Genetic Algorithms

William H. Hsu, Kansas State University, USA

INTRODUCTION

A **genetic algorithm** (**GA**) is an algorithm used to find approximate solutions to difficult-to-solve problems through application of the principles of evolutionary biology to computer science. Genetic algorithms use biologically-derived techniques such as inheritance, mutation, natural selection, and recombination. Genetic algorithms are a particular class of evolutionary algorithms.

Genetic algorithms are typically implemented as a computer simulation in which a population of abstract representations (called *chromosomes*) of candidate solutions (called *individuals*) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0s and 1s, but different encodings are also possible. The evolution starts from a population of completely random individuals and happens in generations. In each generation, multiple individuals are stochastically selected from the current population, modified (mutated or recombined) to form a new population, which becomes current in the next iteration of the algorithm.

BACKGROUND

Operation of a GA

The problem to be solved is represented by a list of parameters which can be used to drive an evaluation procedure, called chromosomes or genomes. Chromosomes are typically represented as simple strings of data and instructions, in a manner not unlike instructions for a von Neumann machine, although a wide variety of other data structures for storing chromosomes have also been tested, with varying degrees of success in different problem domains.

Initially several such parameter lists or chromosomes are generated. This may be totally random, or the programmer may seed the gene pool with "hints" to form an initial pool of possible solutions. This is called the *first generation pool*.

During each successive generation, each organism is evaluated, and a value of *goodness* or *fitness* is returned by a fitness function. The pool is sorted, with those having better fitness (representing better solutions to the problem) ranked at the top. Notice that "better" in this context is relative, as initial solutions are all likely to be rather poor.

The next step is to generate a second generation pool of organisms, which is done using any or all of the genetic operators: selection, crossover (or recombination), and mutation. A pair of organisms are selected for breeding. Selection is biased towards elements of the initial generation which have better fitness, though it is usually not so biased that poorer elements have no chance to participate, in order to prevent the solution set from

converging too early to a sub-optimal or local solution. There are several well-defined organism selection methods; roulette wheel selection and tournament selection are popular methods.

Following selection, the *crossover* (or *recombination*) operation is performed upon the selected chromosomes. Most genetic algorithms will have a single tweakable *probability of crossover (P_c)*, typically between 0.6 and 1.0, which encodes the probability that two selected organisms will actually breed. A random number between 0 and 1 is generated, and if it falls under the crossover threshold, the organisms are mated; otherwise, they are propagated into the next generation unchanged. Crossover results in two new child chromosomes, which are added to the second generation pool. The chromosomes of the parents are mixed in some way during crossover, typically by simply swapping a portion of the underlying data structure (although other, more complex merging mechanisms have proved useful for certain types of problems.) This process is repeated with different parent organisms until there are an appropriate number of candidate solutions in the second generation pool.

The next step is to mutate the newly created offspring. Typical genetic algorithms have a fixed, very small *probability of mutation (P_m)* of perhaps 0.01 or less. A random number between 0 and 1 is generated; if it falls within the P_m range, the new child organism's chromosome is randomly mutated in some way, typically by simply randomly altering bits in the chromosome data structure.

These processes ultimately result in a second generation pool of chromosomes that is different from the initial generation. Generally the average degree of fitness will have increased by this procedure for the second generation pool, since only the best organisms from the first generation are selected for breeding. The entire process is repeated for this second generation: each organism in the second generation pool is then evaluated, the fitness value for each organism is obtained, pairs are selected for breeding, a third generation pool is generated, etc. The process is repeated until an organism is produced which gives a solution that is "good enough".

A slight variant of this method of pool generation is to allow some of the better organisms from the first generation to carry over to the second, unaltered. This form of genetic algorithm is known as an *elite selection strategy*.

MAIN THRUST OF THE CHAPTER

Observations

There are several general observations about the generation of solutions via a genetic algorithm:

- GAs may have a tendency to converge towards local solutions rather than global solutions to the problem to be solved.

- Operating on dynamic data sets is difficult, as genomes begin to converge early on towards solutions which may no longer be valid for later data. Several methods have been proposed to remedy this by increasing genetic diversity somehow and preventing early convergence, either by increasing the probability of mutation when the solution quality drops (called *triggered hypermutation*), or by occasionally introducing entirely new, randomly generated elements into the gene pool (called *random immigrants*).
- As time goes on, each generation will tend to have multiple copies of successful parameter lists, which require evaluation, and this can slow down processing.
- Selection is clearly an important genetic operator, but opinion is divided over the importance of crossover verses mutation. Some argue that crossover is the most important, while mutation is only necessary to ensure that potential solutions are not lost. Others argue that crossover in a largely uniform population only serves to propagate innovations originally found by mutation, and in a non-uniform population crossover is nearly always equivalent to a very large mutation (which is likely to be catastrophic).
- GAs are not good at finding optimal solutions, but can rapidly locate *good* solutions, even for difficult search spaces.

