Biologically Inspired Computation: Coursework 2

Experimenting with Genetic Programming and ECJ

Ronan Smith, H00189534, rs6@hw.ac.uk

Experiment 1

The first experiment aims to explore how the function set, or the set of functions from which solutions can be constructed, affects the ability of tree-based Genetic Programming (GP) to find results. This was done using the *Santa Fe Ant Trail* benchmark problem, and results were found by editing or removing certain parameters inside the parameter file for this problem.

The choice of parameters in the function set can significantly improve or worsen results, as they typically control the abilities of the algorithm. For example, in the Santa Fe Ant Trail problem discussed here, there are parameters in the function set that control the direction a virtual ant can move in (forward, left or right). If one or more of these movement abilities was removed you would expect that the quality of results would be affected negatively, as the ant has less possible ways it can go about solving the problem. It has also been found that the average fitness in randomly generated programs tends to grow as the size of the population increases with the size of their default schema [1], so having less parameters in general would be expected to have a negative effect on the results.

For this experiment I modified a number of different parameters and ran the problem in ECJ. These modifications included removing parameters and editing their values. The first run resulted in a fitness of 21.0. After this I increased the number of generations, hoping that this would improve the result, but this was not the case. Some of the results for fitness in this case were actually higher – for example 22.0 or 29.0. Increasing the population size from 1000 to 5000 did not seem to make a difference either, and the fitness obtained when this was carried out was 25.0.

After those unsuccessful tests I decided to start changing the possible moves an ant could make. Without forward moves, the best fitness obtained was 89.0 – significantly higher (and worse) than the value of 21.0 obtained on the first run. Removing left and right moves one by one also had a negative affect on the fitness but it never reached as high as 89.0. The average fitness obtained without right moves (from 5 runs) was 29.8 and the average fitness with no left moves in the same number of runs was 35.0, although the lowest fitness with no left moves was actually the lowest obtained at 18.0.

These results prove that removing important parameters from the function set can make the results much worse in general however, in this experiment, individual results can vary quite significantly and therefore there may be occasions when you can achieve better results with a smaller function set.

Experiment 2

Experiment 2 looks at *Automatically-Defined Functions (ADFs)* and the *Lawnmore Problem*.

ADFs are an extension to genetic programming where a set of programs or modules are brought together to form one modular system and this is usually implemented in a hierarchical manner with some parent and multiple ADFs inheriting from it [2]. The *Lawnmore Problem* is a benchmark problem that can be used to evaluate ADFs. In the problem, there is a virtual 'lawn' which is made up of a grid of n x m squares and the virtual lawnmower must navigate round each square in the grid 'mowing the grass' [3].

The use of modules allows for a lot of reuse, which in turn saves development time as each module or component only needs to be defined once rather than every time it is needed. Further to this, modules can be reused in different systems and if they are defined correctly they should be able to fit into another modular system almost like a jigsaw piece.

For this experiment, I ran two parameter files in ECJ. These are lawnmower_with_adfs.params, which I will call 'with_adfs' and lawnmower_without_adfs.params, which I will call 'without_adfs'. Each of these files was able to find a perfect member of the population on most runs (i.e. a member of the population with a fitness value of zero). Due to the fact that both run till they solve the problem, I decided to compare them for how many generations it takes for each to successfully do this. Upon running ECJ with each of the files I discovered the following:

- The with_adfs file typically found a member of the population with an ideal fitness inside 1,2 or 3 generations.
- The without_adfs file was much slower, typically solving the problem in 13, 15 or 16 generations.

The average number of generations over the 7 runs for the with_adfs file was 2.28; in other words just over 2000 evaluations. The average for the without_adfs file was 14.29; or just over 14000 evaluations. Both files produce a fairly small standard deviation; with_adf is 0.95 and without_adf is 1.25. The standard deviation is smaller for the with_adf file which shows it is more efficient.

