Benchmarking Unified Evolutionary Computation Methods

Patrick Stephen

May 8, 2017

1 Abstract

Evolutionary computation as a field strives to justify itself by using a suite of tests that run the gamut of problem archetypes, difficulties, and domains. A difficulty that is presented in these benchmarks is how to compare the results of two disparate evolutionary computation methods where each method may have dozens of similar features implemented entirely differently. This paper describes GoEvo, a framework built to unify disparate evolutionary computation methods. and benchmarks these against one-another, commenting on the necessary progress to be made and further benchmarks required for the field and the framework.

2 Overview

Artificial Intelligence has been dogged by the pursuit of two interconnected targets since its inception: General Intelligence, or the concept of a program which presents knowledge over more than a single domain, specifically all domains, able to tackle general problems, and unsupervised learning, a method of achieving a program's intelligence with as little intervention from human actors as possible. Unsupervised learning is particularly notable as a closer target with which General Intelligence could be achieved. Should a program be able to continually teach itself, it may find its own way to General Intelligence.

The pursuit of unsupervised learning has developed its own sub-field within AI, Machine Learning. In Machine Learning statistical methods and, more recently developed, large convolutional neural networks are used to remove responsibility from human AI programmers except for defining some parameters, inputs, and expected outputs.

One method of approach to Machine Learning was through the Genetic Algorithm (GA), the first method in a subgroup called Evolutionary Computation. GAs use biological principles to solve problems, generating a population of random programs that would be selected from, mutated, and crossed with eachother over pseudo-generations to produce hypothetically ideal solutions for problems. In this way, after defining problem parameters human coders could leave their GA evolving and return later to determine what, if any solution was reached. The GA, however, is very precise and hard to adjust. While GAs are suitable for representing specific solutions for specific problems, they aren't suitable for a General Intelligence, as they don't take input or leave output in a flexible manner.

On the evolutionary model of GAs, several successors formed to better apply to General Intelligence. These include, for our interest, the Tree-based Genetic Program (TGP), the Linear Genetic Program (LGP), and the Evolutionary Neural Network (ENN).

The Genetic Program models each execute a series of recognizable instructions such as "add" or "multiply". TGPs structure program instructions in an execution tree, with branches being conditionally executed by parent nodes, and inputs are represented in leaf nodes. LGPs execute a series of instructions in order, and they represent conditions through assembly-styled branch and jump methods. LGPs need to have some concept of memory to save and execute instructions on multiple values, TGPs do not need to have any concept of memory storage, and can just return whatever value leaves their top node.

ENNs do not follow an execution structure like Genetic Programs. ENNs take in some input of number values and run them through a neural network to output some other number values. These two layers are called the input and output neuron layers, and there exist some number of hidden layers between the two which act as a black box, multiplying their inputs by defined weights and passing on values to later hidden neuron layers depending on threshold or activator functions. The distinction between ENNs and the usual neural network is that neural networks are usually modified through backpropagation. ENNs use the same methods as GAs to evolve to reach a goal, instead.

Given these four methods, its of interest to us to measure how each compares against the same problems given as much of a combined effort and evolutionary framework as possible. This paper documents our efforts to model each of these frameworks in the same codebase such that each shares as much of the same algorithms and code as possible. Our goal in this is to avoid any method having an advantage over another due to algorithmic differences, bugs, or inefficiencies that isn't core to each method's structure in the first place.

Benchmarking different learning approaches is an important step in the field's progress towards achieving unsupervised learning. Virtually all methods of unsupervised learning claim their own advantages, and viewed on their own are impressive. Without a broader perspective which combines multiple methods, it takes significant research and experimentation just to establish which method is most appropriate for a particular problem. Having no combined view also makes it difficult to judge if a method is objectively less useful than other methods. To direct where research should be focused, a review of these methods in fair environments needs to produce some conglomerate view, analyzing each method for its benefits and its drawbacks, noting too whether any method is always less useful than another method. This is also true for smaller components of Evolutionary Computation such as which selection method or crossover method is used, what mutation parameters each method is given, the size of crossover groups, etc.

3 Background

There have been several prominent benchmarks of different evolutionary computation methods in the past, far beyond the scope of this project, largely due to the limited time-frame of this project and the establishment of this project in a relatively underdeveloped programming language.

