Hello-Genetic-Algorithms

November 29, 2017

1 Hello Genetic Programming

1.0.1 Definitions

- **Genetic Algorithms** (GA) are search algorithms that mimic the process of natural evolution (blah blah blah) where each individual is a candidate solution: individuals are generally raw data in whatever encoding format has been defined.
- **Genetic Programming (GP)** is considered a special case of GA, where each individual is a computer program (not just raw data). GP explore the algorithmic search space and evolve computer programs to perform a defined task.

```
<div class="row">
    <div class="col-md-12" align='center'>
        <img align='center'alt="Genetic Programming" style="border-width:0" src="https://cdn-images-1.medium.com/max/1600/1
    </div>
</div>
```

1.0.2 Terms

- **Mutation**: a mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation.
- **Crossover**: is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Crossover is a process of taking more than one parent solution and producing a child solution from them.
- **Fitness**: the "fitness" is a function which takes a candidate solution to the problem as input and produces as output how "fit" our how "good" the solution is with respect to the problem in consideration.

1.0.3 Pros

- Faster (and lower memory requirements) than searching a very large search space.
- Easy, in that if your candidate representation and fitness function are correct, a solution can be found without any explicit analytical work.

1.0.4 Caveats

- GAs do not scale well with complexity (i.e. slow convergence towards solutions).
- GAs may converge towards a local optima / minima, or even arbitrary points rather than the global optimum of the problem. Could be solved by using a different fitness function to explore the solution landscape, increasing the mutation rate or modifying the selection criterion. However, there is no known general solution to this problem (i.e. the "No Free Lunch Theorem").
- GAs are difficult to work with dynamic data sets.
- GAs cannot effectively solve decision problems.
- Depending on the problem, other optimization algorithms may be better suited to evaluating and converging towards a solution than GAs.

1.0.5 Let's try to implement this in Python

First, let's setup our imports:

```
In [1]: import random
        import string
        from pprint import pprint
        from datetime import datetime


        POPULATION = 20


        # these are the characters which make up the "DNA" of our example
        GENES = string.letters + '!, '
```

Just like in biological evolution, we're applying a fitness criterion - which in this case is lexicographical distance:

```
In [2]: def calc_fitness(source, target):
            # Compare the lexicographic distance between a source and the target
            # and summate this difference squared.
            fitval = 0
            for i in xrange(len(source)):
                fitval += (ord(target[i]) - ord(source[i])) ** 2
            return fitval
```

Similarly, we also need to mutate each individual and calculate their respective fitness:

```
In [3]: def mutate(parent1, parent2, target):
            child_dna = parent1['dna'][:]  # `child_dna` is a copy of `parent1` at this point

            # now attempt cross-over
            start = random.randint(0, len(parent2['dna']) - 1)
            stop = random.randint(0, len(parent2['dna']) - 1)

            # since we're using the indices to copy over parts of each parent we
            # need the start to be less than stop. This condition ensures this by
            # switching the positions.
            if start > stop:
                stop, start = start, stop

            # now copy over the relevant parts
            child_dna[start:stop] = parent2['dna'][start:stop]

            # now mutate one position
            char_pos = random.randint(0, len(child_dna) - 1)
            #child_dna[char_pos] = GENES[random.randint(0, len(GENES) -1)]
            child_dna[char_pos] = chr(ord(child_dna[char_pos]) + random.randint(-1, 1))
            child_fitness = calc_fitness(child_dna, target)
            return {'dna': child_dna, 'fitness': child_fitness}
```

Then, we need to determine how we select individuals from the pool for mating:

```
In [4]: def random_parent(gene_pool):
            # randomly chose a parent from the pool
            index = random.randint(0, len(gene_pool)-1)
            return gene_pool[index]


        # This is a convenience function to be used for displaying the details of a given
        # genetic pool at some generation.
        def print_gene_pool(gene_pool, generation):
            for candidate in gene_pool:
                print '%i %i %s' % (generation,
                                    candidate['fitness'],
                                    ''.join(candidate['dna']))
            print
```

Now we evolve the pool, mutating the individual members and selecting based on our fitness criterion:

```
In [6]: random.seed(314159265)  # for reproducibility
        target = 'Hello, world!'

        # create a starting pool of dna
        gene_pool = list()
        for i in xrange(0, POPULATION):
            dna = [random.choice(GENES) for g in xrange(0, len(target))]
            fitness = calc_fitness(dna, target)
            candidate = {'dna': dna, 'fitness': fitness}
            gene_pool.append(candidate)


        # Evolve the pool until one the candidates reaches the
        generation = 1
        silent = True
        fittest = list()
        while True:
            # sort the gene pool by fitness in ascending order. This means that the
            # 1st element (index 0) is always the fittest individual
            gene_pool.sort(key=lambda candidate: candidate['fitness'])
            fittest.append(gene_pool[0])
            if not silent:
                print_gene_pool(gene_pool, generation)

            parent1 = random_parent(gene_pool)
            parent2 = random_parent(gene_pool)

            child = mutate(parent1, parent2, target)

            # compare the fitness of the child relative to that of the pool
            # and if the child's fitness is less than most unfit individual in the
            # pool, it's added to the pool - i.e. genetic elitism.
            if child['fitness'] < gene_pool[-1]['fitness']:
```

```
            gene_pool[-1] = child

        # terminate when we've reached the goal (fitness = 0)
        if gene_pool[0]['fitness'] == 0:
            break

        generation += 1

    # display the fittest individuals
    for generation, best in enumerate(fittest):
        if not silent:
            print generation+1, ''.join(best['dna']), best['fitness']
```

2 Activity: What happens when we etremify the genetic mutations?

3 Activity: What happens when we mutate more than one character?

In [ ]: