

The Effects of Parameter Variation on Genetic Algorithm Performance

Daniel Kunkle

March 25, 2002

1 Introduction

A basic genetic algorithm, like most algorithms, has a number of parameters that will affect its performance. Different sets of parameters may be optimal when working with different problem sets. This report examines the effects of varying some parameters of a genetic algorithm while solving a very simple problem, in this case maximizing the number of 1s in the genome.

For this experiment, a genetic algorithm based in part on implementation described in Chapter 3 of Goldberg, D. (1989) "Genetic Algorithms in Search, Optimization, and Machine Learning" was implemented in MATLAB. The following pseudocode describes at high level how the implemented GA works.

```
create new population of n random genomes
while (a solution has not been found) AND (the maximum number of generations
has not been reached)
    calculate the fitness of each member in the population
    if a perfect solution has been found, stop and take solution
    else, continue
    keep track of best individual so far
    create a new generation of individuals
    for i = 1 to n/2
        select two "parent" individuals from old population probabilistically
        based on the calculated fitness
        perform a crossover on parents with probability pCross
        perform a mutation on parents with probability pMutation
        place the two individuals into the new population (next generation)
    end for
end while
```

The code for this implementation is included at the end of this document as Appendix A. The pseudocode above corresponds to the function "ga".

2 GA Parameters

The basic parameters of the genetic algorithm that can be varied are: the population size, the probability of crossover, and the probability of mutation. These correspond to n, pCross, and pMutation in the Introduction.

All of these values control the trade-off between exploration and exploitation in some way. If the population size is very large there will be more diversity and therefore greater exploration, but it comes with a computational price. If the probability of crossover is higher more exploration will take place, if it is lower more individuals will be copied exactly as-is into the new generation. Similarly, if the probability of mutation is low the search will tend to exploit the good genomes it has already found rather than explore new alternatives.

A genetic algorithm is a search mechanism, but finding the optimal values for the parameters of the GA is itself a search problem. In this experiment the search procedure is simply to try a subset of values and note the best one. More advanced search mechanisms could be employed to find optimal parameters, such as using a meta-GA to control a population of GAs.

3 Results

A very simple test problem was used to explore the effects of parameter variation. The fitness function of a binary genome is simply the sum of all of the bits, or equivalently, the number of 1s in the genome. For the experiments here the genome was of length 15 (a relatively short genome to keep the processing time reasonable).

The probability of crossover was kept constant at .6, a value suggested by Goldberg. The probability of mutation was varied from 0 to 1 in increments of .05. The population size was varied from 10 to 200 in increments of 10. The performance measurement is the number of fitness evaluations performed to find the solution. A maximum number of fitness evaluations of 4000 was specified for each search. The number of fitness evaluations required by the GA for each pair of parameters was averaged over ten runs. The results are shown by figure 1.

The best result found was an average of 768 fitness evaluations to find a solution with a population size of 40 and a mutation rate of .25. Other high performers are in the same area, with population sizes around 20 to 60 and mutation rates around .1 to .3. The spike in the graph in the back corner is due to the lack of exploration. With very low mutation rates and population sizes the individuals in the population quickly converge to a set of identical genomes. With no mutation what-so-ever this guarantees that no more exploration is possible.