Linear Genetic Programming and Cartesian Genetic Programming, a form of GPs which is structured as a grid as opposed to LGP's line or TGP's tree, were compared by Wilson and Banzhaf in 2008 [8]. Their research covers few problems and is largely focused on judging whether the two forms can be resolved to be equivalent methods through conversion to a third form of GP. The problems focused on in their research are industry problems and classification problems, however. This is true both of Banzhaf's 2008 paper and his 2001 paper, both focusing on classification problems for medical data, the 2001 paper comparing against neural networks (but not evolutionary neural networks) instead [2]. This project is not concerned with classification, as its the opinion of this project that deep learning has effectively solved classification problems, and so this research is not very helpful to us.

Banzhaf has worked on LGPs for quite some time however and has a written a book which features a large subsection comparing LGPs to TGPs [1]. This work was not considered when developing the model of LGP present in this project, but by coincidence the two models share a good number of features. Whereas the previous paper focused largely on problems beyond the scope of this work, the problems used in Banzhaf's comparison here are of a similar nature to the problems we focus on, attempting to be problems of principle or simple problems that demonstrate a capability in the computation method.

The most important article of background information for this project, however, is aptly titled 'Genetic programming needs better benchmarks' [4]. In this paper McDermott and co. critique the current basis of benchmarking in genetic programming and evolutionary computation, noting that most benchmarks used are used for historical purposes only without regard for whether they represent effective new problem areas or introduce new considerations for problems. While most of the problems that this project focuses on are simpler than those approached by McDermott and co., we did attempt four of the problems they present as potential components of the "GP benchmarking suite": Mario, TSP, Tartarus, and Sorting. Unfortunately due to time constraints and limitations only the latter two problems will be addressed in this paper. The remaining problems we address are perhaps simpler than McDermott suggests are useful for potential benchmarks, but they prove useful for us due to the limited nature of our timeframe.

It's unfortunate to note that McDermott and co. attempted to do what this project also will attempt to do, set up some formalized web location storing benchmarks and benchmark results given various settings, but their wiki page they set up is already down. When considering how to make benchmark results portable and long-lasting, their failure to leave breadcrumbs to their efforts will need to be taken into account.

4 Approach

To perform this benchmark, the codebase GoEvo was built. GoEvo features several primary packages describing primary evolutionary algorithms and interfaces for implementing packages to define evolutionary individuals on.

All selection methods are abstracted out of implementing packages and run directly on fitness values of individuals instead of on individual genome representations. One selection method which converts the fitnesses of a population to weights and then repeatedly performs weighted selection without removal is shown in figure 1.

Other selection methods include stochastically performing the same operation as above by performing a single linear pass with an increment defined by the total weight in the system and the number of elements to be chosen, a greedy selection which simply returns the top percentage of a population, and two tournament selections where several "tournament fights" take place between population individuals and those with a higher fitness either 1) always win or 2) have a greater chance to win.

Before crossover can be performed on a population of selected individuals, individuals need to be paired up into parent groups for crossover to take place between. These parent groups are defined by pairing methods, and GoEvo also abstracts these out from individual methods. The two implemented are very simple: alpha-pairing and random pairing, alpha-pairing requiring that one of each parent pair is among some top percentage of the population.

```
type Probabilistic struct {
   ParentProportion int
   Power         float64
}

func (ps Probabilistic) Select(p *pop.
    Population) []pop.Individual {

   weights, _ := p.Weights(ps.Power)

   maxWeight := 0.0
   for _, w := range weights {
      if w < maxWeight {
         maxWeight = w
      }
   }

   next := 0
   members := make([]pop.Individual, p.Size)

   for i := 0; i < p.Size/ps.
       ParentProportion; i++ {
      for {
         next = rand.Intn(len(weights))
         if rand.Float64() < (weights[next]
             / maxWeight) {
            break
         }
      }
      members[i] = p.Members[next]
   }

   return members
}
```

