Genetic Algorithms

Trevor Brooks

CSCI 446 – Fall 2017

November 27th, 2017

Introduction to Genetic Algorithms

- Genes
- Chromosomes
- Individuals / Populations
- Selection, Reproduction, Mutation
- Fitness Functions

Chromosomes / Genes

- Chromosomes are a set of genes
- Each gene represents an input or an output
- There are multiple representations that can be used:
  - Binary Strings
  - Value
  - Trees
  - Permutations

[2,3,4]

```
#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
```

[1]

```
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
```

[1]

Populations

- Populations are a set of individuals
- Tracks all current knowledge / candidates
- Will have many populations over a series of generations

[2]

```python
# 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
```

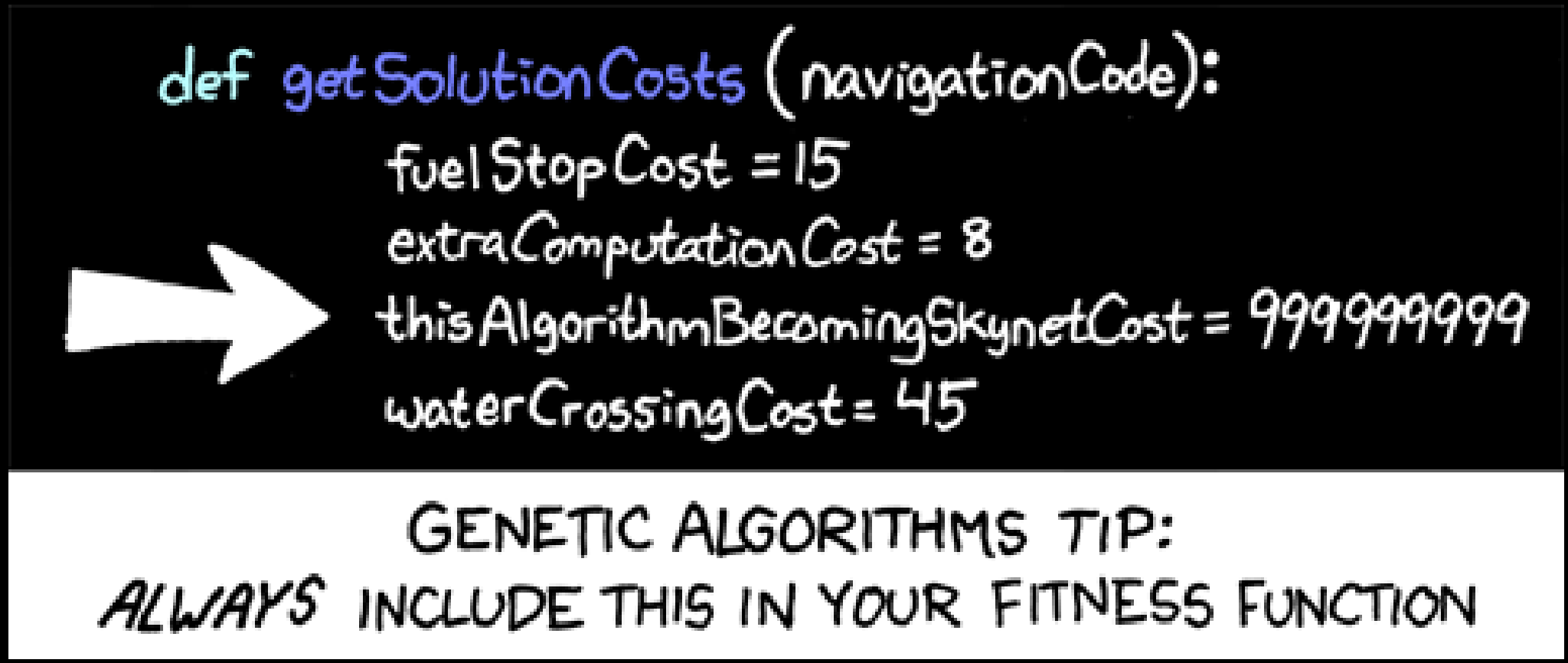
[1]

