A Genetic Algorithm Implementation

Roy M. Turner (rturner@maine.edu)

Spring 2017

Contents

1 Introduction **3**

2 Header information **3**

3 Class definitions **3**
3.1 Individual . . . 3
3.2 Population . . . 4
3.3 GA (genetic algorithm) . . . 4
3.4 Initialization "after" methods . . . 5

4 Variables **6**
4.1 Alphabets . . . 6
4.1.1 condition-string . . . 6
4.1.2 Defined alphabets . . . 6
4.2 UI variables . . . 8

5 Methods/functions **8**
5.1 get-alphabet . . . 8
5.2 random-genome . . . 9
5.3 Fitness computation methods . . . 9
5.3.1 compute-fitnesses . . . 9
5.3.2 compute-fitness . . . 9
5.3.3 sort-by-fitness . . . 10
5.4 status . . . 10
5.5 display . . . 10
5.6 best-fitness, average-fitness, worst-fitness . . . 11
5.7 genome-as-string . . . 11
5.8 reproduce . . . 11
5.9 crossover . . . 12
5.10 mutate . . . 13
5.11 choose? . . . 13
5.12 add-individuals . . . 13
5.13 fittest . . . 13

6 User interface **13**
6.1 Computing fitness . . . 14
6.1.1 string-match-fitness . . . 14
6.2 Running the GA . . . 14
6.2.1 next-generation . . . 14
6.2.2 `run` . . . 14
6.3 Sample GA task: Learn a string . . . 16
6.4 Collect statistics . . . 17
6.5 Finding out what the answer is . . . 17
6.5.1 best-answer . . . 17
6.5.2 best-answers . . . 18
6.6 Another GA task: TSP . . . 18
6.6.1 City map . . . 18

1 Introduction

The file "genetic-algorithm.lisp" contains code for a simple genetic algorithm (GA), written using CLOS. It was written both to provide a genetic algorithm implementation to experiment with as well as to give an example of a reasonably complex Lisp program.

The file is written as a *literate programming* file, with (in this case, LaTeX) documentation interleaved with the code. If you are interested in the program that converts back and forth from Lisp ↔ LaTeX, let me know.

2 Header information

```
1  (in-package cl-user)
```

3 Class definitions

The program is structured using three CLOS classes:

- `ga`: the genetic algorithm object itself;
- `population`: an object holding the population of individuals; and
- `individual`: an object representing an individual solution.

3.1 Individual

The `individual` class holds all the information needed to represent an individual in the population, that is, a candidate solution to the problem. Each has:

- `id`: the name of the individual
- `size`: length of the genome
- `kind`: the name of the alphabet the genome is drawn from
- `fitness`: the individual's current

The size of the genome is set when the GA is set up, as is the kind of genome it is. "Kind" here refers to the alphabet that the elements of the genome are drawn from; see the description of the variable `*alphabet-map*` for the kinds of genomes supported and their names. The fitness is a number that is returned from the fitness function (see class `ga`).

The genome of an individual can be considered a string of characters by the user. Internally, however, the string is converted (via the function `condition-string`) to a list of symbols internally. Each symbol corresponds to a letter in the string. For example:

```
CL-USER> (condition-string "Hi there")
(H |i| | | |t| |h| |e| |r| |e|)
```

The `|`'s in the symbol names mean that the symbol has special characters that are not usually part of symbol names, such as lower-case letters.[1] The function `genome-as-string` converts this list back to a nice-looking string.

[1] These aren't usually found in symbol names if we are using a case-insensitive Lisp, as I usually do.

```
 2  (defclass individual ()
 3    (
 4     (id :initform (intern (symbol-name (gensym "INDIV"))) :initarg :id :accessor id)
 5     (size :initform 20 :initarg :size :accessor size)
 6     (genome :initform nil :initarg :genome :accessor genome)
 7     (kind :initform 'bitstring :initarg :kind :accessor kind)
 8     (fitness :initform nil :initarg :fitness :accessor fitness)
 9     )
10    )
```

3.2 Population

The `population` class defines a population of individuals. It has instance variables:

- `id`: the name of the population
- `individuals`: a list of all individuals in the population
- `size`: the size of the population
- `individual-size`: the size of each individual's genome (see 3.2)
- `individual-kind`: the kind of each individual's genome (see 3.2)

```
11  (defclass population ()
12    (
13     (id :initform (intern (symbol-name (gensym "POP"))) :initarg :id :accessor id)
14     (individuals :initform nil :initarg :individuals :accessor individuals)
15     (size :initform 50 :initarg :size :accessor size)
16     (individual-size :initform 20 :initarg :individual-size :accessor individual-size)
17     (individual-kind :initform 'bitstring :initarg :individual-kind
18                      :accessor individual-kind)
19     )
20    )
```