On each population the following algorithm is run each generation to manage each step of the evolution:

```
func (p *Population) NextGeneration() bool {

   // The number of parents in
   // the next generation
   parentSize := p.Size / p.Selection.
       GetParentProportion()

   p.GenerateFitness()
   if p.LowFitness <= p.GoalFitness {
      return true
   }
   elites := p.GetElites()
   nextGen := p.Selection.Select(p)

   // Ensure that the elites (the best
   // members) stay in the next generation
   for i, elite := range elites {
      nextGen[i+parentSize] = nextGen[i]
      nextGen[i] = elite
   }
   parentSize += p.Elites
   p.Members = nextGen
   pairs := p.Pairing.Pair(p, parentSize)

   // i does not start at 0,
   // but pairs, sensibly, does.
   pairIndex := 0

   // crossover pairs for children
   // in the next generation.
   for i := parentSize; i < len(nextGen); i
        ++ {
      n1 := p.Members[pairs[pairIndex][0]]
      n2 := p.Members[pairs[pairIndex][1]]
      p.Members[i] = n1.Crossover(n2)
      pairIndex++
   }

   // Mutate. The elites are not subject to
        mutation.
   for i := p.Elites; i < len(p.Members); i
        ++ {
      p.Members[i].Mutate()
   }

   return false
}
```

Populations are generated by specific functions for each method, taking in a variety of options. One of these options[10] is the migration chance of a demetic group, as one example of the details that went into these algorithms. An experiment can be run on a population or on a demetic group. A demetic group is a wrapper around a set of populations wherein each population is evaluated on its own and the demetic group controls anything that takes place between demes, a word used for populations in the sense of a demetic group. The following algorithm details how the migration chance is used in a demetic group.

```
func (dg *DemeGroup) NextGeneration() bool {
   migrators := 0.0
   if dg.MigrationChance >= 1.0 {
      migrators = math.Floor(dg.
           MigrationChance)
   }
   if rand.Float64() < dg.MigrationChance-
        migrators {
      migrators += 1.0
   }

   // Shuffle the deme order
   if migrators >= 1.0 {
      for i := range dg.demes {
         j := rand.Intn(i + 1)
         dg.demes[i], dg.demes[j] = dg.demes
              [j], dg.demes[i]
      }
      for i := 0; i < int(math.Floor(
           migrators)); i++ {
         // We pick migrators from demes in
         //order (or, randomly, as we just
         // shuffled them) and need to loop
         // around once we've hit
         // the total deme count.
         j := i % len(dg.demes)
         // We always migrate to the deme
         // immediately following the deme
         // at j.
         k := (j + 1) % len(dg.demes)
         d1 := dg.demes[j].Members
         d2 := dg.demes[k].Members

         // We assume all elements
         // in the length of d1 and d2 are
         // populated with individuals. This
         // should be the case, as they
         // should be full outside of next
         // generation calls.
         m := rand.Intn(len(d1))
         n := rand.Intn(len(d2))

         // Perform the actual migration
              swap
         d1[m], d2[n] = d2[n], d1[m]
      }
   }
   stopEarly := false
   for i := range dg.demes {
      stopEarly = dg.demes[i].NextGeneration
           () || stopEarly
   }
   return stopEarly
}
```

Problems are defined by generating a sequence of inputs and outputs, then giving those inputs and output to implementing method packages through those packages' individuals' fitness functions. The problem of sorting a list follows as an example:

```
func SortListTestCase() TestCase {
   in := [][]float64{
      {2.0, 3.0, 1.0, 5.0, 4.0},
      {7.0, 9.0, 8.0, 11.0, 12.0, 10.0},
      {15.0, 14.0, 13.0},
   }

   out := make([][]float64, len(in))
   for i, f := range in {
      out[i] = make([]float64, len(f))
      copy(out[i], f)
      sort.Float64s(out[i])
   }

   return TestCase{
      in,
      out,
      "SortList",
   }
}
```

