Trevor Brooks

Michele Van Dyne

CSCI 446 – Artificial Intelligence

November 29, 2017

Genetic Algorithms: Use, Concepts, and Project Review

Introduction

Genetic algorithms are a machine learning technique designed to mimic the evolution of species that have evolved over thousands of years. This method performs well for problems that can be represented with a chromosome and can be evaluated for how fit they are for the problem space. Genetic algorithms are a part of evolutionary computation that is especially suited for minimization or maximization problems (Carr, 2014). All code referenced in this paper was written by Trevor Brooks and can be found on Github. The code snippets below does not include all parts of the needed files. The linked code uses Python 3 to run and is not backwards compatible.

Parts of a Genetic Algorithm

There are five main sections to a genetic algorithm, although these can be somewhat collapsed. The five sections of a genetic algorithm are: the population, individuals, representative chromosomes, fitness, and the selection and reproduction methods. Each of these representations not only help break the problem down for an object oriented approach, but delegate concerns to better mimic nature. Each of these sections is broken down in the following sections with both a description of responsibilities, as well as some of the code snippets from the project. The full files can be found on Github in the link in the Introduction.

Chromosomes and Their Representations

Genetic algorithms are based off of the idea of genes and chromosomes. Each gene can take on a specified set or range of values (dependent on implementation, discussed later). After the genes are generated they can be put together to form a cohesive chromosome that represents the whole problem state. Each individual has a chromosome, and together in a population you can represent a range of possibilities. The largest decision for the chromosome (outside of fitness and reproduction issues) is the representation. A common and somewhat classic way of representing these is with binary chromosomes. Each gene is comprised one to many bits dependent on the number of possibilities that needs represented. Genes can also be represented with integer or floating point numbers. The representation must be somewhat consistent per gene to allow for crossover.

```
    def randomize(self):
        return (str(bin(random.randrange(0, self.size)))[2:]).zfill(self.length)
    def set(self, start, size):
        self.start = start
        self.size = size
        self.length = math.ceil(math.log(size, 2))
        return self.length
```

Listing 1: Randomization and setup of a gene.

Before randomization can take place, though, the genes and possible genotypes must be set up. For this project, the genes are constructed as binary strings and are tracked as discrete values. The total length of the chromosome is the total summation of the length of the genes that make it up.

```
    #Generates the length of the chromosome given needed data values to be represented.
    def generate_chromosome_lengths(self, discrete_values):
```

```
        self.discrete_values = discrete_values
        total_bits = 0
        for value in discrete_values:
            gene = Gene()
            total_bits += gene.set(total_bits, len(discrete_values[value]))
            self.__genes.append(gene)
            gene.values = discrete_values[value]
        try:
            float(self.__genes[0].values[0])
            self.is_normalized = False
        except ValueError:
            self.is_normalized = True
```

Listing 2: Generation of the genes and chromosome

For the binary representation to work, the data must be discretized, otherwise decimal position and precision would need to be accounted for, or a switch to a different representation would need to be made. The closest lower value to the input is used to change the input to a matching value. Normalization is performed once on the data before the tests are ran to reduce runtime on Deer Hunter such that the data is not processed repeatedly.

```
    # Retrieves the normalized gene value for a non-normalized data value.
    def get_normalized(self, index, value):
        if self.is_normalized:
            return value
        else:
            sorted_gene = sorted(self.get_gene(index).values)
            for gene_value in sorted_gene:
                if value <= gene_value:
                    return gene_value
```

```
10              return sorted_gene[len(sorted_gene)-1]
```

Listing 3: Normalization of numeric data reduce number of gene values

Other Representations of Chromosomes

Binary strings were picked for this project due to the examples that could be found on Obitko's website (Obitko, 1998) and in Jenna Carr's paper (Carr, 2014). Other common, and potentially more useful and applicable encodings for the chromosome include permutation encodings, value encodings, and tree encodings. Each of these can be used for different purposes, including representing either an ordering (permutation encoding), showing named discrete values or continuous values (value encoding), and more complex logic or equations (tree encoding, as well as the problem can be represented well as a tree). A more readable version of this application could have used value encoding to approach the problem. This would have also allowed for better tracking of numeric data in Deer Hunter, possibly allowing for better results. A couple of common applications of genetic algorithms and the standard encodings they use are discussed later. Representations of problems in genetic algorithms can be relatively flexible, as long as the representation is consistent, fits the data, and can be evaluated fairly by the fitness function to allow for reproduction and prediction.