3.3 GA (genetic algorithm)

The `ga` class holds methods and instance variables needed for the operation of the GA. Instance variables include:

- `id`: the name of the GA
- `population`: the population (an instance of `population`)
- `mutation-rate`: the rate of mutations; that is, how likely a mutation is during each reproduction event
- `crossover-rate`: the rate of crossovers; that is, how likely a crossover event is during each reproduction
- `fitness-function`: a function that is applied to the genome of an individual to determine that individual's fitness (a number)

- `individual-size`: the size of an individual's genome
- `population-size`: the size of the population
- `individual-kind`: the kind of genome in use (see the `individual` class)
- `breeding-pairs`: the number of breeding pairs to select each generation
- `silent`: if t, then no status information is printed; this can speed up the execution of the GA

The `breeding-pairs` instance variable is used to determine how many of the fittest individuals are selected to produce children for the next generation. Currently, the top $2n$ individuals, where n = `breeding-pairs`, are selected and paired off in order. This is a place where other choices could be tried to see if it makes any difference. The pair then create two children. See the method `reproduce` for information about how reproduction actually takes place.

```
21  (defclass ga ()
22    (
23     (id :initform (intern (symbol-name (gensym "GA"))) :initarg :id :accessor id)
24     (population :initform nil :initarg :population :accessor population)
25     (mutation-rate :initform 0.1 :initarg :mutation-rate :accessor mutation-rate)
26     (crossover-rate :initform 0.1 :initarg :crossover-rate :accessor crossover-rate)
27     (fitness-function :initform #'(lambda (i) (random 1.0))
28        :initarg :fitness-function :accessor fitness-function)
29     (individual-size :initform 20 :initarg :individual-size :accessor individual-size)
30     (population-size :initform 50 :initarg :population-size :accessor population-size)
31     (generation :initform 0 :initarg :generation :accessor generation)
32     (individual-kind :initform 'bitstring :initarg :individual-kind :accessor individual-kind
33     (breeding-pairs :initform 4 :initarg :breeding-pairs :accessor breeding-pairs)
34     (silent :initform nil :initarg :silent :accessor silent)
35     )
36    )
```

3.4 Initialization "after" methods

Often in CLOS, it's useful to perform additional actions after an object is initialized by `make-instance`. Instead of calling a user-defined initialization function, it's cleaner in many cases to write an `:after` method for the built-in method `initialize-instance`, which is called as a result of calling `make-instance`. The `:after` method is called after the built-in method without the user having to do anything.

We use the `:after` methods here to set up the population and individuals when `ga` is instantiated.

```
37  (defmethod initialize-instance :after ((instance ga) &rest args)
38    (declare (ignore args))
39    (with-slots (population individual-size
40          population-size generation individual-kind) instance
41      (setq generation 0)
42      (setq population (make-instance 'population :size population-size
43       :individual-size individual-size
44       :individual-kind individual-kind))
```

```
45      (compute-fitnesses instance)
46      )
47    )
48
49  (defmethod initialize-instance :after ((instance population) &rest args)
50    (declare (ignore args))
51    (with-slots (individuals individual-size size individual-kind) instance
52      (setq individuals (loop for ind from 1 to size
53     collect (make-instance 'individual :size individual-size
54    :kind individual-kind
55    )))))
56
57  (defmethod initialize-instance :after ((instance individual) &rest args)
58    (declare (ignore args))
59    (with-slots (size genome kind) instance
60      (setq genome (or genome (random-genome instance size kind)))))
```

4 Variables

4.1 Alphabets

We define several alphabets for you to use. You may define others; if you do, however, you will need to modify the `get-alphabet` function. Each alphabet is present as a genome-like list and is in a variable named `*alphabet-XXXX*`, where `XXXX` describes the alphabet.

First, we define a function needed to convert strings into genomes.