Eventually these input and output sets reach fitness function endpoints where the inputs are assigned to individuals in populations and manipulated by those individuals. Then the manipulations those individuals perform are compared to the output in some way of judging how fit each individual is. Two such methods follow, one which compares the first element in a LGP's memory to the first element in the output, and one which modifies another fitness function by worsening an individual's fitness as it gets more complex.

```
func Mem0Fitness(g *LGP, inputs, outputs
    [][]float64) int {
   fitness := 1
   for i, envDiff := range inputs {
      g.Env = environment.New(envDiff)
      g.Run()
      v := int(math.Abs(float64(*(*g.Mem)
          [0]) - outputs[i][0]))
      fitness += v
   }
   return fitness
}

func ComplexityFitness(f FitnessFunc, mod
    float64) FitnessFunc {
   return func(g *LGP, inputs, outputs [][]
       float64) int {
      i := f(g, inputs, outputs)
      i += int(math.Floor(mod * float64(len(
          g.Instructions))))
      if i < 0 {
         i = math.MaxInt32
      }
      return i
   }
}
```

The list of problems which were implemented and tested against in the timeframe of this project so far follows.

1. Pow 8. Given x, produce x^8. Fitness range for this problem: 1 to infinity.[11] This problem is a specifically interesting instance of the more general problem: given x and y, produce x^y. The general case was avoided as smaller numbers than 8 were not interesting and higher numbers ran into integer overflow issues in implementation.

2. Pow Sum. Given x, produce $\Sigma_i^x i$. Fitness range for this problem: 1 to infinity.[12] As above, a specifically interesting instance of a more general problem: $\Sigma_i^x i^y$.

3. Reverse List. Given a list, produce that list in reverse order. Fitness range for this problem: 1 to 15.

4. Matrix Transpose. Given a matrix, produce its transposition. Fitness range for this problem: 1 to 9.

5. Tartarus. Perform against the Tartarus problem. Fitness range for this problem: 1 to 13. [7][13]

6. Sort List. Given a list of numbers, produce that list, sorted. Fitness range for this problem: 1 to 15.

7. Matrix Multiply. Given two matrices, produce the resultant of the two multiplied together. Fitness range for this problem: 1 to 9.

8. Pathing. Given a grid of weights, find the lowest weighted path from the top left corner to the bottom right corner. Fitness range for this problem: 15 to infinite.

While this project was originally intended to target GAs alongside the remaining methods, this was largely out of interest to compare GA-based solutions to the Traveling Salesman Problem [3] to non-GA solutions. Due to time constraints TSP is not a part of this paper, and so GAs are also not included in these benchmarks.

5 Results

5.1 Experiment Design

Each of the experiments was put together as a test case and settings combination as described in the Approach section. Then a goal fitness was set for each problem, the goal defining the value at which an individual had 'solved' the problem completely. This is equivalent to an individual taking in each test case and perfectly returning the expected output, with some leeway for less-precise fitness methods. When that solution is reached each experiment stops and begins again with a new population, repeating. After 5000 generations, if no ideal solution was found, the experiment would end prematurely and again a new population would form.[1] After 100000 generations running the experiment repeatedly, that test case was considered done and statistics that were tracked over the course of the test case were averaged and used to calculate standard deviations. The statistics tracked were as follows:

1. The mean and standard deviation of the number of generations the ideal solution takes to first evolve (if an ideal solution is found and exists).
2. The mean and standard deviation of the real execution time of the experiment per generation.
3. The mean and standard deviation of the best fitness in the population after the experiment ends.
4. The mean and standard deviation of the average fitness among all population members at the end of each experiment.

Ultimately the latter two statistics turned out to be almost always incredibly large due to very large outliers. They will be ignored for the purpose of this paper, for some problems. All problems were ran with 200 population size and 5 demes, with as many variables between method types consistent as possible for each test, excluding Sort List, Pathing, and Multiply Matrix, which required a larger population to solve in the time limit given and so were increased to a population size of 3000 with 75 demes. The following table is in the form Mean ± Standard Deviation.

