operator so I expected more from multiple trees. An average computation time (in millisecond) of Internal Crossover is small as it should be (An operator only work on one individual). Internal Crossover overall Fitness is very poor and it never found a solution across 20 runs. The same goes for default Crossover operator (CrossoverPipline) but it do slightly better. An average fitness across 20 runs have small different from Internal Crossover operator but some time it can find a solution to the problem (2 out of 20 runs).

Conclusion

With some extra experiment it clearly that only one parameter from one of categories is not enough to find a solution in most standard Benchmark for GP applications. In order to find an effective way solve a problem a combination of many parameters are required for certain problem. I did some extra experiment on Royal tree benchmark by trying many combination, some combination produce much higher fitness than the result in this Coursework (250+ fitness).

References

Luke, S., (2014). The ECJ Owner's Manual. [E-Book]: Department of Computer Science: George Mason University: http://cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf [Accessed 20 December 2014].

Goodman, E., Kinnear, K., Punch, B. and Zongker, Z., (1996). The Royal Tree Problem, a Benchmark for Single and Multi-population Genetic Programming. [E-Book]: https://isotropic.org/papers/GARAGe96-01-01.pdf [Accessed 20 December 2014].