4.1.1 condition-string

As we described when discussing the `individual` class, this will take a string and return a properly-formed genome. A genome is represented as a list of symbols, each of which corresponds to a character in the string. Two versions of this are provided, one, which just returns its argument, for when the genome is already a list, and the other that actually converts the string to a genome.

```
61  (defmethod condition-string ((str list))
62    str)
63  (defmethod condition-string ((str string))
64    (map 'list #'(lambda (a) (intern (string a)))
65         str))
```

4.1.2 Defined alphabets

Here are the alphabets we have defined:

- Lower case letters:

```
66    (defvar *alphabet-lower-case*
67        (condition-string "abcdefghijklmnopqrstuvwxyz"))
```

- Upper case letters:

```
68    (defvar *alphabet-upper-case*
69        (condition-string "ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
```

- Numerals 0–9:

```
70    (defvar *alphabet-numerals*
71        (condition-string "0123456789"))
```

- The space character:

```
72    (defvar *alphabet-space* (condition-string " "))
```

- Punctuation symbols:

```
73    (defvar *alphabet-punctuation*
74        (condition-string ":;.?!,-"))
```

- Special characters:

```
75    (defvar *alphabet-special*
76        (condition-string "@#$%^&*()_+=|\}]{['\"/><"))
```

- Bits (0, 1):

```
77    (defvar *alphabet-bitstring*
78        (condition-string "01"))
```

The variable `*alphabet-map*` contains a mapping from labels to the corresponding alphabets. The format of each entry in the map is:

`(label(s) alphabet(s))`

where the first element is a (keyword) label or list of labels and the remainder is a list of names of alphabet variables or labels. So, for example,

`(:uppercase *alphabet-upper-case*)`

maps the label `:uppercase` to the variable containing the uppercase letters, and

`(:alphanumeric :alphabet :numerals)`

maps the label `:alphanumeric` to the concatenation of the alphabets named `:alphabet` and `:numerals`.

The function `get-alphabet` uses this map to produce alphabets for (e.g.) reproduction and instantiation of individuals.

```
79  (defvar *alphabet-map*
80      '(
81        ((:bitstring :bits :binary) *alphabet-bitstring*)
82        (:uppercase *alphabet-upper-case*)
83        (:lowercase *alphabet-lower-case*)
84        (:punctuation *alphabet-punctuation*)
85        (:numerals *alphabet-numerals*)
86        (:special-chars *alphabet-special*)
87        (:alphabet+space :alphabet *alphabet-space*)
88        ((:letters :alphabet) :uppercase :lowercase)
89        (:alphanumeric :alphabet :numerals)
90        (:alphanumeric+space :alphabet+space :numerals)
91        (:all :alphanumeric+space :punctuation :special-chars)
92        )
93    )
```

4.2 UI variables

The user interface, at the moment consisting of just a few functions (see Section 6), stores the target string, the GA instance, and some execution statistics in these variables

```
94  (defvar *target* '(T h i s space i s space a space t e s t))
95  (defvar *ga* nil)
96  (defvar *stats* nil)
```

5 Methods/functions

5.1 get-alphabet

This uses the `*alphabet-map*` variable to find an alphabet based on a label; see that variable's description to see the valid alphabets. The label can be a symbol or a keyword. The alphabet is returned in the form of a genome, and it has been copied so that no modifications will affect the base alphabets themselves.

```
 97  (defun get-alphabet (label)
 98    (unless (keywordp label)
 99      (setq label (intern (symbol-name label) 'keyword)))
100    (labels ((compose-alphabets (alphabets)
101        (cond
102         ((null alphabets) nil)
103         ((keywordp (car alphabets))
104          (append (get-alphabet (car alphabets))
105          (compose-alphabets (cdr alphabets))))
106         (t
107          (append (eval (car alphabets))
108          (compose-alphabets (cdr alphabets)))))))
109      (let ((alphabets (cdr (assoc label *alphabet-map*
110       :test #'(lambda (a b)
111         (if (listp b)
```

```
112   (member a  b)
113   (eql a b)))))))
114        (when (setq alphabets (compose-alphabets alphabets))
115   (copy-list alphabets)))))
```

5.2 random-genome

This returns a random genome of size *size* composed of characters drawn from the alphabet *kind*, which defaults `bitstring`.

```
116  (defmethod random-genome ((self individual) size &optional (kind 'bitstring))
117    (let ((alphabet (if (consp kind) kind (get-alphabet kind))))
118      (loop with len = (length alphabet)
119   for i from 1 to size
120   collect
121     (nth (random len) alphabet))))
```

5.3 Fitness computation methods

These compute the fitness of individuals based on a *fitness function*, contained in the GA instance.

