Genetic Algorithms

Alexander D Brown (adb9)

October 4, 2013

Contents

1 Introduction

Genetic algorithms are a biologically-inspired approach to heuristic search which mimic natural selection. Unlike many other evolutionary strategies and evolutionary programming, they are not designed to solve a specific problem, but are designed to solve the problem of optimisation which is made difficult by substantial complexity and uncertainty[2].

The complexity of the task should make it such that discovering an optimum solution is a long, maybe even impossible, task. At the same time the uncertainty needs to be reduced so that the knowledge of *available* options can be increased.

The initial design for a genetic algorithm was a method for moving from one population of chromosomes to another using a form of natural selection. This algorithm also included methods for crossover, mutation and inversion. This idea of having a large population was the distinguishing feature from any past attempts which had only considered the parent and one offspring, where the offspring was simply a mutation of the parent[3].

1.1 Evolutionary Algorithms

As their name suggests, an evolutionary algorithm applies elements from the biological theory of evolution to the problem of optimisation. These elements include:

- Reproduction

- Mutation

- Recombination

- Selection

Typically, a population of candidate solutions are generated to which a fitness function can be applied. The population is then subject to some form of evolution, and this process is repeated until a halting criteria is met.

Genetic algorithms are a type of evolutionary algorithm with a focus on the genetic evolution of solutions. Candidate solutions for genetic algorithms, known as *chromosomes* are encoded as a series of *genes*. These genes are a representation of the choices which need to be optimised for the solution and can be as simple as a single bit or as complex as a real number, depending on the problem.

There are many other forms of both evolutionary and genetic algorithms which this report with mention in later sections.

2 Basic Genetic Algorithm Principals

The basic principals of genetic algorithms are to represent candidate solutions as a population of chromosomes, from this population the fittest members can be picked out and used in the next generation and to create new member of the population through reproduction and/or mutation.

This cycle repeats with the aim to produce better performing individuals in each generation until the optimum solution is either reached or gotten close enough to that any future improvement is unnecessary or unwanted due to other constraints such as processing time. The latter of these allows a genetic algorithm to come up with a "good" solution in a reasonable amount of time.
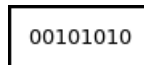
Reproduction and mutation are an important part of genetic programming, and too of evolutionary programming. Without these parts the algorithm would quickly reach a local optimum for the initial population and would not improve past this.

2.1 Chromosome Representation

One of the key parts parts in implementing a genetic algorithm is the representation of chromosomes. This is very dependent of the problem the genetic algorithm needs to optimise and can have a knock on affect on the efficiency and accuracy of the algorithm.

Sometimes a simple solution is enough to represent the problem, binary strings are a commonly suggested approach. However sometimes more complex representations are required, potentially any data structure can be used as a chromosome but lists and trees are the common choices as they are easy to perform crossover[1] and mutation on.

As a very simple example, to maximise y in: $y = f(x)$, one could represent the value of x as a binary string, an example of which is shown in figure 1.

```
00101010
```

Figure 1: Chromosome representation as a binary string

2.2 Fitness Function

To fulfil the step of natural selection; the process of choosing the "best" members of a population, there needs to be a way of evaluating each chromosome, such that they can be compared to one another.

The function for doing so is known as a fitness function, which typically returns either a single number or a list of numbers, depending on the problem. Each chromosome can then be ranked in order of fitness and the top members of a population can then be selected.

As the value returned from a fitness function, with be specific to the domain it has be used within, it is necessary to rescale the fitness value to ensure uniformity when genetic algorithms are applied over several different domains at the same time.

---

[1]A term used instead of reproduction in genetic algorithms.

2.3 Selection

Selection simulates the "survival of the fittest" nature of biology. However, it is beneficial not to keep some lesser performing members of the population to avoid getting trapped in local optimum. Most selection algorithms will introduce an element of randomness into the selection to deal with this.

This report will discuss the effects of different selection algorithms in Section 3.

3   The Effects of Selection Algorithms on GAs

The selection of the "best" individuals is an important step in genetic algorithms and as such several selection algorithms have been put forward.