Individuals

The next step up from a chromosome is an individual, which essentially the representation of a single chromosome state that currently exists in the population. Each individual can be used to represent a possible solution to the issue and is rated on fitness. Individuals are the functional representation of this state and as such can be mutated, selected, have cross-over, evaluated for fitness, and also preserved between generations of the population.

Populations

Evolution happens over generations on a population in nature. As genetic algorithms try to mimic this, there is a similar relation in the representation of the problem in genetic algorithms. A set of individuals are used to create a population, which can be reproduced, mutated, and used to predict an outcome. Populations can also have their fittest preserved across generations to help make sure that the population continues in the correct direction. Saving a part of the population can also help reduce the number of individuals that need to be generated.

```
    # Generates next generation until the population hits Size
    def generate_next_gen(self):
        next_gen  = Population()
        population = self
        if self.preserve_fittest == True:
            population = Population()
            for i in range(0, self.population_max // 2):
                fittest = self.pop_fittest_individual()
                population.add_member(fittest)
                next_gen.add_member(fittest)
        next_gen.chromosome = self.chromosome
        while next_gen.get_num_members() < self.population_max:
            child1, child2 = next_gen.reproduce(population)
            next_gen.add_member(child1)
            next_gen.add_member(child2)
        return next_gen
```

Listing 4: Generation of the next generation from an existing population

Fitness

Fitness is the cornerstone operation of a genetic algorithm. Evolution causes the best individuals to have a better chance to survive and reproduce which helps ensure a longer lasting species. Similarly, genetic algorithms measure fitness to make sure that the generated chromosomes match the problem set that is being worked with. The fitness function should represent how close the individual is to the expected outcome for its given set of genes. Fitness is primarily used in the creation of new generations, and may or may not be used in the selection of a candidate solution, dependent on implementation.

```
1    #Calculate the fitness of the individual given the training data.
2    def calc_fitness(self, test_data, chromosome, header):
3        max = -1;
4        for data in test_data:
5            current = 0
6            for i in range(0, len(header)):
7                gene = chromosome.get_gene(i)
8                #switch binary to set value
9                gene_value = int(self.binary_chromosome[gene.start:gene.start+
     gene.length], 2)
10               if chromosome.get_normalized(i, data[i]) == gene.values[
     gene_value]:
11                   current += 1 / (chromosome.get_num_genes())
12
13           if current == 1:
14               self.fitness = current
15               return current
16           #set max if better
17           else:
18               max = max if max > current else current
19       self.fitness = max
```

```
20          return self.fitness
```

Listing 5: Generalized fitness function for normalized datasets

Selection, Reproduction, and Mutation

The first part of creating a new generation after performing fitness analysis is to go through selection. Selection is based on two different models, depending on if the top members are preserved. If the top members are not preserved each individual has a chance to be picked as a ratio of their fitness over the total fitness of all individuals. If the top members are preserved, then the top half is kept as is, then the selection only happens from those individuals. This has the added benefit of ensuring that the maximum fitness is not lost.

```
1     #Selects a member of the population for reproduction, based on fitness.
2     def selection(self, population):
3         totalFitness = population.get_average_fitness() * len(population.
    __members)
4         selectCnt = random.random() * totalFitness
5         individual = None
6         for member in self.__members:
7             selectCnt -= member.get_fitness()
8             if selectCnt <= 0:
9                 individual = member
10                break
11        return individual
```

Listing 6: Selection of individuals for reproduction

For reproduction, two individuals are selected using the method above and then a crossover point is selected. As in biological crossover, the genes on either side of the crossover point are switched. This method uses the two parents to create two new children. For example, if the two

parents were represented by [0110] and [1001] and the crossover point was the middle (this is not a requisite, the crossover is randomly selected) the children would be [0101] and [1010].

```
    #Selects two individuals, does crossover, and calls mutation
    def reproduce(self, population):
        individual1 = population.selection(population)
        individual2 = population.selection(population)
        crossover_point = random.randrange(0, self.chromosome.get_num_genes())
        crossover_index = self.chromosome.get_gene(crossover_point).start
        child1 = Individual()
        child2 = Individual()
        child1.binary_chromosome = individual1.binary_chromosome[0:
    crossover_index] + individual2.binary_chromosome[crossover_index:]
        child2.binary_chromosome = individual2.binary_chromosome[0:
    crossover_index] + individual1.binary_chromosome[crossover_index:]
        if(random.random() < self.mutation_probability):
            child1 = self.mutate(child1)
        if(random.random() < self.mutation_probability):
            child2 = self.mutate(child2)
        return child1, child2
```

Listing 7: Reproduction method. Uses selection and mutation as well as crossover of genes.