Test Title	Generations/Sol	Time	Top Fitness	Av. Fitness
Pow8TGP	2943.714 ± 2210.715	2.136ms ± 874.381μs	N/A	N/A
Pow8LGP	14.743 ± 13.374	1.507ms ± 361.077μs	N/A	N/A
Pow8ENN	4594.364 ± 617.293	7.326ms ± 807.351μs	N/A	N/A
PowSumTGP	1396.472 ± 1612.882	2.352ms ± 385.309μs	N/A	N/A
PowSumLGP	864.336 ± 899.238	1.446ms ± 209.299μs	1.000 ± 0.000[2]	N/A
PowSumENN	1931.722 ± 2063.022	6.300ms ± 708.695μs	N/A	N/A
MultiplyMatrixTGP	4331.083 ± 971.135	46.802ms ± 23.171ms	3.565 ± 2.551	3.409 ± 1.499
MultiplyMatrixLGP	3784.556 ± 1190.873	4.133ms ± 4.121ms	1.654 ± 0.782	4.778 ± 0.696
MultiplyMatrixENN[3]	N/A	N/A	N/A	N/A
TransposeMatrixTGP	1242.296 ± 975.174	1.617ms ± 1.134ms	1.000 ± 0.000	2.119 ± 0.293
TransposeMatrixLGP	69.001 ± 15.763	47.688ms ± 5.788ms	1.000 ± 0.000	5.907 ± 0.395
TransposeMatrixEN	N/A	N/A	N/A	N/A
PathingTGP	1209.506 ± 1456.517	74.478ms ± 37.203ms	32.634 ± 25.424	319.504 ± 28.514
PathingLGP	164.953 ± 282.7647	109.585ms ± 68.327ms	29.353 ± 1.560	431.668 ± 65.309
PathingENN	N/A	N/A	N/A	N/A
TartarusTGP	5000	68.824ms ± 30.396ms	7.550 ± 0.740	10.061 ± 1.231
TartarusLGP	5000	82.523ms ± 24.667ms	7.400 ± 0.800	8.947 ± 1.101
TartarusENN	N/A	N/A	N/A	N/A
ReverseListTGP	5000	4.449ms ± 2.917ms	5.150 ± 0.792	5.469 ± 0.683
ReverseListLGP	1536.788 ± 637.164	61.937ms ± 14.147ms	1.000 ± 0.000	5.735 ± 0.507
ReverseListENN	N/A	N/A	N/A	N/A
SortListTGP	4549.696 ± 698.114	13.598ms ± 1.915079ms	3.000 ± 1.168	6.888 ± 1.100
SortListLGP	4174.625 ± 972.788	72.746ms ± 18.954167ms	1.652 ± 0.560	5.597 ± 0.686
SortListENN	N/A	N/A	N/A	N/A

[1]It is worth remarking that due to this, as values approach 5000 they become less accurate and more representative of a method's inability to solve a problem given our configuration.

[2]This represents that this test always reached the ideal fitness.

[3]ENNs failed to reach a solution for this problem and for problems marked similarly.

6 Analysis

Structural choices which are difficult to have represented in multiple ways proved the results of experiments always in flux over the course of the project. At the outset of the project, TGPs had a performance of at or a little below 100 average generations to solve Pow8, but more complex TGP problems had significant difficulty dealing with TGP's environment variables being accessed all by individual actions. In this format, for a TGP to access a specific environment variable they needed to evolve into the action "env#" where # was replaced with the environment variable's position. This was an issue with large problems that had upwards of a few dozen environment variables, as it was impractical to evolve into each environment variable's action. This was addressed by replacing the above schema with one where TGPs accessed environment variables through the action "env" which took a number as an argument, so the number itself could be mutated or done math on before corresponding to a specific environment position. However, this resulted in a staggering decrease for the evolution time of problems like Pow8, where there was only one environment variable.

Ultimately LGPs are the clear frontrunner from these results. LGPs outperformed TGPs and ENNs in every test in terms of generations to reach a solution. There's a significant exception that should be pointed out here, and that is that LGPs were developed *last* out of the three methods. ENNs were first. From a personal standpoint I would resolve that LGPs outperformed their competitors because when I developed LGPs I had already recognized the problems apparent in other methods and accommodated for them in the design of the method.

As an example, LGPs have baked in memory and environment access, compared to TGPs which required building upon an initial design to add those features to TGPs. As mentioned before, TGPs have very problematic environment use and their memory use is even worse, as referring to memory locations often worsens a TGP's average Generations to reach a solution from the increased number of potential actions the TGP can take.

That's the pessimistic perspective. The optimistic perspective is that LGPs outperformed their competitors because their syntax is much more flexible to mutate and crossover, and they don't suffer from the disadvantages other LGP implmentations do.

