

ECTA Homework 1

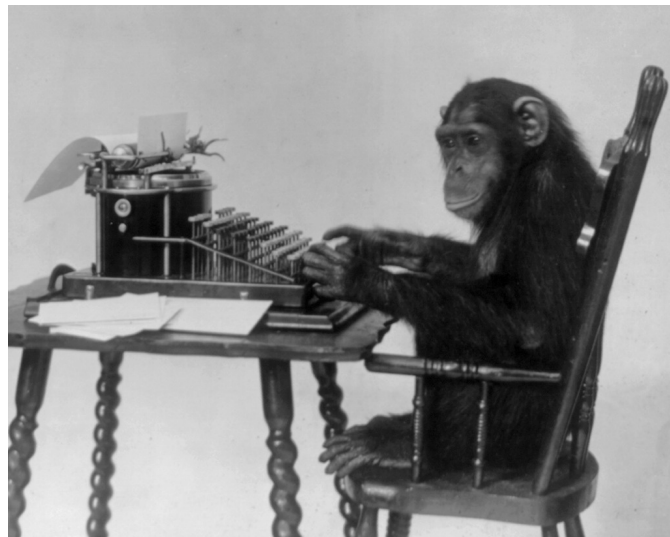
Genetic Algorithms

and

Infinite Monkeys

Djordje Vukcevic, djordje.vukcevic@smail.inf.h-brs.de
Supriya Vadiraj, supriya.vadiraj@smail.inf.h-brs.de

April 26, 2018



The infinite monkey theorem states that if a chimpanzee hits keys at random on a typewriter for an infinite amount of time, it will eventually type the complete works of William Shakespeare. Is this what evolution is doing? It has been argued that the genetic mutations required to move from a single cell to multicellular life are as unlikely as a monkey typing Hamlet's soliloquy. But is evolution just a monkey banging on a typewriter?

1 Assignment Description

1. Build a simple Genetic Algorithm and test the effect of each component
 - Mutation
 - Crossover
 - Elitism
2. Answer the question, "is evolution just a monkey banging on a typewriter?"

- Grading Scheme

- ☐ Code a GA (40 pts)
 - ☐ Selection (10pts)
 - ☐ Crossover (10pts)
 - ☐ mutation (10pts)
 - ☐ Elitism (10pts)
- ☐ GA Component Comparisons (40 pts)
 - ☐ Comparison vs. Standard Implementation (5 pts)
 - ☐ Comparison of Components (5 pts)
 - ☐ Short Answer #1 (10 pts)
 - ☐ Short Answer #2 (10 pts)
 - ☐ Short Answer #3 (10 pts)
- ☐ GA vs Monkey (20 pts)
 - ☐ Solve the soliloquy (10 pts)
 - ☐ Brute force calculation (5 pts)
 - ☐ Short Answer #4 (5 pts)

2 Submission Instructions

Follow along with the instructions in this PDF, filling in your own code, data, and observations as noted. Your own data should be inserted into the latex code of the PDF and recompiled. All code must be done in MATLAB. The basic structure of the code and fitness function are provided, but all code should be submitted as a separate zipped file in LEA. Relevant sections of code can be inserted directly into this document using the mcode latex package. This package is attached with documentation, and in this document I have provided usage examples.

To be perfectly clear we expect two submissions to LEA:

1. 1 PDF (report) – a modified version of this PDF, with your own code snippets, figures, and responses inserted
2. 1 ZIP (code and data) – a .zip file containing all code use to run experiments (.m files) *and* resulting data as a .mat file

3 Assignment Overview

the Task

Like our monkey, your Genetic Algorithm will be tested as to how closely it can reproduce Shakespeare. Two benchmarks are given: `hamletQuote` and `hamletSoliloquy`. These functions take one or more genes of length 18 for the quote or 1446 for the soliloquy and return a fitness value which corresponds to the number of letters that match the target text.

One gene is a number between 0 and 27, corresponding to a space (0), letters a-z (1-26), and a new line (27).

the Algorithm

I have created the basic structure of the GA for you. The magic happens in the loop here in `monkeyGa.m`:

```
[fitMax(iGen), iBest] = max(fitness); % 1st output is the max ...
    value, 2nd the index of that max value
fitMed(iGen)          = median(fitness);
best(:,iGen)          = pop(iBest,:);

%% Evolutionary Operators

% Selection -- Returns [MX2] indices of parents
parentIds = my_selection(fitness, p); % Returns indices of parents

% Crossover -- Returns children of selected parents
children  = my_crossover(pop, parentIds, p);

% Mutation -- Applies mutation to newly created children
children  = my_mutation(children, p);

% Elitism -- Select best individual(s) to continue unchanged
eliteIds  = my_elitism(fitness, p);

% Create new population -- Combine new children and elite(s)
newPop    = [pop(eliteIds,:); children];
```

It will be your job to implement each of the evolutionary operators and measure how they effect performance of the algorithm on the `hamletQuote` task.