5.3.1 compute-fitnesses

These two methods, one for `ga` instances and one for `population` instances, together compute the fitness of all the individuals in the GA.

```
122  (defmethod compute-fitnesses ((self ga) &key fcn &allow-other-keys)
123    (with-slots (population fitness-function) self
124      (compute-fitnesses population  :fitness-function (or fcn fitness-function))))
125
126
127  (defmethod compute-fitnesses ((self population) &key fitness-function &allow-other-keys)
128    (with-slots (individuals) self
129      (loop for indiv in individuals
130   do (compute-fitness indiv fitness-function))
131      (sort-by-fitness self)))
```

5.3.2 compute-fitness

This applies a fitness function to the individual's genome. It updates the individual's `fitness` instance variable. There are two versions, one for an `individual` instance, and one for a genome; the former calls the latter.

```
132  (defmethod compute-fitness ((self individual) fcn)
133    (with-slots (fitness genome) self
134      (setq fitness (compute-fitness genome fcn))))
135
136  (defmethod compute-fitness ((genome list) fcn)
137    (funcall fcn genome))
```

5.3.3 sort-by-fitness

This method (of `population`) sorts all of a population's individuals by their fitness, with the best individuals first.

```
(defmethod sort-by-fitness ((self population))
  (with-slots (individuals) self
    (setq individuals (sort individuals #'(lambda (a b)
 (> (fitness a) (fitness b)))))))
```

5.4 status

This prints the status of `ga` to the screen.

```
(defmethod status ((self ga))
  (with-slots (generation silent) self
    (unless silent
      (format t
       "~&Generation ~s: fitness (b/a/w) = ~,2f/~,2f/~,2f, best answer=~s~%"
       generation (best-fitness self) (average-fitness self)
       (worst-fitness self)
       (car (best-answers self 1))))))
```

5.5 display

The `display` generic function and its methods (one each for `ga`, `population`, and `individual`) show the GA and its pieces in a nice form for the user. If `:individuals-too?` is specified for the population or GA version, then all the individuals are displayed as well.

```
(defmethod display ((self individual) &key (stream t))
  (with-slots (id fitness genome) self
    (format stream "~&~2T~s (fitness=~,2f)~15T~s~%"
     id fitness (genome-as-string self))))

(defmethod display ((self population) &key (stream t) (individuals-too? nil))
  (with-slots (id individuals size individual-size individual-kind) self
    (format stream "~&~2T~s: ~s individuals with ~s genomes~%"
     id size individual-kind)
    (format stream "~5THighest/average/lowest fitness:~,2f/~,2f/~,2f~%"
     (fitness (car individuals))
     (average-fitness self)
     (fitness (car (last individuals))))
    (when individuals-too?
      (mapcar #'display individuals))))

(defmethod display ((self ga) &key (stream t) (individuals-too? nil))
  (with-slots (population id mutation-rate crossover-rate generation breeding-pairs) self
    (format stream "~&GA program ~s:~%" id)
    (format stream "~& Current generation: ~s~%" generation)
```

```
170      (format stream "~& Mutation rate: ~s~%" mutation-rate)
171      (format stream "~& Crossover-rate: ~s~%" crossover-rate)
172      (format stream "~& # breeding pairs: ~s~%" breeding-pairs)
173      (format stream "~& Population:~%")
174      (display population :stream stream :individuals-too? individuals-too?)))
```

5.6 best-fitness, average-fitness, worst-fitness

These generic functions and their methods (for `population`, `ga`, `individual`) return the best, average, and worst fitnesses in the population.

```
175  (defmethod best-fitness ((self ga))
176    (with-slots (population) self
177      (best-fitness population)))
178
179  (defmethod worst-fitness ((self ga))
180    (with-slots (population) self
181      (worst-fitness population)))
182
183  (defmethod best-fitness ((self population))
184    (with-slots (individuals) self
185      (fitness (car individuals))))
186
187  (defmethod worst-fitness ((self population))
188    (with-slots (individuals) self
189      (fitness (car (last individuals)))))
190
191  (defmethod average-fitness ((self ga))
192    (with-slots (population) self
193      (average-fitness population)))
194
195  (defmethod average-fitness ((self population))
196    (with-slots (individuals size) self
197      (float (/ (apply #'+ (mapcar #'fitness individuals)) size))))
```

5.7 genome-as-string

This takes a genome (a list) and converts the symbols back into characters and concatenates them. This make is much easier for the user to read.

```
198  (defmethod genome-as-string ((self individual) &key genome-list)
199    (with-slots (genome) self
200      (apply #'concatenate (cons 'string (mapcar #'symbol-name (or genome-list genome))))))
```

5.8 reproduce

These methods of the generic function `reproduce` together take two individuals and return two children (as two separate values). With probability `crossover-rate`, the genomes of the parents will be crossed over at some random point to create the children; with probability `mutation-rate`,

one or both of the children will suffer a random mutation. If no mutation or crossover occurs, the children are copies of the parents.

```
201  (defmethod reproduce ((p1 individual) (p2 individual) &key mutation-rate
202      crossover-rate alphabet)
203    (multiple-value-bind (ng1 ng2)
204        (reproduce (genome p1) (genome p2) :mutation-rate mutation-rate
205    :crossover-rate crossover-rate
206    :alphabet alphabet)
207      (values (make-instance 'individual :genome ng1
208      :size (size p1) :kind (kind p1))
209       (make-instance 'individual :genome ng2
210      :size (size p1) :kind (kind p1)))))
211
212  (defmethod reproduce ((g1 list) (g2 list) &key mutation-rate crossover-rate alphabet)
213    (when (symbolp alphabet)
214      (setq alphabet (get-alphabet alphabet)))
215    (setq g1 (copy-list g1)
216   g2 (copy-list g2))
217    (when (choose? crossover-rate)
218      (multiple-value-setq (g1 g2) (crossover g1 g2)))
219    (when (choose? mutation-rate)
220      (setq g1 (mutate g1 alphabet)))
221    (when (choose? mutation-rate)
222      (setq g2 (mutate g2 alphabet)))
223    (values g1 g2))
```

5.9 crossover

The `crossover` method picks a random spot and crosses over two genomes: i.e., from that point on, the genomes of the two are switched. The method returns the genomes as two values.

```
224  (defmethod crossover ((a list) (b list))
225    (let ((position (1+ (random (1- (length a)))))
226   kids)
227      (setq kids
228        (loop for counter from 1 to (length a)
229     for i in a
230     for j in b
231     when (>= counter position)
232     collect (list i j)
233     else
234     collect (list j i)))
235      (values (mapcar #'car kids)
236       (mapcar #'cadr kids))))
```

5.10 mutate

This method will mutate a genome by replacing a character at a random position by a random character from the alphabet the genome is composed of.

```
237  (defmethod mutate ((genome list) alphabet)
238    (setf (nth (random (length genome)) genome)
239      (nth (random (length alphabet)) alphabet))
240    genome)
```

5.11 choose?

This function takes a probability and returns `t` with that probability.

```
241  (defun choose? (probability)
242    (<= (random 1.0) probability))
```

5.12 add-individuals

Add the `new` individuals to the population. This then sorts the list of individuals by decreasing fitness. (Probably not the most efficient way I could have done this; I'll maybe change it later to insert rather than sort.)

```
243  (defmethod add-individuals ((self population) new)
244    (with-slots (individuals size) self
245      (setq individuals (append new individuals))
246      (sort-by-fitness self)
247      (setq individuals (fittest self size))))
```

5.13 fittest

This returns the `n` fittest individuals in the population as a list.

```
248  (defmethod fittest ((self ga) &optional (n 1))
249    (with-slots (population) self
250      (fittest population n)))
251
252  (defmethod fittest ((self population) &optional (n 1))
253    (with-slots (individuals) self
254      (loop for i from 1 to n
255   for indiv in individuals
256   collect indiv)))
```

6 User interface

There are several functions the user can use to set up and run a genetic algorithm. One way, of course, is to create a `ga` instance using `make-instance`, then run it using the `run` method. For complex problems (e.g., learning a traveling salesperson task, etc.), this is probably the best thing to do.

However, we provide a simple interface here for learning strings. The `learn-string` function creates a `ga` instance, stores it in the variable `*GA*`, and runs the GA.

6.1 Computing fitness

You can define your own fitness functions, of course. Each should take one argument, a genome, and return a number. Other than that, there are no restrictions on what kinds of functions you can have.

