

picoGA: A Vectorized Implementation of a Genetic Algorithm in Matlab

Keki Burjorjee
DEMO Lab
Computer Science Department
Brandeis University
Waltham, MA 02454
kekib@cs.brandeis.edu

1. INTRODUCTION

This document describes picoGA, our submission to the Tiny GA programming competition at GECCO 2006. picoGA is optimized to have small code size and a small memory footprint. It is implemented in the Matlab programming language. Matlab is an interpreted language that is optimized for performing operations on arrays. Loops, especially nested loops, tend to run slowly in Matlab. It is possible however to significantly improve the performance of Matlab programs by converting loops into array operations. This process is called vectorization (see [1] for further information). Matlab provides a rich set of functions, and many expressive indexing schemes that make it possible to vectorize code. Vectorized code not only runs faster, it also requires fewer lines of code.

A drawback of vectorization is that it sometimes results in the consumption of more memory. Another drawback is that vectorized code can be harder to understand for users who are new to Matlab programming. In our submission we have sought to create a GA implementation that balances the costs and benefits of vectorization. There are some loops that we could have been vectorized to reduce the number of lines of code and to improve the performance of picoGA, however because the vectorization of these loops would have resulted in a drastic increase in the memory footprint, and because performance is not one of the criteria of the competition we decided to leave these loops unchanged. We will point out places where we made such decisions, and show what the vectorized code might have been in each case.

In other places vectorization allowed for a big reduction in program size without drastically affecting the memory footprint. In such cases we have vectorized the code. For sophisticated users of Matlab, vectorization oftentimes improves the clarity of code, however it can make matters more difficult for others. Hence, in this document, we will explain how our vectorized code works.

In the next section we describe features of our code that are relevant to the Tiny GA competition. In the following section we visit places in the code where we have implemented vectorization, and other places where we have avoided it. We will assume that readers are familiar with the vector indexing and logical indexing syntax of Matlab. For an introduction to this syntax see [2]

2. PROGRAM FEATURES

picoGA can be run from the Matlab command line

as follows:

```
>> picoGA(<randomSeed>)
```

If a random seed is not passed to picoGA it will use the system clock to initialize the random number generator.

The total number of non-commented lines of code is 56, with no line exceeding 76 characters. Of these, 7 lines of code are devoted purely to displaying system information — the behavior of the code will not change if these lines are commented out. The real work of the algorithm thus takes 49 lines of code. Calculating the fitness of individuals in the population takes 6 lines, fitness proportional selection takes 8 lines. Reproduction, uniform recombination, and mutation take a mere 15 lines.

The size of the source code file (which includes comments) is 4,289 bytes.

Table 1. shows the memory that is used by the variables in an executing picoGA system. Using the information listed in that table we calculate that the a running picoGA system uses approximately

$$4 * \text{POPSIZE} * \text{LEN} + 260 * \text{POPSIZE} + 8 * \text{LEN} + 3 \text{ bits}$$

. This does not include the overhead of code execution — stack memory used during function calls etc.

We trust that the memory footprint of interpreters and virtual machines will not be included in the calculation of the memory footprints of interpreted systems that are submitted to the tinyGA competition. If this is not the case then it seems inappropriate that a warning was not issued on the competition website to authors of interpreted systems. We therefore trust that the judges will *not* include the memory footprint of Matlab in their calculation of the memory footprint of picoGA; doing so would obviously disqualify picoGA from this competition, but it would be unfair.

3. COMMENTS

The population is a matrix with dimensions $\text{POPSIZE} \times \text{LEN}$. Each chromosome in this population is a row in the matrix. Implementing the population in this way is the natural choice, and it allows us to use the logical and vector indexing capabilities of Matlab to vectorize recombination, reproduction and mutation.

Datatype	Variables
POPSIZE×LEN booleans	pop,newPop,masks,secondMate
POPSIZE×1 booleans	reprodIndices
(POPSIZE+1)×1 booleans	index, two temporary variables
1×POPSIZE doubles	fitnessVals, cumNormFitnessVals
POPSIZE×2 doubles	matingIndices
1×LEN chars	bestIndiv

Table 1: A table showing the memory usage of variables in picoGA. A Matlab double is 64 bits. To the best of our knowledge, a Matlab boolean (logical) is a single bit, and a Matlab char is 8 bits.

