Whitepaper

IPROFS

Genetic Algorithms in Java

Overview

Topics

Genetic algorithms, evolutionary computing, Darwin, natural selection, survival of the fittest, fitness functions, cross-over, mutation, optimisation, artificial intelligence, expert systems, decision support systems, software engineering and architecture, algorithms, complexity, search spaces, combinatorial explosion, NP-complete problems, GA frameworks, heuristics.

Solution Overview

Generate fast and high-quality results for complex search problems where traditional, brute-force algorithms fail.

Applicable Industries

- Logistics
- Telecommunications
- Bioinformatics
- Data mining
- Planning
- Scheduling
- Resource Allocation
- Gaming

Frameworks Discussed

- GA
- Java GALib

„Living organisms are consummate problem solvers. They exhibit a versatility that puts the best computer programs to shame."

John H. Holland, founder of genetic algorithms.

In biology, natural selection [1] is the process by which favourable heritable traits become more common in successive generations of a population of reproducing organisms, and unfavourable heritable traits become less common, due to differential reproduction of genotypes. Natural selection acts on the phenotype, or the observable characteristics of an organism, such that individuals with favourable phenotypes are more likely to survive and reproduce than those with less favourable phenotypes. The phenotype's genetic basis, genotype associated with the favourable phenotype, will increase in frequency over the following generations. Over time, this process may result in adaptations that specialize organisms for particular ecological niches and may eventually result in the emergence of new species. In other words, natural selection is the mechanism by which evolution may take place in a population of a specific organism. This mechanism can also be used within the context of solving hard search problems, using genetic algorithms.

IPROFS

“Computer programs that "evolve" in ways that resemble natural selection can solve complex problems even their creators do not fully understand.”

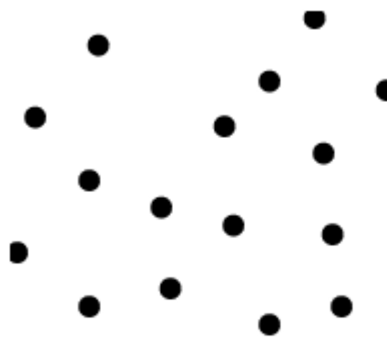
John H. Holland.

Introduction

Travelling Salesman

The Traveling Salesman Problem (TSP) is a classic search problem which focuses on the following question [2]: given a number of cities and the costs of traveling from any city to any other city, what is the least-cost round-trip route that visits each city exactly once and then returns to the starting city?

This problem may be visualized as follows:



Each dot represents a city. The question is which route a traveling salesman should follow (i.e., in which order should he or she visit these cities) in order to travel the shortest distance and return at the starting point? An example solution to the above-mentioned example is:

Complexity

The difficulty with the aforementioned problem is that there exists no straight-forward algorithm which is guaranteed to find an optimal solution. This in contrast to, for example, the shortest path problem [3] which has been solved by the famous Dutch computer scientist Edsger Dijkstra.

The Traveling Salesman Problem is an NP-complete problem which is subject to combinatorial explosion. Please refer to our previous article [4] on solving constraint satisfaction problems in Java for more information on hard search problems, NP-completeness and combinatorial explosion.

One (naïve) approach to solve a TSP is through the use of a brute-force, generate and test algorithm – generate all possible solutions to the problem (i.e., generate all possible routes), determine the length of each route, and select the shortest one.

The size of the solution space (possible solutions to the problem) is $\frac{1}{2}(n-1)!$ for $n>2$, where n is the number of cities. A TSP with 10 cities therefore has 181.440 possible solutions. If the computer used to solve this problem is able to generate a solution in, for example, one millisecond, the problem is solved in three minutes.

However, if we now want to solve a TSP with 100 cities, the same computer would need **1.48×10^{145}** years to find the optimal solution. One can easily see that such or similar approaches with a complexity of order O(n!) are not feasible for these classes of complex search problems.