Fitness

- Tracking how close a solution is to the expected or correct value (supervised)
- Tracking result as a minimum or maximum, with some exceptions (Unsupervised)

- Drives selection for reproduction, tracks status of population

[2]



https://xkcd.com/534/

```python
#Calculate the fitness of the individual given the training data.
def calc_fitness(self, test_data, chromosome, header):
    max = -1;
    for data in test_data:
        current = 0
        for i in range(0, len(header)):
            gene = chromosome.get_gene(i)
            #switch binary to set value
            gene_value = int(self.binary_chromosome[gene.start:gene.start+gene.length], 2)
            if chromosome.get_normalized(i, data[i]) == gene.values[gene_value]:
                current += 1 / (chromosome.get_num_genes())

        if current == 1:
            self.fitness = current
            return current
        #set max if better
        else:
            max = max if max > current else current
    self.fitness = max
    return self.fitness
```

[1]

Reproduction

- Reproduction allows for crossover and mutations
- Uses fitness for selection
- Should drive population closer (in general) to a solution

```python
#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
```

[1]

```python
#Selects a member of the population for reproduction, based on fitness.
def selection(self, population):
    totalFitness = population.get_average_fitness() * len(population.__members)
    selectCnt = random.random() * totalFitness
    individual = None
    for member in self.__members:
        selectCnt -= member.get_fitness()
        if selectCnt <= 0:
            individual = member
            break
    return individual
```

[1]

```
# Perform mutation on individual. Called from population.
def mutate(self, chromosome):
    mutated_gene = chromosome.get_gene(random.randrange(0, chromosome.get_num_genes()))
    start = mutated_gene.start
    end = mutated_gene.length + start
    size = mutated_gene.size
    gene_value = self.binary_chromosome[start:end]

    if int(gene_value, 2) + 1 < size:
        new_gene = (str(bin(int(gene_value, 2) + 1))[2:]).zfill(end-start)
    else:
        new_gene = (str(bin(int(gene_value, 2) - 1))[2:]).zfill(end-start)
    self.binary_chromosome = self.binary_chromosome[0:start] + new_gene + self.binary_chromosome[end:]
    return self
```

[1]

Considerations

- Normalization
- Prediction Method
- Fitness

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

[1]

```
# Predicts result given data of scenario. Only called after training.
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)
            ans = gene.values[gene_value]
            best_data_fitness = current
            best_test_fitness = member.get_fitness()
    return ans
```

[1]

Results

	Trial 1	Trial 2	Trial 3	Average
Weather	71.43	57.14	71.43	66.66
Class	70	70	70	70
Deer Hunter	45.84	55.78	52.77	51.43

Generations: 10
Mutation Rate: .0025 (0.25%)
Population Size: 100

Possible Enhancements

- Customized fitness functions
- Dial in the mutation rate per problem
- Adjust population size and number of generations per problem

Common Sample Applications

- Minimization and maximization problems
- Travelling Salesperson
  - Can track order of cities by having each gene be an integer number that is a city.
  - Swapping cities around using crossover or mutation causes extra work to be necessary
- Knapsack
  - Binary chromosome (whether or not something was picked)
  - Fitness is evaluated as expected
    - Value and weight

[2,4]

Other applications

- Scheduling
- Fraud Detection
- Product creation (processors, etc)

[2]

Conclusion

- Copies a working model from nature
- Main component for decision making is the fitness function
- Well-suited for minimization or maximization problems

References

[1]Brooks, T. (2017). Genetic algorithm. Retrieved from https://github.com/trevorlbrooks/genetic-algorithm

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

[3] Chromosome (genetic algorithm). (2016). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Chromosome_(genetic_algorithm) [3]

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