Running the Algorithm

To run the algorithm and view the results, you can use the snippet provided at the start of `monkeyExperiment.m`:

```
%% Run the algorithm once
clear;
p = monkeyGa('hamletQuote'); % Set hyperparameters
output = monkeyGa('hamletQuote',p); % Run with hyperparameters

% View Result
gene2text(output.best(:,end))
plot([output.fitMax; output.fitMed], 'LineWidth', 3);
legend('Max Fitness', 'Median Fitness', 'Location', 'NorthWest');
xlabel('Generations'); ylabel('Fitness'); set(gca, 'FontSize', 16);
title('Performance on Hamlet Task')
```

To run a section in matlab (a code block marked by `%%`, with the cursor inside the code block click the ‘Run Section’ button in the editor portion of the ribbon, or more simply hit ‘CTRL + Enter’). Run it a few times. As the only operator which is implemented is initialization, it will give you a pretty terrible result.

Comparing Algorithms

As evolutionary algorithms are based on stochastic processes, they will not perform the same every time. Whenever a comparison between two algorithms or algorithm settings is made, it *must* be a comparison over several runs. Comparisons between runs must take into account the effect of randomness, including significance of results (how likely the result is to be because of chance).

4 The Assignment

4.1 Coding a simple GA

Begin by implementing the four given genetic operators, replacing the filler code with your own. The expected inputs and outputs, as well as hints as how to perform each operation are included within the code. Please put your code in the report here using the given ‘firstline/lastline’ syntax in the L^AT_EX.

Don’t overthink it! Each of these can be done in less than 10 lines!

1. Tournament Selection

```
%----- BEGIN CODE -----
%Initialize 100x2 matrix of parent indices
parentIds = NaN([p.popSize 2]);

%Get 200 winners and make 100 pairs of them
for i=1:p.popSize*2
    for k=1:2
        %Get random indeces of individuals.
        %Syntax: range, [size of matrix]
        randomPair = randi(p.popSize,[2,1]);

        %choose index of max between two rows
        [winner_value, winner_index]= max(fitness(randomPair));

        %Fill out parentIds matrix...syntax for matrix(i):
        %goes over whole first row, than over whole second ...
        %row, and so on.
        %Fill 100x2 matrix -> 100 pairs of 200 parents.
        parentIds(i) = randomPair(winner_index);
    end
end
%----- END OF CODE -----
```

2. Crossover

```
%----- BEGIN CODE -----
%Initialize children matrix by assigning only first individual ...
  of all pairs
children = pop(parentIds(:,1) ,:);
for child=1:p.popSize
    %Should crossover be perform or not
    % rand(size of vector/matrix)
    crossover_decision = rand(1) < p.crossProb;

    if crossover_decision
        %position of a gene can be from 1 to 17
        % randi([range], size of vector/matrix)
        cross_point = randi(p.nGenes-1);
        first_part = pop(parentIds(child,2),1:cross_point);
        second_part = ...
            pop(parentIds(child,1), (cross_point+1):p.nGenes);
        %concatenate partA indices/genes with partB ...
        %indices/genes from pop
        children(child,:) = [first_part second_part];
    end
end
%----- END OF CODE -----
```

3. Mutation

```
%----- BEGIN CODE -----
%Initialize children matrix and seed of random generator
children = input.children; rng;
%Iterate over all childrens and their genes
for child = 1:p.popSize
    gene_mutate = rand(1,p.nGenes);
    for gene = 1:p.nGenes
        if(gene_mutate(gene) < p.mutProb)
            children(child,gene) = randi([0 27],1);
        end
    end
end
%----- END OF CODE -----
```

4. Elitism

```
%----- BEGIN CODE -----
elite_individuals = p.popSize* p.elitePerc;

[sorted_values,sorted_ids]= sort(fitness,'descend');

eliteIds = sorted_ids(1:ceil(elite_individuals));
%----- END OF CODE -----
```

4.2 Ablation Study

One common technique for better understanding an algorithm is remove each component and see the result. What happens when we don't use elitism or we skip crossover? In this section we test a few combinations.