IPROFS

"Genetic algorithms are based on a biological metaphor: they view learning as a competition among a population of evolving candidate problem solutions. A 'fitness' function evaluates each solution to decide whether it will contribute to the next generation of solutions. Then, through operations analogous to gene transfer in sexual reproduction, the algorithm creates a new population of candidate solutions."

Artificial Intelligence, Structures and Strategies for Complex Problem Solving

Backtracking, hill climbing and nearest neighbor

Other, more efficient approaches include backtracking (please refer to the afore-mentioned article on constraint satisfaction problems [4]) and hill climbing [5], although these are still not sufficient (by far) to solve these classes of problems with extremely large search spaces.

Another approach is the nearest neighbor algorithm. This algorithm starts at a random node (city) and then travels to the nearest node that has not been visited yet. It does so until all nodes have been visited. The complexity of this algorithm is O(n) and it thus finds a solution quickly, but in most cases, the solution found is far from optimal. What is needed is a totally different kind of strategy - one that is able to find a (near) optimal solution in a reasonable amount of time.

Survival of the fittest

In nature, species have, from a biological point of view, one main goal, which is to survive. As a result, we can observe that (almost) all species are perfectly adapted to their surroundings, therefore maximizing the chance of survival as a whole. This Darwinist perspective on biology focuses on evolution. Within this context, individual entities (e.g., organisms) within a population strive for survival on the micro-level. The weaker entities die and the fittest survive. These surviving entities mate with each other and generate off-spring. This off-spring has combined features of its parents, which results in a new population of somewhat altered entities, each generation being fitter (on average, as a whole) than the previous one. This results in optimized chances of population survival on the macro-level.

The beauty of this system lies in the fact that it is not being controlled by a central, intelligent entity which governs the optimisation process (although religion and the creationist or 'intelligent design' paradigms, which do not fall within the scope of this paper, would state otherwise). On the other hand, the basic rules of nature on the micro-level *automatically* induce evolving species on the macro-level, without the need of a centralised, intelligent entity (or algorithm) which governs the optimisation process. The entities themselves do not know they are evolving, nor do they have a conscious influence on it, nor is nature itself an intelligent entity which consciously governs this process. Evolution is in a sense a very 'dumb' process which leads to *highly* optimised entities in nature.

The aforementioned paradigm forms one of the fundamental ideas within evolution theory, and can also be used to solve hard computational problems, through the use of evolutionary computing. One effective example of evolutionary computing is genetic algorithms. This will be explained and demonstrated by solving the aforementioned Traveling Salesman Problem using this approach.

Genetic Algorithms

Genetic algorithms [6] approach complex computational problems the same way as nature lets species evolve. The genetic paradigm consists of several steps and concepts, which are explained within the following paragraphs.

1. Initialisation of a population

The first step within a genetic algorithm (GA) consists of generating a population of random solutions to a problem (the gene pool).

Within the context of the TSP, we have the following objects:

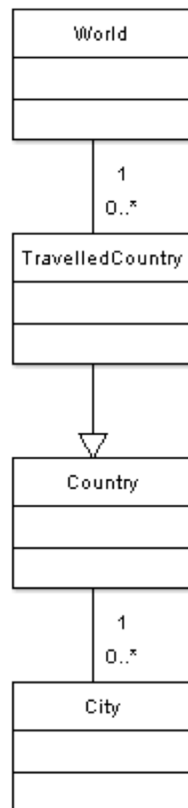
- City: one individual city, containing two attributes - an X and an Y coordinate;
- Country: a collection of City objects, as depicted in the aforementioned TSP example;
- TravelledCountry: a Country object, including a route through its City objects;
- World: a population (collection) of TravelledCountry objects.

The input to the TSP is a Country object. The GA then generates, for example, 1.000 random

“A genetic algorithm is a search technique used in computing to find exact or approximate solutions to optimisation and search problems.”

Wikipedia.