Mutation allows for the introduction of gene values that were not initially available, or that may have been lost during selection. Mutation continues the model of biological genetics by allowing changes outside of selection and crossover. The low percentage chance (in this application's case: 0.25%) of mutation ensures that changes can happen but do not cause loss of the current fitness in the general case.

```
    #Mutates random gene
    def mutate(self, individual):
        individual = individual.mutate(self.chromosome)
```

```
        return individual
```

Listing 8: Select a gene and then mutate it by one value

Predictions

Predictions are the end goal of most, if not all, machine learning techniques. In the case of genetic algorithms the final population is used to determine a best fit for the given problem. In some cases you can also grow more populations after training to better fit the input. The method used in the code below does not grow new populations, but only finds the best possible match out of the final population to supply an expected result.

```
    def get_prediction(self, header, data):
        ans = ''
        best_data_fitness = -1
        best_test_fitness = -1
        for member in self.__members:
            current = 0
            for i in range(0, len(header)-1):
                gene = self.chromosome.get_gene(i)
                #switch binary to set value
                gene_value = int(member.binary_chromosome[gene.start:gene.
    start+gene.length], 2)
                if self.chromosome.get_normalized(i, data[i]) == gene.values[
    gene_value]:
                    current += 1 / (self.chromosome.get_num_genes() -1)
            if current > best_data_fitness: # and  member.get_fitness() >
    best_test_fitness:
                gene = self.chromosome.get_gene(len(header)-1)
                gene_value = int(member.binary_chromosome[gene.start:gene.
    start+gene.length], 2)
```

```
16                  ans = gene.values[gene_value]
17                  best_data_fitness = current
18                  best_test_fitness = member.get_fitness()
19          return ans
```

Listing 9: A method that fits the best individual to given case.

This project also assumes a few things about the data files being passed to it for prediction purposes, such as the last, and only the last, entry of each line being the end result. Some problems and chromosomes will have not only multiple genes representing a situation or input, but could also have multiple separate outcomes that are all happening simultaneously. This would impact all parts of the operation of the system, from the design of the chromosomes, reproduction, fitness functions, and the prediction given. For a problem where multitasking or separate simultaneous outcomes are achieved this would need rework, as well as very careful consideration.

Common Application Fields

As was mentioned in the introduction, genetic algorithms are especially suited for minimization and maximization problems. Due to this there are some common problems that are solved with genetic algorithms to get a good solution fast where a provably optimal solution may be restrictively hard to compute. A couple of examples of this are Travelling Salesman Problem and the Knapsack problem.

Knapsack is a problem that maximizes value while staying at or under a given weight. Knapsack can be easily encoded as a binary string (Obitko, 1998). The binary string can show whether or not the n^{th} item is in the knapsack by using a 1 or 0 respectively. The fitness function can then track the total weight and value, as well as account for if the total weight is too high for the given problem.

Travelling Salesperson gives a different view of encoding, however. The classic encoding for

Travelling Salesperson is permutation encoding (Obitko, 1998). This is completed by each gene representing the next city in the salesperson's path. The main differentiation for this method is that each gene must have a unique value across the chromosome. This can lead to complications with crossover and mutation, where the values changing or crossing over would cause duplicates. This must be solved by further swaps to create a consistent string. Fitness can be calculated as the distance of the path.