4.2.1 Comparing Algorithms

Provided are versions of each operator with the prefix 'adam' instead of 'my'. These can be used to validate your own results. I included a version which uses them in the file 'adamGa', which is exactly the same except this part:

```
% Selection -- Returns [MX2] indices of parents
parentIds = adam_selection(fitness, p); % Returns indices of ...
      parents

% Crossover -- Returns children of selected parents
children = adam_crossover(pop, parentIds, p);

% Mutation -- Applies mutation to newly created children
children = adam_mutation(children, p);

% Elitism -- Select best individual(s) to continue unchanged
eliteIds = adam_elitism(fitness, p);
```

As this is a stochastic algorithm to get a fair comparison we should run the algorithm multiple times and compare statistically. Lets use all the cores on your computer to do this as fast as possible using a `parfor` loop. This is just like a `for` loop, except it runs each iteration on a different core. Get the result of 20 runs of your code and mine and save it to disk:

```
% changing 'for' to 'parfor'.
clear; p = monkeyGa('hamletQuote');
parfor iExp = 1:20
    output = adamGa('hamletQuote',p);
    fitness(iExp,:) = output.fitMax;
end
standardResult = fitness;

parfor iExp = 1:20
    output = monkeyGa('hamletQuote',p);
    fitness(iExp,:) = output.fitMax;
end
myResult = fitness;
save('runData.mat','standardResult','myResult')
```


With this data saved you can compare the two algorithms and compute the significance of the comparison. I have given you a few helper functions:

```
load('runData.mat')
gens = 1:length(standardResult);

% Get Significance of comparisons
fit1 = standardResult; fit2 = myResult;
[p,h] = sigPerGen(fit1,fit2);

% Plot results at every generation
figure(2); clf; hold on; C = parula(8); % Create figures and color map

% Plot Significance at every generations
hS1 = scatter(gens(~h),ones(1,sum(~h))*19,20,C(1,:), 'filled','s');
hS2 = scatter(gens(h),ones(1,sum(h))*19,20,C(7,:), 'filled','s');

% Plot median and percentiles
[hLine(1), hFill(1)] = percPlot(gens,fit1 ,C(2,:));
[hLine(3), hFill(2)] = percPlot(gens,fit2,C(5,:));

% Label and make pretty
hLeg = legend([hFill hS1 hS2], 'Baseline', 'No Mutation', 'p > 0.05', ...
    'p < 0.05', 'Location', 'SouthEast');
```

Standard algorithm vs our implementation

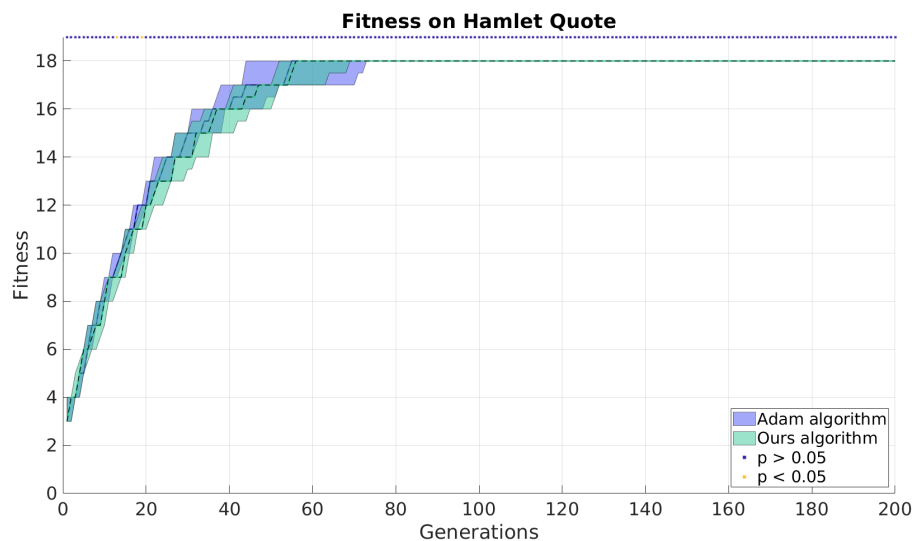


Figure 1: Median performance of Adam's algorithm (purple) compared with median performance of our (green) algorithm.

This plot shows the median performance at each generation (dashed lines) of each algorithm along with their upper and lower quartiles. Indicated at the top is the probability that the two algorithms are the same. Unsurprisingly, both runs of the same algorithm are statistically the same. Replace this plot with one of your own creation, comparing my code with your own implementation, to ensure that your code is working.

Comparison of our implementation with various components removed

Perform the following comparisons of your algorithm with various components removed and replace the plots with your own, this can be done by replacing the functions in the code and saving the result (e.g. replacing the `my_crossover` function with the `no_crossover` function: (1) Your full implementation vs. No crossover, (2) Your full implementation vs. No mutation, (3) No crossover vs. No crossover AND no elitism, and (4) No mutation vs. No mutation AND no elitism.