ENNs have two crossover methods, neither of which has the ability to crossover two ENNs with a different number of columns, and an overwhelming number of mutation options, most of which do very little. TGPs have a single crossover method and two mutation methods. LGPs have two crossover methods and five mutation methods, all of which have a variety of minor or major effects on the individual. LGPs have the most functional methods of mutation and crossover, and these are much easier to consider and implement because LGPs are linear where TGPs and ENNs have more complex structures.

Our specific implementation of LGPs also avoids the major problem LGPs commonly have, which prevent them from being as successful– that many formations of Individuals are completely invalid programs. LGPs are classically formed as a series of low-level code instructions, like the following:

```
x = 2 + 3
y = x * 2
z = 4 - y
```

But this syntax has the following major problem: what do you do if a variable hasn't been defined when it is accessed? If the third line above was instead $y = 4 - z$, when z did not exist, the program would be invalid and would need to be thrown out of the population. Comparatively, the LGPs implemented in GoEvo do the following:

```
add(0,2,3)
multiply(1,-1,2)
subtract(2,4,-1)
```

GoEvo's LGPs allocate to each individual a number of memory registers. These registers are the first argument to every function, and they represent what register a value should be assigned to. These values are explicitly limited such that they whenever they are assigned or changed they cannot exceed the number of memory locations the individual has. LGPs also offer special registers like -1, which as an example will always refer to the most recent register modified, making chaining operations easy to mutate into.

One of the more interesting problems to look at is the Tartarus problem. This is a specific scenario where LGPs fared only slightly better than TGPs when looking at the average best fitness over each generation, but it was much better in terms of average fitness, a difference from about 0.15 fitness for the best member and 1.1 fitness for the average fitness. This is perhaps the only problem where two methods performed significantly similarly to each other in the whole of the benchmark set, which may represent a sort of asymptotic bound that both LGPs and TGPs reached, but LGPs clumped around tighter. Surpassing 6 fitness in the Tartarus problem is known to be difficult, although possible given significant enhancements to the problem, and this similar top fitness result represents that well.

As far as analyzing more specific options goes, there are a few conclusions that can be reached, but unfortunately these conclusions are just disqualifications of certain approaches.

1. Random Pairing is always better than Alpha Pairing[14]. This makes sense, as Alpha Pairing was solely made as something to compare against Random Pairing.
2. Tournament Selection in general outperforms Proportional or Stochastic selection methods.
3. Mutation rates exceeding 15 percent are generally not helpful.
4. Tests with more test cases evolve faster in terms of generations, although not faster in terms of real execution time, which should be expected as more time is spent to run more test cases to generate fitness values per generation.
5. A minimum number of test cases of about 3 is required to avoid individuals mutating into random chance solutions or assigning their output variables to be the expected output variables without observing the input.
6. Having at least 1 elitism is always better than not having any elites. This project ended up rounding to about 5 elites per deme of size 40, but cannot conclusively say that is optimal.

One last thing of note is that regardless of TGPs general failure to achieve solutions, ENNs performed overwhelmingly worse to the point where they failed to ever reach a solution for certain problems. This is ultimately likely attributable, again, to them being developed before the need for environments and memory was clear, and even more so perhaps on account of their inability to perform many tasks which neural networks are expected to do in current fields, such as store results and loop back in on themselves, to name a few. These failures may be a result of poorly designed problem implementations for ENN specifically, however, as well.

7 Conclusion

Ultimately this project was unsuccessful in its hypotheses as the conclusion reached was the same as reached by previous studies, that LGPs perform far and away above their competitors. This project will continue, with a hope that benchmarking results like those produced by this project can be made open and publicly updated on a broader set of problems and methods to solve those problems.

7.1 Next Steps

This project lacks control-type problems, or problems where the result should already be known, and adding such problems would help confirm whether TGPs or ENNs had proper functionality to begin with, for example, to confirm results in general.

Other problems in the course of this project's time-frame were begun, but ran into implementation issues or conceptual problems. The most significant of these was the Traveling Salesman Problem, which was cut out due to GAs being one of the more efficient ways of achieving a solution– other methods were unlikely to find anything nearly as effective without just representing a GA themselves. The second most significant set of dropped problems were Nintendo Entertainment System game players, which required calling GoEvo from Lua, which requires registering functions to Lua via C, and then calling GoEvo functions from C, all of which will eventually be done but was not finished by the end of the project's current time-frame.[15] The remaining problems from the benchmark set provided by McDermott and co. [4] will be heavily weighed when considering future problems to address.

