Genetic Algorithms

Alexander D Brown (adb9)

October 2, 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[1].

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[2].

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.

$\boxed{00101010}$

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.

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

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.

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.

3 Code Example

A simple python implementation is shown in figure 1.

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

MUTATION = 0.06
CROSSOVER = 0.2

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):
    self.genes = genes
    self.graph = graph

  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 GA:
  def __init__(self, graph, pop_size):
    self.graph = graph
    self.pop_size = pop_size
    self.mutants_rate = int(math.ceil(pop_size * MUTATION))
    self.crossover_rate = int(math.ceil(pop_size * CROSSOVER))

  def run(self, secs):
    self.gen_pop()
    original_pop = self.population
    start_time = time.time()
    end_time = start_time + secs
    while time.time() < end_time:
      ranked = self.rank()
      best = self.select(ranked)
      offspring = self.crossover(best)
      mutants = self.mutate(best)
      self.population = best + offspring + mutants
    best = next(iter(self.rank()))
    print "Best solution is: {}".format(best)
```

```
    if best in original_pop:
      print "Best was already in original population"



  def gen_pop(self):
    self.population = [Chromosome.create(self.graph) for i in
   xrange(self.pop_size)]

  def rank(self):
    return sorted(self.population, key=Chromosome.fitness)

  def select(self, ranked):
    return self.population[0:self.pop_size - self.mutants_rate -
    self.crossover_rate]

  def crossover(self, population):
    return [self.perform_crossover(population) for i in xrange(
   self.crossover_rate)]

  def perform_crossover(self, population):
    point = int(random.random() * self.graph.num_nodes())
    p1 = random.choice(population)
    p2 = random.choice(population)
    return Chromosome(p1.genes[:point] + p2.genes[point:], graph
   )

  def mutate(self, population):
    return [population[i] for i in xrange(self.mutants_rate)]
graph = Graph(7, seed="genetic-algorithm")
random.seed()
ga = GA(graph, 100)
ga.run(10)
```

Listing 1: A Python implementation of a simple Genetic Algorithm

```
import random

BOUND=25

class Node:
  def __init__(self, id, graph):
    self.x = int(random.random() * BOUND)
    self.y = int(random.random() * BOUND)
    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, **kwargs):
    if 'seed' in kwargs:
      random.seed(kwargs['seed'])
    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, BOUND):
      for y in xrange(0, BOUND):
        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

if __name__ == "__main__":
  g = Graph(9)
  print str(g)
```

Listing 2: The Graph Code for listing 1

References

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

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