Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **11** Due: **24 Nov 98**

PLEASE WRITE YOUR TA'S NAME ON YOUR PAPER.

Goals

- You learn how quickly or slowly independent, incremental, random changes would "evolve" towards a goal.
- You learn about simulated evolution, using Genetic Algorithms (GAs), and how these differ from independent, random, incremental evolution: better changes are accumulated through time by use of a "fitness" function.
- You learn about the strengths and limitations of GAs.

Introduction to Genetic Algorithms

There is a common saying that, given enough time, a monkey bashing on a typewriter could produce all the works of Shakespeare.[1] Presumably, given enough time, the same monkey could a produce all the 6.034 problem-sets you get in this course. (Surely there is additional, independent evidence for this fact.)

The amount of time required will, of course, be rather longer than the time you'll spend here at MIT. So we make the task a bit easier: we want a monkey to type randomly until it generates the string '*intelligent*'. However, real monkeys are difficult to get hold of. We will do the next best thing: decide to use a Scheme program to simulate the monkey's random typing.

The target string *intelligent* is 11 characters long, so your program should generate strings of the same length $l = 11$ using the 26 lower-case characters '`a`' to '`z`'. Use l to represent the length of the strings and c to represent the number of characters in the alphabet. Recall that the probability of an event X happening can be calculated from 1.0 minus the probability that X won't happen.

In the first few problems, we will try to calculate how unlikely it is that one could arrive at exactly the target string by chance alone, without some measure of "fitness" that accumulates good changes.

[1] These problems are based on those given in Chapter 1 of John Maynard Smith's *Evolutionary Genetics* (1989) which are in turn based on an example in Chapter 3 of Richard Dawkins' *The Blind Watchmaker: Why the evidence of evolution reveals a universe without design* (1986). Despite coming from the biology literature, they are still very relevant for studying artificial evolution and genetic algorithms.

Then, we will carry out, step by step, the calculations that are done to "simulate" evolution. This method has a number of components that we cover in turn in the problem set:

1. The representation of populations in terms of individual strings
2. Copying these strings as a way of simulating reproduction
3. The notion of *mutation* by introducing random error into the copying process
4. The notion of *fitness* as a way to test that some strings are "closer" to the target string (in this case "intelligent") than others

For reference, please read the Winston text, Chapter 25, up to page 516. You may want to skip reading the next section, concerning the Scheme code, and return to it after consulting the Winston text and working the first few problems that follow the code.

Code

The file `ga-runs-faster.scm` in `/ps11/` has Scheme code for a simple genetic algorithm (GA) that evolves character strings, with fitness being determined by the number of characters that match with a target string. **Important: run the code a few times to get a feel for how it works.** Please try it; it's rather important to get a sense of the system working, which is rather remarkable.[2] We give an example of output from the algorithm immediately below.

The basic procedure for the genetic algorithm is the function `ga`, with arguments as follows.

```
(ga popsize repros target mrate)
```

Where `popsize` is the population size (number of offspring); `1000` is the number of reproductions (that is, the number of "generations"), `target` is the target string; and `mrate` is the mutation rate. We always start from a random initial string of the given length. For instance, for our problem set we might call `ga` as follows:

```
(ga 10 1000 "intelligent" 0.01)
```

This says to have each generation produce 10 offspring; compute 1000 generations; start at a random string, and introduce copying errors with a rate of 1% per letter per generation (that is, out of 100 copying operations, 1 will go wrong, introducing another randomly drawn letter).

You shouldn't call `ga` directly, however. Because genetic algorithms are inherently probabilistic, we generally carry out multiple runs of the same system; this is done via the function `ga-multi`, which calls `ga` repeatedly, as follows:

[2] Also, you may have to be patient about Scheme/Solaris signal interrupts, we are sorry to say. In this case, you can restart the Scheme process.

```
(define (ga-multi nruns popsize repros target)

Example:

(ga-multi 10 10 1000 "intelligent")
```

This runs `ga` over and over, 1000 generations each time, 10 times total, with 10 random draws for the starting point.