Variants

The simplest algorithm represents each chromosome as a bit string. Typically, numeric parameters can be represented by integers, though it is possible to use floating point representations. The basic algorithm performs crossover and mutation at the bit level.

Other variants treat the chromosome as a list of numbers which are indexes into an instruction table, nodes in a linked list, hashes, objects, or any other imaginable data structure. Crossover and mutation are performed so as to respect data element boundaries. For most data types, specific variation operators can be designed. Different chromosomal data types seem to work better or worse for different specific problem domains.

Efficiency

Genetic algorithms are known to produce good results for some problems. Their major disadvantage is that they are relatively slow, being very computationally intensive compared to other methods, such as random optimization.

Recent speed improvements have focused on speciation, where crossover can only occur if individuals are closely-enough related.

Genetic algorithms are extremely easy to adapt to parallel computing and clustering environments. One method simply treats each node as a parallel population. Organisms are then migrated from one pool to another according to various propagation techniques.

Another method, the Farmer/worker architecture, designates one node the *farmer*, responsible for organism selection and fitness assignment, and the other nodes as *workers*, responsible for recombination, mutation, and function evaluation.

Genetic Wrappers

A genetic program is ideal for implementing wrappers where parameters are naturally encoded as chromosomes such as bit strings or permutations. This is precisely the case with variable (feature subset) selection, where a bit string can denote membership in the subset, and with variable ordering, where a permutation denotes $\boldsymbol{\alpha}$, the order in which nodes are added to the BN. Both of these are methods for *inductive bias control* where the input representation is changed from the default [Be90] – here, the full subset χ or an arbitrary ordering $\boldsymbol{\alpha}_0$.

With a genetic wrapper, we seek to evolve parameter values using the performance criterion of the overall learning system as fitness. In learning to classify, this may simply mean validation set accuracy. However, as we have noted, many authors of GA-based wrappers have independently derived criteria that resemble *minimum description length* (*MDL*) estimators – that is, they seek to minimize model size and the sample complexity of input as well as maximize generalization accuracy. [CS96, RPG+97, GW99, HWRC00]

An additional benefit of genetic algorithm-based wrappers is that it can automatically calibrate "empirically determined" constants such as the coefficients *a*, *b*, and *c* introduced in the previous section. As we noted, this can be done using individual training data sets rather than assuming that a single optimum exists for a large set of machine learning problems. This is preferable to empirically calibrating parameters as if a single "best mixture" existed. Even if a very large and representative corpus of data sets were used for this purpose, there is no reason to believe that there is a single *a posteriori* optimum for genetic parameters such as weight allocation to inferential loss, model complexity, and sample complexity of data in the variable selection wrapper.

Finally, genetic wrappers can "tune themselves" – for example, the *GA-Based Inductive Learning* (*GABIL*) system of Dejong *et al* [DSG93] learns propositional rules from data and adjusts constraint parameters that control how these rules can be generalized. Mitchell notes that this is a method for evolving the learning strategy itself. [Mi97] Many classifier systems also implement performance-tuning wrappers in this way. [BGH89] Finally, population size and other constants for controlling elitism, niching, sharing, and scaling can be controlled using *parameterless GAs*. [HL99]

Problem domains

Problems which appear to be particularly appropriate for solution by genetic algorithms include timetabling and scheduling problems, and many scheduling software packages

are based on GAs. GAs have also been applied to engineering. Genetic algorithms are often applied as an approach to solve global optimization problems. Genetic algorithms have been successfully applied to the study of neurological evolution (see NeuroEvolution by Augmented Topologies).

History

John Holland was the pioneering founder of much of today's work in genetic algorithms, which has moved on from a purely theoretical subject (though based on computer modelling) to provide methods which can be used to actually solve some difficult problems today.