We have provided one for the `learn-string` function (below):

6.1.1 string-match-fitness

The `string-match-fitness` function simply counts the number of correct letters. Probably the GA's hill-climbing could be made more efficient by changing this to take into account how close the letters are to the target's and using the order of the letters in the alphabet.

```
257  (defun string-match-fitness (i)
258    (count t (mapcar #'eql i *target*)))
```

6.2 Running the GA

6.2.1 next-generation

This method does the bulk of the GA's work. It is called by `run` repeatedly to evolve the GA. Each call, `next-generation` selects the fittest n individuals (as specified by the `ga` instance's `breeding-pairs` instance variable) and lets pairs of them reproduce. The children are added to the list of individuals, then the list is sorted and the best are kept (as specified by the population's `size` instance variable).

```
259  (defmethod next-generation ((self ga))
260    (with-slots (population generation fitness-function breeding-pairs
261          crossover-rate mutation-rate individual-kind) self
262      (let ((pairs (fittest population (* 2 breeding-pairs))))
263        (loop with c1
264     with c2
265     for i from 0 to breeding-pairs by 2
266     do (multiple-value-setq (c1 c2)
267          (reproduce (nth i pairs) (nth (1+ i) pairs)
268     :crossover-rate crossover-rate
269     :mutation-rate mutation-rate
270     :alphabet individual-kind))
271        (compute-fitness c1 fitness-function)
272        (compute-fitness c2 fitness-function)
273        (add-individuals population (list c1 c2))
274        )
275        (incf generation)
276        )))
```

6.2.2 `run`

The primary method for running the GA is just called `run`. It takes one positional argument, the `ga` instance, and several keyword arguments:

- `:for` - if set, run for this many generations, then stop

- `:until-best-fitness` - if set, run until the fitness of the best individual is $\geq$ this
- `:until-average-fitness` - if set, run until the average fitness of the population is $\geq$ this
- `:until-generation` - if set, run until this generation
- `:recompute-fitness?` - if set, recompute the fitness of all the individuals each generation; this would be useful, for example, if the situation is changing such that the fitnesses are likely to change
- `:file` - a filename to hold the data generated by the run
- `:status-every` - if set, print the status only one every this many generations

Data is written to the file in the form:

```
generation#  best-fitness average-fitness worst-fitness fittest-genome
```

Careful, though, since this will overwrite any old file of the same name.

```
277  (defmethod run ((self ga) &key until-best-fitness until-average-fitness
278          for until-generation recompute-fitness? (file "ga-results.dat")
279          (status-every 50))
280    (with-open-file (out file :direction :output
281      :if-exists :supersede
282      :if-does-not-exist :create)
283      (with-slots (generation silent) self
284        (loop with status-counter = status-every
285     do (next-generation self)
286        (when recompute-fitness?
287          (compute-fitnesses self))
288        (format out "~s ~s ~s ~s ~s~%"
289        generation (best-fitness self)
290        (average-fitness self)
291        (worst-fitness self)
292        (genome-as-string (car (fittest self 1)))
293        )
294        (when (or (null status-counter)
295          (zerop status-counter))
296          (status self)
297          (when status-counter
298    (setq status-counter status-every)))
299        (when for (decf for))
300        (when status-counter (decf status-counter))
301     until (or (and until-best-fitness (>= (best-fitness self) until-best-fitness))
302       (and until-average-fitness (>= (average-fitness self) until-average-fitness))
303       (and until-generation (>= generation until-generation))
304       (and for (zerop for))))
305        (unless silent
306   (format t "Run complete:")
307   (status self))
308        (values (best-answer self) generation))))
```

6.3 Sample GA task: Learn a string

The `learn-string` function is used to create a GA and learn a string. It takes one positional argument, the string to be learned (as a string, which is converted to genome form), and several keyword arguments:

- `:individual-kind` - the kind of individual (genome); defaults to `:alphanumeric+space`
- `:until-best-fitness` - if set, the GA will run until the best fitness $\geq$ this value
- `:until-average-fitness` - if set, the GA will run until the average fitness $\geq$ this value
- `:for` - if set, the GA will run for this many generations
- `:until-generation` - if set, the GA will run until this generation is reached
- `:status-every` - this determines how often a status message is printed (default is every 50 generations)
- `:breeding-pairs` - how many breeding pairs to use (default 4)
- `:mutation-rate` - the mutation rate (0.1)
- `:crossover-rate` - the crossover rage (0.1)
- `:population-size` - the size of the population (50)
- `:silent` - if set, no there are no status messages (default is nil)
- `:file` - data is written to this file (default is "learn-string-results.dat") in the form described in the discussion of `run`.

```
309  (defun learn-string (string &key (individual-kind :alphanumeric+space)
310    until-best-fitness until-average-fitness
311    for until-generation
312    (file "learn-string-results.dat")
313    (status-every 50)
314    (breeding-pairs 4)
315    (mutation-rate 0.1)
316    (crossover-rate 0.1)
317    (population-size 50)
318    (silent nil)
319    (fitness-function #'string-match-fitness))
320
321    (setq *target* (condition-string string))
322
323    (setq *ga* (make-instance 'ga
324          :breeding-pairs breeding-pairs
325          :mutation-rate mutation-rate
326          :crossover-rate crossover-rate
327          :population-size population-size
328          :individual-size (length *target*)
```

```
329          :individual-kind individual-kind
330          :fitness-function fitness-function
331          :silent silent))
332
333    (run *ga* :status-every status-every :file file
334         :until-generation until-generation
335         :until-best-fitness (or until-best-fitness (length string))
336         :until-average-fitness until-average-fitness
337         :for for))
```