GA performance when operators are removed:

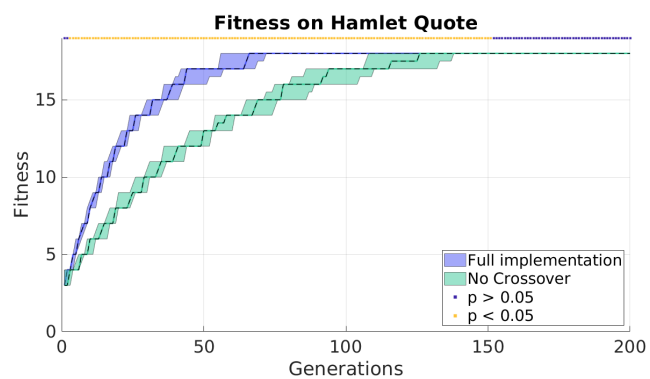


Figure 2: No Crossover

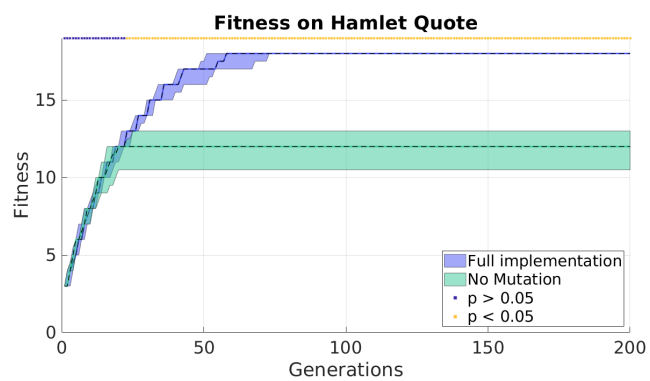


Figure 3: No Mutation

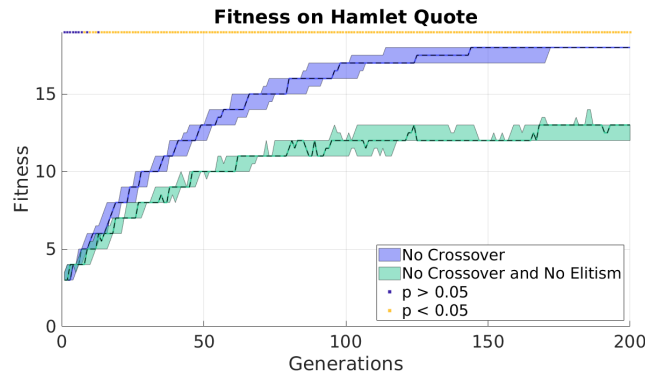


Figure 4: No Crossover vs. No Crossover and No Elitism

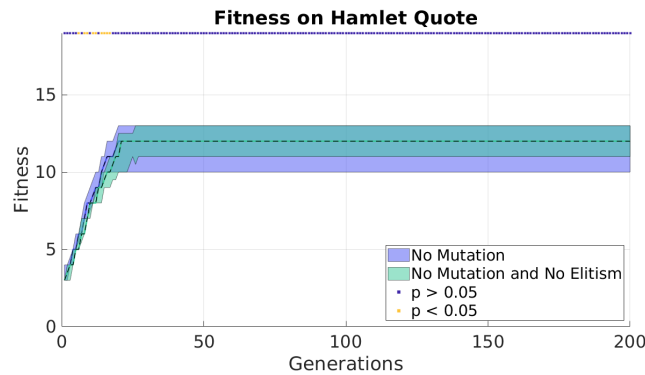


Figure 5: No mutation vs. No mutation AND no elitism.

4.2.2 Analyzing the Results

1. Describe the main purposes of crossover and of mutation, how do your results illustrate their operation?

- *Crossover and Mutation perform two important roles in genetic algorithm.*
- *Crossover is a convergence operation which is intended to pull the population towards a local minimum/maximum.*
- *Mutation is a divergence operation. It is intended to occasionally break one or more members of a population out of a local minimum/-maximum space and potentially discover a better minimum/maximum space.*
- *As described in our result in the comparison graph, the aim is to*

bring the population to convergence.