Note that `ga-multi` **fixes** the mutation rate as `1/(length target)` (in our examples, the target length is 11, so the mutation rate is approximately 0.09). You can easily change this in the code. `Ga-multi` is useful because it will automatically compute the mean and standard deviation of the results of a number of complete runs.

Here is some sample output from the algorithm. We call the algorithm for 1000 generations, a mutation rate of about 0.09, 6 times over. The program generates **a lot** of output — be patient!

```
(load "ga-runs-faster.scm")
;Loading "ga-runs-faster.scm" -- done
;Value: ga-multi

(ga-multi 10 6 1000 "intelligent")

;; Note that we START with the random string efeeekscaan, which
;; happens to have 1 letter in the right spot matched against intelligent
;; (the fourth letter e), so the FITNESS is 1

Experiment 0
Repro=0 Current best: (efeeekscaan . 1)
Repro=1 Current best: (efeeekscaan . 1)
Repro=2 Current best: (efeeekscaan . 1)
Repro=3 Current best: (efeeekscaan . 1)
Repro=4 Current best: (efeeokscaan . 1)
Repro=5 Current best: (efeeekscaan . 1)
Repro=6 Current best: (efeeeksozan . 1)
Repro=7 Current best: (efeeekseaan . 1)
Repro=8 Current best: (efeeokscaan . 1)
Repro=9 Current best: (efeeekscaan . 1)
...
...
...
Repro=17 Current best: (efeeekscaan . 1)
Repro=18 Current best: (efeeeksoean . 2)
...
Repro=62 Current best: (ebzelksoean . 3)
Repro=63 Current best: (ebzelksoean . 3)
...
..
;; After 1000 generations, the system has got all but 1 letter
;; correct, so the fitness is 10.
```

```
Repro=997 Current best: (intellxgent . 10)
Repro=998 Current best: (intellxgent . 10)
Repro=999 Current best: (intellxgent . 10)

Finished after 1000 reproductions

;; Here is one run in this series that came out perfectly

Experiment 2

Repro=0 Current best: (uminyuindiy . 1)
...
...
Repro=843 Current best: (intellicent . 10)
Repro=844 Current best: (intellicent . 10)
Repro=845 Current best: (intelligent . 11)

Finished after 846 reproductions
```

With this prefatory material complete, we turn to the problems.

Problem 1 Basic string probability calculations

Starting with one $l = 11$ string where each letter is chosen randomly (with equal probability) from the $c = 26$ alphabet:

Part A

How many possible such strings are there?

Part B

What is the probability that at least one of the 11 characters correctly matches the corresponding character in the target string?

Part C

What is the probability that exactly one of the 11 characters is correct? *(Remember: each character is generated independently and with equal likelihood)*.

Problem 2 Introducing mutation in copying

The original string is copied ten times, but errors creep in. For each letter copied, there is a 99% chance that everything goes OK, but 1% of the copy operations "mutate" the letter, changing it to any one of the other characters in the alphabet, with equal probability.

Part A

If none of the characters in the original string are correct, what is the probability that at least one character does match in at least one of the ten copies?

Problem 3 Fitness

We can now measure the "fitness" of each copy by calculating the number of characters that *correctly* match the corresponding character in the target string. The copy with the highest fitness value is chosen as the *elite*. If there are two or more copies with the same highest fitness value, one of them is chosen at random to be the elite. The elite copy is then used as the "parent" of the next generation" ten copies are made of the elite, with mutations, in the same way as before.

Part A

Approximately how many generations will pass before a string with a nonzero fitness value is created, if the original string had a fitness of zero? *(Hint: in these questions, you can estimate the number of generations before an event X occurs by calculating the reciprocal of the probability of X occurring on any one generation).*

Part B

If you waited around long enough, a string with a fitness value of 10 will be created. What is the probability that at least one of the ten "children" of such a string is correct at *all* 11 sites?