Pseudo-code Algorithm

```
Choose initial population
Evaluate each individual's fitness
Repeat
    Select best-ranking individuals to reproduce
    Mate pairs at random
    Apply crossover operator
    Apply mutation operator
    Evaluate each individual's fitness
Until terminating condition (see below)
```

Terminating conditions often include:

- Fixed number of generations reached
- Budgeting: allocated computation time/money used up
- An individual is found that satisfies minimum criteria
- The highest ranking individual's fitness is reaching or has reached a plateau such that successive iterations are not producing better results anymore.
- Manual inspection. May require start-and-stop ability
- Combinations of the above

FUTURE TRENDS

Related techniques

Genetic programming is a related technique developed by John Koza, in which computer programs, rather than function parameters, are optimised. Genetic programming often uses tree-based internal data structures to represent the computer programs for adaptation instead of the list, or array, structures typical of genetic algorithms. Genetic programming algorithms typically require running time that is orders of magnitude greater than that for genetic algorithms, but they may be suitable for problems that are intractable with genetic algorithms.

CONCLUSION

Genetic programming is an emerging methodology that promotes a crosscutting and integrative view. It looks across both technologies and application domains to identify and organize the techniques, tools, and models that improve data-driven discovery.

There are significant research questions as this methodology evolves. Continuing progress will be eagerly received from efforts in individual strategies for knowledge discovery and machine learning, such as the excellent contributions in (Koza, Keane, Streeter, Mydlowec, Yu, and Lanza, 2003). An additional opportunity is to pursue the recognition of unifying aspects of practices now associated with diverse disciplines. While the anticipation of new discoveries is exciting, the evolving practical application of discovery methods needs to respect individual privacy and a diverse collection of laws and regulations. Balancing these requirements constitutes a significant and persistent challenge as new concerns emerge and laws are drafted.

Looking ahead to the challenges and opportunities of the 21st century, discovery informatics is poised to help people and organizations learn as much as possible from the world's abundant and ever growing data assets.

REFERENCES

Brameier, M. & Banzhaf, W. (2001). Evolving Teams of Predictors with Linear Genetic Programming. *Genetic Programming and Evolvable Machines* **2**(4), p. 381-407.

Burke, E. K., Gustafson, S. & Kendall, G. (2004). Diversity in genetic programming: an analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation* **8**(1), p. 47-62.

Goldberg, David E (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*.

Harvey, Inman (1992), *Species Adaptation Genetic Algorithms: A basis for a continuing SAGA*, in 'Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life', F.J. Varela and P. Bourgine (eds.), MIT Press/Bradford Books, Cambridge, MA, pp. 346-354.

Keijzer, M. & Babovic, V. (2002). Declarative and Preferential Bias in GP-based Scientific Discovery. *Genetic Programming and Evolvable Machines* **3**(1), p. 41-79.

Kishore, J. K., Patnaik, L. M., Mani, V. & Agrawal, V.K. (2000). Application of genetic programming for multicategory pattern classification. *IEEE Transactions on Evolutionary Computation* **4**(3), p. 242-258.

Koza, John (1992), *Genetic Programming: On the Programing of Computers by Means of Natural Selection*

Krawiec, K. (2002). Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks. *Genetic Programming and Evolvable Machines* **3**(4), p. 329-343.

Mitchell, Melanie, (1996), *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA Addison-Wesley

Muni, D. P., Pal, N. R. & Das, J. (2004). A novel approach to design classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation* **8**(2), p. 183-196.

Nikolaev, N. Y. & Iba, H. (2001). Regularization approach to inductive genetic programming. *IEEE Transactions on Evolutionary Computation* **5**(4), p. 359-375.

Nikolaev, N. Y. & Iba, H. (2001). Accelerated Genetic Programming of Polynomials. *Genetic Programming and Evolvable Machines* **2**(3), p. 231-257.

TERMS AND THEIR DEFINITION

Clickstream: The sequence of mouse clicks executed by an individual during an online Internet session.

Data Mining: The application of analytical methods and tools to data for the purpose of identifying patterns and relationships such as classification, prediction, estimation, or affinity grouping.

Discovery Informatics: The study and practice of employing the full spectrum of computing and analytical science and technology to the singular pursuit of discovering new information by identifying and validating patterns in data.

Evolutionary Computation: Solution approach guided by biological evolution, which begins with potential solution models, then iteratively applies algorithms to find the fittest models from the set to serve as inputs to the next iteration, ultimately leading to a model that best represents the data.

Knowledge Management: The practice of transforming the intellectual assets of an organization into business value.

Neural Networks: Learning systems, designed by analogy with a simplified model of the neural connections in the brain, which can be trained to find nonlinear relationships in data.

Rule Induction: Process of learning, from cases or instances, if-then rule relationships consisting of an antecedent (if-part, defining the preconditions or coverage of the rule) and a consequent (then-part, stating a classification, prediction, or other expression of a property that holds for cases defined in the antecedent).