A significant difficulty when working on this project was deciding how to set various parameters to problem methods. Hypothetically, all problems have some set of ideal conditions to be run under depending on whether they are being run with a TGP, LGP, etc, but these parameters were only guessed at and estimated up until now. This is obviously sub-optimal, and a method of improving on this has already been suggested and implemented in some environments, called meta-genetic programming [5] or as I'll call it here to generalize it, meta-evolutionary computation (MEC). As one might conclude from its name, MEC uses evolutionary computation techniques on populations of evolutionary computation methods and settings. This is already done in part in Demetic Grouping. Demetic Grouping is a sort of MEC where each Deme can have individual settings.

One of the next things that should happen with this project is the implementation of a generic MEC on top of Demetic Grouping such that each deme can have as many unique settings a possible which each can be mutated over time just as individuals in a population can be mutated. Alternatively, demes could never pass individuals between themselves and selection could occur on a Group-scale instead of on a Deme scale, so that demes with poor settings that did not evolve best solutions would be removed from the population and would give room for more demes and more room within better-performing demes.

The addition of MEC would help for discovering more optimal conditions and removing more obvious sub-optimal conditions from populations, and would even allow for the evolution of multiple methods at the same time (granted it would require using a model which did not allow crossover between incompatible methods).

The next obvious step for this project will be in improving both efficiency and method versatility. This will involve steps like parallelizing crossover and perhaps replacing the neural package, which is inflexible and overly complex, with an adaptation of another open source neural network package that may already have features like memory and convolutional network structure built-in. Alternatively we may go and build something like this ourselves, wherein the first step would be to implement NEAT [6] or an algorithm similar to NEAT in Go.

In a similar vein as improving neural networks, adding more forms of GPs would always be an improvement. Further, documenting the best results so far on a publicly available webpage would be a significant step forward.

References

[1] Wolfgang Banzhaf and Brameier Markus. *Linear Genetic Programming (Genetic and Evolutionary Computation)*. Springer, 2007.

[2] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, Feb 2001.

[3] Carlos Contreras-Bolton and Victor Parada. Automatic combination of operators in a genetic algorithm to solve the traveling salesman problem. *PLoS ONE*, 10(9):1–25, 09 2015.

[4] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 791–798, New York, NY, USA, 2012. ACM.

[5] Juergen Schmidhuber. Evolutionary principles in self-referential learning.

[6] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 06 2002.

[7] Adrian Trenaman. Concurrent genetic programming, tartarus and dancing agents. In *Proceedings of the Second European Workshop on Genetic Programming*, pages 270–282, London, UK, UK, 1999. Springer-Verlag.

[8] Garnett Wilson and Wolfgang Banzhaf. A comparison of cartesian genetic programming and linear genetic programming. *EuroGP*, pages 182–193, 2008.

Notes

[10]For more details, as well as all code and all algorithms implemented for this project, see bitbucket.org/stephenpatrick/goevo

[11]This is a very trivial problem, but we are interested if giving the programs memory helps them evolve to produce these answers faster, recursively. The y values for this problem will be restricted to interesting values that do not risk integer overflow.

[12]This is intended to test if the program can come to proven conclusions about these statements, i.e. $\Sigma_i^x i^1 = \frac{n(n-1)}{2}$ or if it is easier for the program to recursively produce the answer.

[13]Given a grid with 4 boxes and 1 agent, move the agent such that it pushes all the boxes into the corners of the grid, or against the walls of the grid for less points. The agent can only look at the grid location in front of it, turn right and move forward, and is limited to 80 moves. The problem may be modified to give the agent more sensors, change the number of boxes, or change the max number of moves.

[14]Alpha Pairing is the practice of making every pair in the crossover process with at least one individual from some top percentage of the population's fitness ranking.

[15]While there are several projects out there which allow Lua to be called from Go, or Lua states to be generated from Go, there are no Lua projects to my knowledge which make running Go code from a process which begins as Lua easier.