TravelledCountry objects (using a randomising method) and stores them in a World object. It contains all kinds of random solutions to the TSP. This structure is depicted within the following UML class diagram:



The World object forms the initial population of solutions and as such the starting point for the GA.

2. Fitness

One key concept within genetic algorithms is the fitness function. A fitness function merely determines the value or fitness of a (already generated) solution. It does not generate a solution itself.

Within the context of the TSP, the fitness method merely determines the length of a generated route through the cities, and returns this value. All it needs is the coordinates of the cities, the generated route through these cities and some Pythagoras calculations (one for each pair of City objects that are linked by the generated route). Based on this input it calculates the length of a route.

Within the aforementioned class diagram, the fitness method for the TSP is contained within the TravelledCountry class. Each TravelledCountry object within the World collection object has a different fitness, as can be determined by the fitness method.

The second step within a GA consists of determining the fitness of each generated (random) solution within the population (i.e., the World object within the TSP example). These solutions will most probably be poor (i.e., have a low fitness), as they are all generated in a completely random way.

Finally, the calculated fitness values of all solutions are normalised by multiplying the fitness value of each individual solution by a fixed number, so that the sum of all fitness values equals 1. These normalised values are then stored within the TravelledCountry objects.

3. Generating a new population

The third step in a GA consists of generating a new (fitter) population of solutions, by repeating the following steps until the new population is complete (i.e., has the same amount of solutions as the original population):

- (a) selection;
- (b) cross-over;
- (c) mutation;
- (d) acceptance.

3a. Selection

Select two (parent) solutions from the original (or previous) population according to their fitness - the better fitness, the bigger chance to be selected.

Most selection functions [6] are stochastic and designed so that a small proportion of less fit solutions are selected. This helps keep the

diversity of the population large, preventing premature convergence on poor solutions (local optima). One popular and well-studied method is the roulette wheel selection method [7]:

- The population is sorted by descending (normalised) fitness values;
- Accumulated normalised fitness values are computed (the accumulated fitness value of an individual is the sum of its own fitness value plus the fitness values of all the previous individuals);
- A random number R between 0 and 1 is chosen;
- The selected individual is the first one whose accumulated normalized value is greater than R.

There exist several variations to the selection process. One of them focuses on *elitism*, which means that the *n* fittest solutions are automatically selected.

This process is then repeated for the second to be selected parent solution, after which the GA proceeds to the next step.

3b. Cross-Over

Cross-over [8] is a genetic operator which is used to create two new individual solutions from two parent solutions (i.e., the two selected solutions in step 3a.).

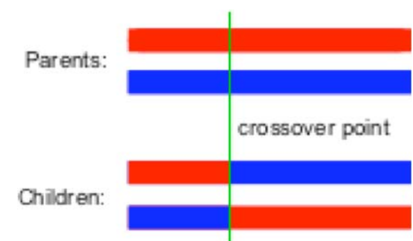
Several cross-over operators exist. To illustrate, we could represent a solution to a TSP with 9 cities (A, B, C, D, E, F, G, H, I) as a string - for example: ABCDEFGHI. This solution means that the cities are traveled in the mentioned order. Let's say that the following two parent solutions within this example have been selected within step 3a.:

- ABCDEFGHI;
- IGAHFDBEC.

These two parent solutions can be 'cross-overed' (thus generating two new child solutions) in several ways, as described below. Within all these strategies, a cross-over point is selected. This cross-over point is a location within the string representation of the solutions – for example, the fourth element. In general, this cross-over point is not fixed, but determined randomly each time the cross over operator is used.

- One point cross-over:

A single crossover point on both parents' solution strings is selected. All data beyond that point in either solution string is swapped between the two parent solutions. The resulting solutions are the children:



Within the aforementioned example, the following two child solutions are created if the cross-over point is 4:

- ABCDFDBEC;
- IGAHEFGHI.