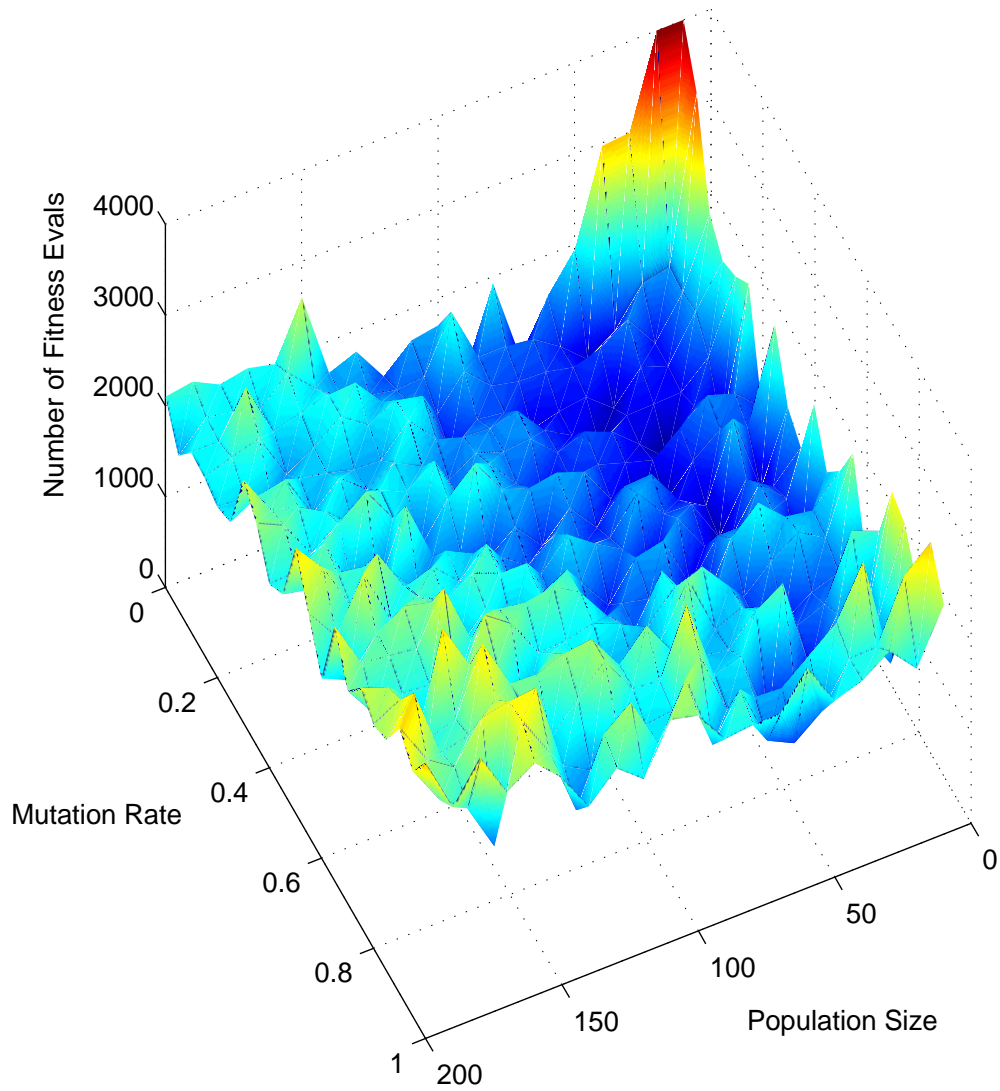


Figure 1: Number of Fitness Evaluation for Different Population Sizes and Mutation Rates