It is clear to see that in this example the modular method is much more efficient. The *Lawnmower Problem* is modular by nature and this may be one of the reasons that the approach and the problem work so well together.

Experiment 3

The third and final experiment for this coursework assignment involves comparing the *tree-based* representation of Koza GP and the *grammar-based* representation of grammatical evolution.

The Koza GP tree-based representation of genetic programming is an approach that represents problems by their parse trees [4]. This is a very visual representation and can be much more efficient than others. Grammatical evolution (GE) is a completely different approach that models individuals as strings of binary characters (1s or 0s), and evolves them by moving the characters around. In grammatical evolution the following steps are carried out [5]:

- A context-free grammar is defined in BNF which has a series of production rules.
- A variable-length binary string is defined for representing individuals in the population.
- Bits are read from an individual string in groups of 8, called codons, and decoded to some integer from this binary representation.
- These integers are then repeatedly mapped to expressions using the BNF grammar that has been defined until a syntactically correct expression – a solution – is formed.

For this, experiment I ran each of the algorithms (Koza GP and grammatical evolution) 14 times, with each being run 7 times on each of the problems (the lawnmower problem and the santa fe ant trail problem).

The lawnmower problem was not solved completely with the grammatical evolution algorithm, with the lowest fitness found being 36.0, and the average being 43.0. The standard deviation was 5.7. This algorithm always ran 50 times in these experiments meaning 50000 evaluations took place. In comparison, the Koza GP algorithm was much more successful, finding a solution in 14.6 generations on average, around 14600 evaluations, and finding an ideal candidate on every run.

Neither of the algorithms were able to find a solution within the maximum 50 generations for the Santa Fe Ant Trail problem. However, Koza GP once again seemed to be the better candidate. Over the 7 times each algorithm was run on this problem, the Koza GP algorithm achieved a mean fitness of 26.9 in comparison with the much higher mean of 53.7 for the GE algorithm. Koza GP did have much more variation however, with a standard deviation of 10.5 in comparison with 5.4 for GE. The lowest fitness value achieved for the Koza GP algorithm was 11.0 whereas the lowest value achieved for the GE algorithm was 46.0. Furthermore, the value of 46.0 was higher than the highest fitness value for Koza GP; 39.0.

According to these results, the Koza GP tree-based algorithm is much more efficient and successful on both the lawnmower and santa fe ant trail problems in comparison with the grammatical evolution algorithm.

Conclusion

In conclusion, I have familiarised myself with ECJ and its facilities through carrying out the 3 experiments described, and I have investigated and discussed how different design choices can affect the behaviour of GP. The design choices discussed include the manipulation of a function set and ways to structure your data including modular styles and hierarchies. Furthermore, I discussed two different types of genetic algorithm; namely Koza GP and grammatical evolution, and compared their results when run on different problems.

Please see the full file of test data at the following URL:
https://drive.google.com/file/d/1polvqmkBm11lS5jgw39xTd23n8WjceK0/view?usp=sharing

References

1. Wilson, D. & Kaur, D., 2013. *How Santa Fe Ants Evolve*.
2. Bruce, W.S., 1997. *The lawnmower problem revisited: Stack-based genetic programming and automatically defined functions*. In Genetic Programming 1997: Proceedings of the Second Annual Conference.
3. Sotto, L.F.D.P. and de Melo, V.V., 2016, July. *Solving the Lawn Mower problem with Kaizen Programming and λ-Linear Genetic Programming for Module Acquisition*. In Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion (pp. 113-114). ACM.
4. Lones, M. *Genetic Programming* [online] Available from http://www.macs.hw.ac.uk/~ml355/common/thesis/c6.html [Last accessed 29th November 2017].
5. Brownlee, J. *Clever Algorithms: Nature-Inspired Programming Recipes* [online]. Available from http://www.cleveralgorithms.com/nature-inspired/evolution/grammatical_evolution.html [Last accessed 29th November 2017].