6.4 Collect statistics

The `stats` function takes a string and runs `learn-string` on substrings of it `iterations` times each, from 2 to the length of the string, collecting the average of the generations needed fro each substring. The stats are put into `*stats*`. The `kind` argument determines the kind of alphabet to use. (I should probably have this generate random strings for this in future.)

```
338  (defun stats (string &key (iterations 10) (kind :all))
339   (setq *stats* nil)
340    (format t "~%")
341    (loop for i from 2 to (length string)
342        do (format t "~s" i)
343    (loop
344        for j from 1 to iterations
345        collect (multiple-value-bind (val gens)
346    (learn-string (subseq string (- (length string) i))
347          :until-best-fitness i
348          :silent t
349          :individual-kind kind)
350          (format t ".")
351          gens)
352        into sum
353        finally (push (list i (float (/ (apply #'+ sum) iterations))) *stats*))
354        finally (setq *stats* (reverse *stats*)))
355    (loop for run in *stats*
356        do (format t "~&~s~10T~,2f~%" (car run) (cadr run)))
357    *stats*)
```

6.5 Finding out what the answer is

6.5.1 best-answer

This returns the best answer (i.e., the genome of the fittest individual).

```
358  (defmethod best-answer ((self ga))
359    (car (best-answers self 1)))
360
361  (defmethod best-answers ((self ga) &optional (n 1) (as-strings? t))
362    (with-slots (population) self
363      (best-answers population n as-strings?)))
```

6.5.2 best-answers

This returns the `n` best answers at the current time. If `as-strings?` is set, then the genomes are translated to strings before being returned.

```
364  (defmethod best-answers ((self population) &optional (n 1) (as-strings? t))
365    (with-slots (individuals) self
366      (loop
367   for i from 1 to n
368   collect (if (not as-strings?)
369     (genome (nth (1- i) individuals))
370     (genome-as-string (nth (1- i) individuals)))
371   into ba
372   collect (fitness (nth (1- i) individuals)) into bf
373   finally (return (values ba bf)))))
```

6.6 Another GA task: TSP

The Traveling Salesperson Problem (TSP) is a well-known NP-hard problem. Here, we will attempt to solve it with a GA. The task is to find the shortest route through a group of cities, visiting each city only once and ending up at the start city. The model uses (usually) a complete graph, though we can add extremely long (or infinite) paths between cities that don't have a real connection.

We'll model the cities as a simple list of the form:

```
((a b 15)
 (b c 20)...)
```

etc., meaning that the route from city a to city b is 15 units, etc. We assume a undirected graph.

To convert this into a GA, we'll simply use a letter per city and let a string represent a path. Thus for 4 cities, a path might be:

```
"abcda"
```

representing starting at city "a", going to "b", "c", and "d", then back to "a".

The fitness function will compute the cost, and the GA will try to minimize that cost. Note that this inverts the sense of fitness. Thus, we'll make the costs negative, so that as we "reduce" the costs, we really get closer to 0, and so the fitness can be in the usual, increasing, sense.

6.6.1 City map

1. generate-tsp

   First, a function that will generate a random TSP problem given the number of cities and min and max link costs.