The simplest is to just pick the top individuals of a generation to carry on their genes in the next generation, either through directly copying them, crossover or mutation.

This has its flaws and doesn't mirror the biological process of evolution. The main flaw of this approach is that it can be beneficial to include weaker members of a population to avoid getting stuck in local optimum. By including other, seemingly worse, members of a population, the effects of crossover and mutation can overcome this problem.

Another, commonly used, technique is to use a tournament-based selection algorithm, which involves selecting the best member from a pool made up of random members of the population. The size of the pool has a big effect on the selection of individuals, and is generally inversely proportional to the number of weaker members selected.

Tournament selection is one of the closest to biological natural selection, and has the benefits that it is efficient, especially on parallel architectures.

Another popular selection algorithm is roulette-wheel selection, where the chance a particular individual has of being selected is proportional to its fitness. Those with a higher fitness have a higher probability of being selected. The equation used to calculate the probability is shown in equation 1, where i is the individual in question, f_x is the fitness of individual x and N is the number of individuals in the population.

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{1}$$

There have been studies into the effects of different selection algorithms on genetic algorithms. Goldberg and Deb[1] compared four methods of selection; proportional reproduction, ranking selection, tournament selection and the GENITOR algorithm.

They analysed each selection scheme in terms of:

1. growth ratio; the expected ratio for the members of the best class to the umber of members of the population.

2. Takeover time; the approximate number of generations it takes to converge, this gives an idea of how long it would take before mutation, rather than crossover, becomes the primary method for exploring the search space.

3. Time complexity; the standard big O measure for time complexity.

It was found that linear ranking and binary tournament (a tournament of only two individuals) selection had similar growth rates, but that tournament selection with larger pool sizes had higher growth rates, though a non-linear ranking function can also achieve this.

Takeover times all converged in around $O(log\ n)$ generations. Most interestingly the time complexities had a large range, from $O(n^2)$ to $O(n)$, with

tournament selection being an $O(n)$. This is particularly interesting as tournament selection is a very easy algorithm to make parallel, this fits in with genetic algorithms nicely as they improve drastically from parallel implementation.

These results should be taken with some scepticism as they are formulated from the pure mathematics of the selection techniques and not from running experiments and the authors even mention that it is only designed as a simple method to better understand the *expected* behaviour and suggest looking at better methods, citing the inherently noisy nature of genetic algorithms.