Project Results

The goal of this project was to read through three data files and be able to predict new outcomes. The first two files 'class.csv' and 'weather.csv' both contain twenty or less lines of normalized data. 'DeerHunter.csv' on the other hand has continuous data and over 6000 lines of data. For the smaller datasets there was some variability due to the random start that affected the results. For Deer Hunter, the results typically hovered around fifty-seven percent for the most optimal setting. Figure 1 shows three runs of the provided test sets with the program running ten generations with a mutation rate of 0.25%, and a population size of 100. Variations on each of these parameters effect the end results. For the smaller datasets, increasing the number of generations or the population size can cause overfitting, where Deer Hunter does not overfit until closer to 15 generations. Randomization within the initial population, crossover points, and selection causes the variations within the results.

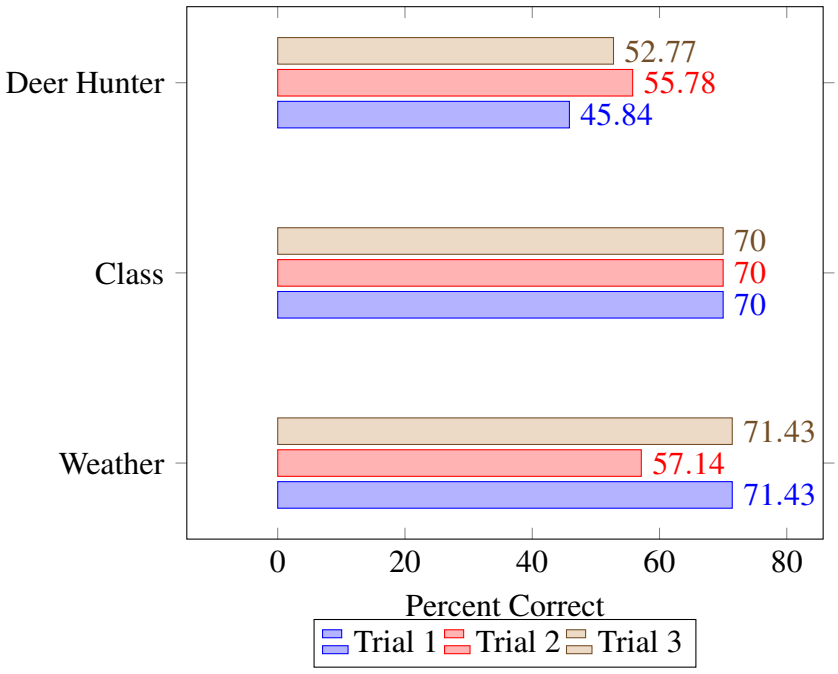

Figure 1: 10 Generations, .0025 Mutation Rate, 100 Population

Needed Improvements and Reflections

The results of this project were somewhat shaky for the smaller datasets, showing that with minimal data to fit, the population had issues converging with the data without overfitting which also causes issues. For Deer Hunter (a large, noisy, and continuous dataset) the results were somewhat more reliable, but still did not get over sixty percent reliably. This shows some of the importance of a specific fitness function that represents and evaluates the given data well. The fitness function is the key to generating successive generations, due to this a generic fitness function may get better than average (better than fifty percent for Deer Hunter) results, but will not be able to get much higher. Due to this, with more time the best optimization for this project would be to break out the fitness functions to more specific implementations for known datasets. A final consideration would be the option to allow new generations to be created until a perfect match could be found for the inputs. This may also require a different fitness function for selection of this solution. The parameters already accounted for that could be further adjusted

would be population size and mutation rate. Allowing for a bigger population would allow for the algorithm to express more possibilities at any given point, this would not be helpful within the smaller two datasets, but within Deer Hunter this may have been somewhat helpful, but would require more time per generation. Mutation rate also may be able to be moved to a higher number for Deer Hunter. Values up to .0075 and .01 were tried and the results were inconclusive if they were of any help. Further time ensuring optimal mutation rates could have been beneficial, but if there is a sweet spot, it is likely a small window.

Conclusion

Genetic algorithms provide a way to learn about a given problem set given a representative training set, and a fitness function that properly fits the problem. Genetic algorithms are very flexible in their representation, but are very particular in that the representation and fitness function must match the problem very well. This is apparent within the results that were gathered in this project with a generalized method for creating a chromosome and evaluation with a fitness function.

References

Carr, J. (2014). An introduction to genetic algorithms. Retrieved from https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf

Obitko, M. (1998). X. encoding. Retrieved from https://courses.cs.washington.edu/courses/cse473/06sp/GeneticAlgDemo/encoding.html