Appendix A – MATLAB Implementation

```
function children = cross ( parents )

% parents: matrix with two rows, one for each parents genome

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cross.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
% Takes a matrix with two parents and returns one with two children created
% by crossover of the parents
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[ x, genLen ] = size( parents ); % length of genome

min = 1;
max = genLen * 1;
cutPoint = randint( min, max );

children = parents;
children( 1, 1:cutPoint ) = parents( 2, 1:cutPoint );
children( 2, 1:cutPoint ) = parents( 1, 1:cutPoint );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function f = fitness ( pop )

% pop: the population to return a fitness vector for (each row is an individual)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% fitness.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
% A very simple GA Fitness function - returns sum of bits normalized to [0,1]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[ popSize, genomeLen ] = size( pop );
f = sum( pop' ) ./ genomeLen;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function fitEvals = ga ( popSize, genomeLen, maxFitEvals, pCross, pMut )

% popSize: size of population (must be divisible by 2)
% genomeLen: length of genome
```

```

% mexFitEvals: maximum number of fitness evaluations to perform (should be
%      multiple of popSize)
% pCross: probability of crossing over new parents each time
% pMut: probability of each new string being mutated

% returns -- the number of fitness evaluations processed to find solution (or
%      to reach maxGen)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ga.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
% A basic generation-based genetic algorithm. Based in part on implementation
% described in Chapter 3 of Goldberg, D. (1989) "Genetic Algorithms in Search,
% Optimization, and Machine Learning"
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

maxGen = floor( maxFitEvals / popSize );
genNum = 0; % number generations processed
perfectFound = 0; % true if a perfect individual is found
bestIndv = []; % genome of best individual
highestFit = 0; % fitness of best individual

% if popSize is not divisible by 2, make it so
if ( mod( popSize, 2 ) == 1 )
    popSize = popSize + 1;
    disp( 'Population Size must be divisible by 2' );
    disp( 'It has been incremented to make it so' );
end

% create random population of individuals
pop = hardlim( rand( popSize, genomeLen ) - .5 );
newPop = pop;

% search until perfect individual is found or maxGen is reached

while ( ~perfectFound & genNum < maxGen )

    pop = newPop;
    % calculate each members fitness (fitness being in the range [0,1])
    fit = fitness( pop );
    genNum = genNum + 1;

    % check for perfect individual
    perfectIndv = find( fit == 1 );
    if ( ~isempty( perfectIndv ) )
        % if there is a perfect individual take him
        perfectFound = 1;
        bestIndv = pop( perfectIndv(1), : );
        highestFit = 1;
    else

```

```

% if there is no perfect individual find the best individual so far
% and make a new population
[ maxFitInPop, maxIndvInPop ] = max( fit );
if ( maxFitInPop > highestFit )
    highestFit = maxFitInPop;
    bestIndv = pop( maxIndvInPop, : );
end

% select individuals from pop using a probablistic "roulette wheel"
% each two individuals will either be replicated or crossed over
% into the new population with some probability of mutation

newPop = [];
for i = 1:popSize/2
    indivsIndex = rws( fit, 2 );
    indivs(1,:) = pop( indivsIndex(1), : );
    indivs(2,:) = pop( indivsIndex(2), : );
    crossChance = rand;
    if ( crossChance < pCross )
        indivs = cross( indivs );
    end
    mutChance = rand;
    if ( mutChance < pMut )
        indivs(1,:) = mut( indivs(1,:) );
    end
    mutChance = rand;
    if ( mutChance < pMut )
        indivs(2,:) = mut( indivs(2,:) );
    end
    newPop( i*2-1:i*2, 1:genomeLen ) = indivs;
end
end
end

fitEvals = genNum * popSize;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function newGenome = mut ( genome )

% parents: matrix with two rows, one for each parents genome

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% mut.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
% perform a mutation on a genome by flipping one of the bits
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
[ x, genLen ] = size( genome );
flipPoint = randint( 1, genLen );
newGenome = genome;
newGenome( flipPoint ) = ~newGenome( flipPoint );
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
function r = randint ( min, max )
```

```
% min: low end of random range
% max: high end of random range
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
% randint.m
% Dan Kunkle
% March, 2002
%
```

```
% Returns a random integer in the range (min, max)
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
r = floor( min + ( max - min + 1 ) * rand );
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
function i = rws ( fit, num )
```

```
% fit: fitnesses of population of individuals to select from
% num: number of members of population to select
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
% rws.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
```

```
% A roulette-wheel selection mechanism
```

```
%%%%%%%%%%
%%%%%%%%%%
```

```
[ x, popSize ] = size( fit ); % size of population
i = zeros( 1, num ); % indicies of individuals to select
```

```
for mem = 1:num
```

```
    partsum = 0; % fitness sum counter
    totFit = sum( fit ); % sum of all fitnesses
    r = rand * totFit; % random point on wheel to select
```

```
    while ( ( partsum < r ) & ( i <= popSize ) )
```

```

        i(mem) = i(mem) + 1;
        partsum = partsum + fit(i(mem));
    end

end

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

function [] = testPopMut ( numRuns, popLow, popHigh, popStep, ...
    mutLow, mutHigh, mutStep, fileName )