3.1 Genetic Drift

4 Code Example

A simple python implementation is shown in figure 1.

```python
import random, math, time
from graph import Graph

MUTATION = 0.001
CROSSOVER = 0.6

class Chromosome:
    @classmethod
    def create(cls, graph):
        genes = [i for i in xrange(graph.num_nodes())]
        random.shuffle(genes)
        return Chromosome(genes, graph)

    def __init__(self, genes, graph):
        while not self.valid(genes):
            genes = self.make_valid(genes)
        self.genes = genes
        self.graph = graph

    def valid(self, genes):
        for i in xrange(len(genes)):
            if i not in genes:
                return False
        return True

    def make_valid(self, genes):
        missing = [i for i in xrange(len(genes)) if i not in genes]
        double = [i for i in xrange(len(genes)) if genes.count(i) > 1]

        random.shuffle(missing)
        random.shuffle(double)

        for i in double:
            genes[genes.index(i)] = missing.pop()
        return genes

    def __str__(self):
        return "{} - {}".format(self.genes, Chromosome.fitness(self))

    def __repr__(self):
        return self.__str__()

    @classmethod
    def fitness(cls, c):
        return sum([graph.distance(n1, n2)
                    for (n1, n2) in zip(c.genes, c.genes[1:])])


class BasicGA(object):
    def __init__(self, graph, population):
        self.graph = graph
        self.pop_size = len(population)
        self.population = population
        self.mutants_rate = int(math.ceil(self.pop_size * MUTATION))
        self.crossover_rate = int(math.ceil(self.pop_size * CROSSOVER))

    def run(self):
        """Runs the GA once."""
        best = self.select()
```

```python
        offspring = self.crossover(best)
        mutants = self.mutate(best)
        cur_best = self.population[0]
        self.population = best + offspring + mutants
        return cur_best

    def select(self):
        """Selects the best individuals from a population."""
        retained_pop = self.pop_size - self.mutants_rate - self.
        crossover_rate
        return sorted(self.population, key=Chromosome.fitness)[0:
        retained_pop]

    def crossover(self, population):
        """Performs crossover on a certain amount of the population."""
        return [self.perform_crossover(population)
                for i in xrange(self.crossover_rate)]

    def perform_crossover(self, population):
        """
        Perform crossover by:
        - Choosing a crossover point randomly.
        - Choosing two parents.
        - Slicing the genes of both parents together at the crossover
        point.
        """
        point = int(random.random() * self.graph.num_nodes())
        p1 = random.choice(population)
        p2 = random.choice(population)
        return Chromosome(p1.genes[:point] + p2.genes[point:], self.
        graph)

    def mutate(self, population):
        """Performs mutation of a certain number of the population."""
        return [self.perform_mutation(population)
                for i in xrange(self.mutants_rate)]

    def perform_mutation(self, population):
        """Randomly swap two genes to mutate the chromosome."""
        target = random.choice(population)
        genes = [i for i in target.genes]
        pos1 = int(random.random() * len(genes))
        pos2 = int(random.random() * len(genes))
        temp = genes[pos1]
        genes[pos1] = genes[pos2]
        genes[pos2] = temp
        return Chromosome(genes, self.graph)

class TournamentGA(BasicGA):
    def __init__(self, graph, num, tournament_size):
        super(TournamentGA, self).__init__(graph, num)
        self.tournament_size = tournament_size

    def select(self):
        """
        Selects the best members of the population using a tournament-
        based
        algorithm.
        """
        retained_pop = self.pop_size - self.mutants_rate - self.
        crossover_rate
        temp = self.population
```

```python
      best = []
      while len(best) != retained_pop:
        tournament = random.sample(temp, min(self.tournament_size,
      len(temp)))
        b = min(tournament, key=Chromosome.fitness)
        best.append(b)
      return best


if __name__ == "__main__":
  graph = Graph(25, 10, seed="genetic-algorithm")

  # Reset the random seed from graph creation.
  #random.seed()
  pop = [Chromosome.create(graph) for _ in xrange(100)]
  gas = [BasicGA(graph, pop), TournamentGA(graph, pop, 2),
    TournamentGA(graph, pop, 10)]

  for ga in gas:
    avg = []
    improving = True
    runs = 0
    while improving:
      runs += 1
      best = ga.run()
      avg.append(Chromosome.fitness(best))
      if len(avg) > 100:
        median = avg[len(avg)/2 + 1]
        improving = Chromosome.fitness(best) != median and len(avg)
     < 10000
    print "{} found best: {} in {} runs".format(ga.__class__.
    __name__,
                                                str(best),
                                                runs)
```

Listing 1: A Python implementation of a simple Genetic Algorithm

```python
import random

class Node:
  def __init__(self, id, graph):
    self.x = int(random.random() * graph.size)
    self.y = int(random.random() * graph.size)
    self.id = id

  def distance(self, other):
    return pow(self.x - other.x, 2) + pow(self.y - other.y, 2)

  def __str__(self):
    return "({}, {})".format(self.x, self.y)

class Graph:
  nodes = None
  def __init__(self, nodes, size, **kwargs):
    if 'seed' in kwargs:
      random.seed(kwargs['seed'])
    self.size = size
    self.nodes = [Node(i, self) for i in xrange(nodes)]

  def distance(self, n1, n2):
    return self.nodes[n1].distance(self.nodes[n2])

  def num_nodes(self):
```

```
    return len(self.nodes)

def __str__(self):
    s = ""
    for x in xrange(0, self.size):
        for y in xrange(0, self.size):
            added = False
            for node in self.nodes:
                if node.x == x and node.y == y:
                    s += str(node.id)
                    added = True
            if not added:
                s += " "
        s += "\n"
    return s
```

Listing 2: The Graph Code for listing 1

References

[1] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93, 1991.

[2] John H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press, April 1992.

[3] Melanie Mitchell. An introduction to genetic algorithms, 1996.