The one point cross-over operator is used for many GA problems. However, this operator (i.e., a direct swap) is not suitable for GAs which include ordered lists, such as the TSP. As can be seen in the aforementioned two child solutions, duplicates are introduced and necessary candidates are removed, thus generating invalid 'solutions' (i.e., within the aforementioned child solutions, some cities are visited twice, and some are not visited at all). For such problems, we need another cross-over operator.

- Cross-over for ordered solutions:

The solution up to the cross-over point is retained for each parent solution. The information after the cross-over point is ordered as it is ordered in the other parent.

Within the aforementioned string example, the following two child solutions are created if the cross-over point is 4:

- ABCDIGHFE;
- IGAHBCDEF.

These are both valid solutions.

Please note that cross-over does not always have to take place. For example, when the cross-over point equals the last position within a string representation of a solution, the created child solutions are exact copies of their parents.

There are several other cross-over operators, which will not all be discussed in detail here. They are variations of the ones already discussed. An example is the two point crossover operator, which uses two instead of one crossover points.

When the two selected parent solutions have been cross-overed within this step, and thus two new child solutions have been generated, the GA proceeds to the next step.

3c. Mutation
Mutation [9] is a genetic operator used to maintain genetic diversity from one generation of a population of solutions to the next.

The classic example of a mutation operator involves a probability that an arbitrary element in a solution will be changed from its original state. A common method of implementing the mutation operator involves generating a random variable for each element in a sequence. This random variable tells whether or not a particular element will be modified.

The purpose of mutation in GAs is to allow the algorithm to avoid local minima by preventing the population of solutions from becoming too similar to each other (degeneration), thus slowing or even stopping evolution.

The mutation operator takes one child solution (as generated within the previous step) as its argument, performs its mutation operation on it, and then returns the mutated solution. It does so for both generated child solutions. The GA then proceeds to the next step.

For example, if the input solution is ABCDIGHFE, then the mutation operator might return AECDIGHFB. Here, the second and last element have been swapped. The mutation operator does this by traveling to each element within the input solution, and then, depending on a probability, swapping this element with a random other element.

3d. Acceptance
The two newly created child solutions (which were selected, cross-overed and mutated) are placed into a new population (i.e., World object).

These four steps are repeated until the new population contains the same amount of solutions as the original (or previous) population.

4. Replacing the previous population
The previous population is replaced by the new population.

5. Test the solution
If the end condition is satisfied, stop, and return the best solution within the current population. An end condition could for example be:

- The algorithm has run for a certain amount of time;
- The algorithm has found a solution that has a certain fitness.

If the end condition has not been satisfied yet, the algorithm needs to re-iterate to step 2.

The aforementioned steps form the Genetic Algorithm. A GA does not necessarily find the *best* solution to a complex problem (although it often will), but it will find a *good* solution - within a reasonable amount of time.

"Problems which appear to be particularly appropriate for solution by genetic algorithms include timetabling and scheduling problems."

Wikipedia.

Java GALib

Within the previous sections, the structure of genetic algorithms has been explained.

Based on this information, a software engineer is able to develop a genetic algorithm manually in Java. Another option is to use a standard Java GA library. There are several open-source GA libraries available on the Internet, one of which is Java GALib [10]. This library contains an abstraction layer which facilitates modeling a problem so that it can be solved by a genetic algorithm, and it contains pre-defined genetic algorithms which can be used to solve the modeled problem.

The following is an example of how to model and solve a GA problem in Java using GALib. This problem focuses on finding a bitstring with the largest possible value.

For example, given a bitstring containing 20 positions (bits), the largest possible value is 11111111111111111111.

This is obviously a very simplistic problem, which in fact does not need a genetic algorithm to be solved, as one can easily compute the largest value of a bitstring without a genetic algorithm. However, it *can* be modeled as a search problem and thus be solved with a genetic algorithm, and due to the simplicity of the problem, it is a good and easy to follow example to show how GALib works.