Part C

Roughly how many generations will it take to progress to a copy with all 11 characters correct, starting from a string with a fitness of 10?

Problem 4 Running the system: what happens?

Part A Convergence

What do you notice about the speed with which the system approaches the correct string "intelligence"? You will probably have to consider multiple runs, and do a bit of thinking, because the GA is inherently probabilistic. Please explain your observation in a sentence or two.

Part B Mutation rate

Change the mutation rate, `mrate`, in `ga-multi`, from 0.09 to 0.01. Does it take more or less time to find an optimally fit string? Why or why not?

Part C Sex

The current code shows only *asexual* reproduction, where copies of the best individual are made with mutation. Describe in a few sentences how you would alter the code so as to implement *sexual* reproduction, with recombination or crossover.

Problem 5 The limits of genetic algorithms

Part A

In what **single major** way does the evolutionary model described here *differ* from biological evolution (Hint: one does not need to be a biologist to know the answer.)

Part B

Assume this kind of evolutionary system starts with an *arbitrary* two-letter English word in the form consonant-vowel (C-V), or vowel-consonant (V-C), such as *do*, *by*, *at*, *in* (but **not** *yb* or *oo*; in fact there are no legitimate V-V or C-C words in English — take our word for it. The English vowels comprise *a, e, i, o, u, y*; consonants are all the other letters.).

Suppose our evolutionary system starts with the word string "do" and tries to evolve it into the string "by" with this additional constraint: the intermediate strings that are evolved must all be "viable" — they must all be legitimate two-letter English words. Given this constraint, is the evolution of "do" into "be" possible (given enough time)? Similarly, given this viability constraint, is it possible for *any* legitimate two-letter English word to evolve into *any other* two-letter English word? Why or why not?

Note that this corresponds to a kind of "viability" constraint on the evolutionary process (the organisms in between generations must all "live").

What implications does this have for Genetic Algorithms in general?

Code Comments

Here are the comments from the beginning of `ga-runs-faster.scm`.

This code implements a genetic algorithm that gradually evolves a 'population' of individual 'genomes'. Each genome is a string of characters from an alphabet specified in the function rand-char, and the fitness of each genome is the number of characters that match at homologous positions on a 'target' string, passed as an argument to the function eval-fit.

NOTE: `ga-multi` must be called with a string instead of a list. E.g., `(ga-multi 10 10 1000 "intelligent")`

The style of genetic algorithm is "steady state" (rather than "generational"): on each iteration, an individual is selected to be reproduced and another individual is selected to be replaced. The individual to be reproduced is copied with errors ("mutations") into the individual to be replaced. Thus, reproduction is *asexual*; there is no "recombination" or "crossover). Sexual repoduction could be introduced by using the gene-copy function.

In the current version, the best individual in the population is selected for reproduction, with a random choice if more than one individual holds the highest fitness score. Alternative selection methods could be used by altering the code in the reproduce function.

The `ga` function performs one experiment — it takes as arguments a population size (`popsize`), the number of reproductions `repros`, the target string, `target`, and the mutation rate `mrate`. Because this is a steady-state GA, the number of "generations" can be determined by the number of reproductions divided by the population size.

Because GAs are stochastic systems, it is generally necessary to make make multiple runs and compute summary statistics such as the mean and standard deviation. The `ga-multi` function does this for you: the argument nruns specifies how many runs are to be executed, and `ga-multi` displays a count of the number of runs that terminated with success (i.e., having evolved a string that matches the target), along with the mean and standard deviation of the number of reproductions necessary for a success. Note also that `ga-multi` does *not* take a mutation-rate parameter: instead, it estimates the optimal mutation rate at (1/length-of-target-string) — there are theoretical arguments for doing things this way, but for current purposes you can consider this a convenient hack.