```

```

% numRuns: number of runs to average results over
% popLow: low end of population sizes to test
% popHigh: high end of population sizes to test
% popStep: increment of populations to test
% mutLow: low end of mutation rates to test
% mutHigh: high end of mutation rates to test
% mutStep: increments of mutation rates to test
% fileName: name of file to store data and figure in

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% testPopMut.m
% Dan Kunkle
% Genetic Algorithms
% March, 2002
%
% A test of a range of GA parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

% HEADER FOR GA FUCNTION
% function fitEvals = ga ( popSize, genomeLen, maxFitEvals, pCross, pMut )

[ pops, muts ] = meshgrid( popLow:popStep:popHigh, mutLow:mutStep:mutHigh );

[ rows, cols ] = size( pops );
numGaTrialsToRun = numRuns * rows * cols

fitEvals = zeros( rows, cols );

genomeLen = 15
maxFitEvals = 4000
pCross = .6

bestTime = inf;
bestPopSize = -1;
bestMutRate = -1;

for i = 1:rows
    for j = 1:cols
        fitEvalsTot = 0;
        for n = 1:numRuns

```

```

        fitEvalsTot = fitEvalsTot + ga( pops( i, j ), genomeLen, ...
            maxFitEvals, pCross, muts( i, j ) );

        %PROGRESS UPDATE
        trialNum = ((i-1) * cols * numRuns) + ((j-1) * numRuns) + n;
        if ( mod( trialNum, 10 ) == 0 )
            disp( trialNum );
        end
    end
    fitEvalsAvg = fitEvalsTot / numRuns;
    fitEvals( i, j ) = fitEvalsAvg;
    if ( fitEvalsAvg < bestTime )
        bestTime = fitEvalsAvg;
        bestPopSize = pops( i, j );
        bestMutRate = muts( i, j );
    end
end
end

h1 = figure;
surf( pops, muts, fitEvals );
shading interp
title( 'Analysis of GA Params' );
xlabel( 'Population Size' );
ylabel( 'Mutation Rate' );
zlabel( 'Number of Fitness Evals' );

% save data
dataFileName = [ fileName '.data' ];
dlmwrite( dataFileName, fitEvals, '\t' );
popDataFileName = [ fileName '.pops' ];
dlmwrite( popDataFileName, pops, '\t' );
mutDataFileName = [ fileName '.muts' ];
dlmwrite( mutDataFileName, muts, '\t' );
figureFileName = [ fileName '.fig' ];
saveas( h1, figureFileName );
propertiesFileName = [ fileName '.prop' ];
fid = fopen( propertiesFileName, 'w' );
fprintf( fid, 'numRuns: %d\n', numRuns );
fprintf( fid, 'popLow: %d\n', popLow );
fprintf( fid, 'popHigh: %d\n', popHigh );
fprintf( fid, 'popStep: %d\n', popStep );
fprintf( fid, 'mutLow: %d\n', mutLow );
fprintf( fid, 'mutHigh: %d\n', mutHigh );
fprintf( fid, 'mutStep: %d\n', mutStep );
fprintf( fid, 'genomeLen: %d\n', genomeLen );
fprintf( fid, 'maxFitEvals: %d\n', maxFitEvals );
fprintf( fid, 'pCross: %d\n', pCross );
fprintf( fid, '\n', 0 );
fprintf( fid, 'BEST TIME: %d\n', bestTime );
fprintf( fid, 'BEST POP SIZE: %d\n', bestPopSize );
fprintf( fid, 'BEST MUT RATE: %d\n', bestMutRate );
fclose( fid );

```