Please note that within GALib, a possible (random) solution to a problem is called a chromosome and an element of a chromosome is called a gene (which is the 'official' terminology within genetic algorithms).

```
// MaximalBitString extends the
// default GALib class GAString
public class MaximalBitString
extends GAString
{
   // The fitness function. It
   // calculates and returns the
   // fitness of a solution (i.e.,
   // the value of the bitstring)
   protected double getFitness(int
   randomSolution)
   {
      String s =
      this.getChromosome
      (randomSolution).
      getGenesAsStr();

      return
      (getChromValAsDouble(s));
   }
   // The constructor
   public MaximalBitString()
   throws GAException
   {
      super(
      20, /* a chromosome
          (solution) has 20 bits
          */
      50, /* population of N
          chromosomes */
      0.7, /* crossover
           probability */
      5, /* random selection
         chance % (regardless of
         fitness) */
      50, /* stop after N
          generations */
      0, /* number of preliminary
         runs */
      10, /* maximum preliminary
          generations */
      0.01, /* chromosome mutation
            probability */
      0, /* number of decimal
         places in chromosome (0
         means: treat chromosome
         as integer) */
      "01", /* gene space
            (possible gene values
            are '0' or '1') */
      Crossover.ctTwoPoint, /*
            crossover type */
```

```
      true); /* compute
             statistics? */
   }

   // The main method
   public static void
   main(String[] args)
   {
      System.out.println
      ("MaximalBitString...");

     try
      {
          // Create an object
          // using the defined
          // constructor
          MaximalBitString
          bitString = new
          MaximalBitString();

          // Create a thread
          Thread
          threadMaximalBitString =
          new Thread(bitString);

          // And solve the problem
          threadMaximalBitString.
          start();
      }
      catch (GAException gae)
      {
         System.out.println
         (gae.getMessage());
      }
   }
}
```

This is an example of how to solve a basic problem using a genetic algorithm in Java with GALib. Of course, more realistic and complex problems, such as the Travelling Salesman Problem, can be modelled and solved with GALib. Please refer to Java GALib [10] for more documentation and examples.

IPROFS

For More Information

For more information about **IPROFS**, call us on +31 – (0)23 – 5476369. To access information via the World Wide Web, go to:
www.iprofs.nl

For referenced material and more information about Genetic Algorithms, go to:

[1] en.wikipedia.org/wiki/Natural_selection
[2] en.wikipedia.org/wiki/Traveling_salesman_problem
[3] en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[4] www.iprofs.nl (Hanxleden Houwert, N. van, Aircraft Scheduling, DNA Sequencing and Sudoko - Solving Constraint Satisfaction Problems in Java, IPROFS whitepaper, Haarlem, the Netherlands, 20 June 2008.)
[5] en.wikipedia.org/wiki/Hill_climbing
[6] en.wikipedia.org/wiki/Genetic_algorithm
[7] en.wikipedia.org/wiki/Selection_%28genetic_algorithm%29
[8] en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29
[9] en.wikipedia.org/wiki/Mutation_%28genetic_algorithm%29
[10] sourceforge.net/projects/java-galib

© 2008 **IPROFS**. All rights reserved. This whitepaper is for informational purposes only. IPROFS makes no warranties, express or implied, in this paper. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Document published June 2008.

Author

Nicolas van Hanxleden Houwert

is a Solutions Architect at IPROFS with an interest in expert and decision support systems, complex search problems and their solution using specialised models and algorithmic strategies. His experience gained within this field covers several domains, including those of automated aircraft scheduling and decision support (Dutch National Aerospace Laboratory, Schiphol Airport, Eurocontrol), optimised routing of telephone calls for the largest call-center in the Netherlands (Siemens Business Services, Belastingdienst), bioinformatics and DNA sequencing (University College Dublin, Ireland). In the past, he has evangelised these topics within his capacity as a Software Engineering lecturer.

For more information about IPROFS and what we can do for you, go to: www.iprofs.nl