3.1 Cases in Which Vectorization was not Implemented

We could have eliminated the subfunction

```
generateRandomBooleanMatrix
```

and saved 5 lines of code by using

```
rand(POPSIZE,LEN)<pTrue
```

in place of each call of the form

```
generateRandomBooleanMatrix(POPSIZE,LEN, pTrue)
```

However, `rand(POPSIZE,LEN)` causes the internal generation of a matrix of `POPSIZE×LEN` 64 bit doubles, which significantly increases the memory footprint of the system. We therefore decided not to vectorize the code in this case.

The population itself could have been implemented as a `POPSIZE×LEN` matrix of 64 bit doubles. Such an implementation would reduce the number of lines necessary for calculating the fitness from 6 to 1 — `fitnessVals=sum(pop,2)`. Once again we eschew this choice because of the large increase in memory footprint that it entails.

3.2 Vectorization in the Fitness Proportional Selection Code

The use of logical indexing in lines 71 and 74 of the fitness proportional (i.e. roulette wheel) selection code is best described by example: let `POPSIZE = 5`, and suppose the cumulative normalized fitness values array `cumNormFitnessVals` and the roulette ball variable `k` are as follows:

```
cumNormFitnessVals =
    0.2468    0.4557    0.6392    0.8544    1.0000
```

```
k =
    0.5438
```

Given these values the third chromosome in `pop` should be added to `newPop` since $0.4557 < 0.5438 \leq 0.6392$. The following lines show how this is implemented using vectorization.

```
[cumNormFitnessVals>=k false] =
    0    0    1    1    1    0
```

```
[ true  cumNormFitnessVals<k] =
    1    1    1    0    0    0
```

```
index =
    0    0    1    0    0    0
```

The logical array `index` has a 1 in the third position and zeros everywhere else. `pop(index(1:POPSIZE),:)` will therefore be the third chromosome in `pop`.

3.3 Vectorization of Recombination, Reproduction and Mutation

Reproduction and uniform recombination are implemented using a combination of logical indexing and vector indexing. Line 82 initializes `matingIndices` to be a `POPSIZE×2` array of randomly chosen mating pairs where each mating pair is a pair of indices for the first dimension of the `pop` array. Line 83 uses vector indexing to set `newPop` to be the first mates of each pair in `matingIndices`. In Line 84 an array of masks is generated. `masks(1,:)` is the mask for the first mating pair, `masks(2,:)` is the mask for the second mating pair, and so on. Next a `POPSIZE×1` array of boolean values `reprodIndices` is created such that 1's occur with probability given by the reproduction probability `1-CROSSOVER_PROB`. A 1 at some index in `reprodIndices` indicates that the first chromosome of the mating pair at that index should be reproduced. This is implemented by using logical indexing to set appropriate rows of the `masks` array to zero. Logical indexing with the `masks` array is used in line 88 to implement reproduction and uniform crossover simultaneously. Mutation of the resulting population is implemented using logical indexing with a new boolean `masks` array (in which the probability of a 1 is `PMUT`) and the `xor` function.

4. AN INTERESTING OBSERVATION

When picoGA is run with the following control parameters:

```
LEN=60
POPSIZE=200
GENERATIONS=100
CROSSOVER_PROB=.8
PMUT=.5/LEN
```

it does not find the maximum individual of the one-max problem in the allotted number of generations. However, if in each generation the minimum fitness value is subtracted from all fitness values before fitness proportional selection then the the maximum individual is always found in about 30 steps. We call the subtraction of the minimum fitness value from all fitness values *subtractive transformation*. It can be implemented by uncommenting line 53:

```
%fitnessVals=fitnessVals-min(fitnessVals)+realmin;
```

Subtractive transformation allows fitness proportional selection to be sensitive to fitness differences even when the fitness values of individuals are large. Fitness proportional selection without subtractive transformation does not have this property.

5. REFERENCES

1. http://www.mathworks.de/access/helpdesk/help/techdoc/matlab_prog/f8-784135.html
2. <http://www.mathworks.com/company/newsletters/digest/sept01/matrix.html>