   ```
   374  (defun generate-tsp (n mincost maxcost)
   375    (let ((possible (loop for city in *alphabet-upper-case*
   376         for i from 1 to n
   377         collect city)))
   378      (labels ((paths (city rest min max)
   379          (cond
   ```

```
380   ((null rest) nil)
381   (t (append (loop for other in rest
382          collect (list city other (+ min (random (- max min)))))
383      (paths (car rest) (cdr rest) min max))))))
384        (paths (car possible) (cdr possible) mincost maxcost))))
```

2. `*tsp*`

   Now, a variable to hold the newly-created TSP.

```
385  (defvar *tsp* (generate-tsp 5 1 10))
```

3. solve-tsp

   This function uses `learn-string` in order to solve a TSP. The fitness function is defined below.

```
386  (defun solve-tsp (&key tsp (n 5) (min 1) (max 10)
387          (individual-kind :alphanumeric+space)
388          until-best-fitness until-average-fitness
389          for until-generation
390          (file "learn-string-results.dat")
391          (status-every 50)
392          (breeding-pairs 4)
393          (mutation-rate 0.1)
394          (crossover-rate 0.1)
395          (population-size 50)
396          (silent nil)
397                         ga)
398
399    (setq *tsp*
400      (if tsp
401        tsp
402        (setq tsp (generate-tsp n min max))))
403
404    (let ((cities (remove-duplicates (apply #'append
405     (mapcar #'(lambda (a)
406         (list (car a) (cadr a)))
407     tsp)))))
408      (unless ga
409        (setq *ga* (make-instance 'ga
410    :breeding-pairs breeding-pairs
411    :individual-kind (copy-list cities)
412            :individual-size (1+ (length cities))
413    :breeding-pairs breeding-pairs
414    :mutation-rate mutation-rate
415    :crossover-rate crossover-rate
416    :population-size population-size
417    :silent silent
```

```
418    :fitness-function #'tsp-fitness-function
419    )))
420
421      (run *ga* :status-every status-every :file file
422    :until-generation until-generation
423    :until-best-fitness until-best-fitness
424    :until-average-fitness until-average-fitness
425    :for for)))
```

4. `tsp-fitness-function`

   This simply calculates the cost of the current path and returns it as a negative number. Thus shorter paths will be less negative, hence fitter. If a path does not begin and end at the same place, then it's penalized, as it is if a city occurs more than once (other than start and end).

```
426  (defvar *tsp-non-circuit-penalty* most-positive-fixnum)
427  (defvar *tsp-duplicate-city-penalty* most-positive-fixnum)
428
429  (defun tsp-fitness-function (path)
430    (cond
431     ((not (eql (car path) (car (last path))))
432      (- *tsp-non-circuit-penalty*))
433     ((< (length (remove-duplicates path)) (1- (length path)))
434      (- *tsp-duplicate-city-penalty*))
435     (t
436      (- (path-cost path *tsp*)))))
437
438
439  (defun path-cost (path &optional (tsp *tsp*))
440    (cond
441     ((< (length path) 2) 0)
442     (t (+ (segment-cost (car path) (cadr path) tsp)
443    (path-cost (cdr path) tsp)))))
444
445  (defun segment-cost (a b &optional (tsp *tsp*))
446    (or (third (car (member (list a b) tsp
447     :test #'(lambda (target candidate)
448       (or (and (eql (car target) (car candidate))
449        (eql (cadr target) (cadr candidate)))
450   (and (eql (car target) (cadr candidate))
451        (eql (cadr target) (car candidate)))))))
452        most-positive-fixnum))
```

Index

TSP (variable), 19
alphabet-bitstring (variable), 7
alphabet-lower-case (variable), 6
alphabet-map (variable), 7
alphabet-numerals (variable), 7
alphabet-punctuation (variable), 7
alphabet-space (variable), 7
alphabet-special (variable), 7
alphabet-upper-case (variable), 6
ga (variable), 8
stats (variable), 8
target (variable), 8
tsp-duplicate-city-penalty (variable), 20
tsp-non-circuit-penalty (variable), 20

add-individuals (method), 13
average-fitness (method), 11

best-answer (method), 17
best-answers (method), 18
best-fitness (method), 11

choose? (function), 13
compute-fitness (method), 9
compute-fitnesses (method), 9
condition-string (method), 6
crossover (method), 12

display (method), 10

fittest (method), 13

ga (class), 4
generate-tsp (function), 18
genome-as-string (method), 11
get-alphabet (function), 8

individual (class), 3

learn-string (function), 16

mutate (method), 13

next-generation (method), 14

path-cost (function), 20
population (class), 4

random-genome (method), 9
reproduce (method), 11
run (method), 15

segment-cost (function), 20
solve-tsp (function), 19
sort-by-fitness (method), 10
stats (function), 17
status (method), 10
string-match-fitness (function), 14

tsp-fitness-function (function), 20

worst-fitness (method), 11