- *As shown in the graphs in our implementation , by performing only crossover, the fitness takes longer generations/time to reach the solution.*
- *By performing without mutation, it shows that the fitness reaches a particular level and gets stagnant/constant.*

References

- https://www.researchgate.net/post/Is_crossover_a_pillar_of_genetic_algorithms
- https://www.researchgate.net/post/What_is_the_role_of_mutation_and_crossover_probability_in_Genet

2. **When only crossover is used the problem is not typically solved. Why? Could you devise an experiment that would support your explanation? One in which crossover could solve the problem every time?**

- *A Genetic Algorithm is supposed to have the possibility to explore the problem space. Mutation is a small step in exploring, whereas Crossover is potentially a larger step if crossover happens between individuals that are far apart. However, the larger the step is, the higher probability of producing a failure. When there is only crossover performed ,the algorithm selects genes from the individuals in the initial population and recombines them. The algorithm cannot create any new genes because there is no mutation.*
- *Crossover is a convergence operation which is intended to pull the population towards a local minimum/maximum.*

References

- https://www.researchgate.net/post/Is_crossover_a_pillar_of_genetic_algorithms
- https://www.researchgate.net/post/What_is_the_role_of_mutation_and_crossover_probability_in_Genet
- <https://stackoverflow.com/questions/17882992/genetic-algorithm-is-elitism-necessary-if-crossover-is-low>

3. **Describe the benefits of elitism in the crossover and mutation only cases.**

- *Even with a high crossover rate, and a selection process which selects higher chromosome to crossover there is always a chance (no matter how small) that you will lose the best chromosome from one generation to the next. Elitism means that once a good result is found, it will always remain in the population until a better result is found.*

- *No matter how small the chance of losing your best result, it is always a good idea to employ a safe guard to ensure that it won't ever happen.*

References

- <https://stackoverflow.com/questions/17882992/genetic-algorithm-is-elitism-necessary-if-crossover-is-low>

4.3 Monkeys on a Typewriter

4.3.1 Using the GA

Now time to test your algorithm on the entire soliloquy. Is it really better than just banging on a typewriter? Switch out the fitness function and give the whole speech a try. It might take a little time, you may have to increase the number of generations to get to 100%, for this purpose don't worry about replicates:

```
%% The whole monologue
clear; p = monkeyGa('hamletSoliloquy');
p.maxGen = 20000; % Increase the number of generations
tic; % Start the timer
output = monkeyGa('hamletSoliloquy',p);
gene2text(output.best(:,end)) % Show the found text
percentCorrect = (output.fitMax(end)/1446);
timeToComplete = toc; % End the timer
disp([num2str(100*percentCorrect) '% correct in ' ...
      num2str(timeToComplete) ' seconds'])
```

By using the `tic` and `toc` commands we can time how long a program takes to execute. How long did it take your algorithm to find the whole speech?

With 20000 generations and 100% accuracy, the algorithm took 235.1 seconds (3.9 min) to finish.

4.3.2 Brute force

How long would it take to find the same solution by a monkey on a type writer, i.e. by brute force? The average and worst case for a brute force algorithm can be easily calculated by counting the possible states. Let's be charitable and say this is a particularly clever monkey, who is systematic and never typing the same thing twice. Lets be even more charitable and say that this clever monkey also has a MATLAB license and has created a program to do the typing for him. How many possible states are there? How long will it take this MATLAB monkey to explore them all? Please show your work and use appropriate units for your answer.

(hint: to time a very fast piece of code, repeat in many times and take the average time, like this:)

```
%% Timing a single evaluation
aWholeBunchOfTimes = 100000;
test = randi([0 27], [aWholeBunchOfTimes, p.nGenes]);
tic; hamletSoliloquy(test); tEnd = toc;
oneEval = tEnd/aWholeBunchOfTimes;
disp(['One evaluation in ' num2str(oneEval) ' seconds'])
```

*Possible Number of states is 28^{1446} and time taken to compute one evaluation is $2.0268e-06$ seconds. Hence, it will take $28^{1446} * 2.0268e-06$ to reach the worst case.*

How comparable are these methods? Is random search comparable to evolutionary search?

- *Genetic Algorithms use randomness to create initial solutions, new individuals (via mutation and crossover operators) and even select individuals. However, selection tends to favor the most fittest individuals. Also, genetic operators are not purely random but use randomness to (often slightly) change individuals.*
- *Whereas random search methods take much longer time to reach the solution.*

5 Inserting MATLAB code into LATEX — 3 ways

1) This inline demo `for i=1:3, disp('cool'); end;` uses the `\mcode{}` command.¹

2) The following is a block using the `lstlisting` environment.

```
for i = 1:3
    if i ≥ 5 && a ≠ b           % literate programming replacement
        disp('cool');          % comment with some  $\TeX$  in it:  $\pi x^2$ 
    end
    [:,ind] = max(vec);
    x_last = x(1,end) - 1;
    v(end);
    really really long really really long really really long really ...
        really long really really long line % blaaaaaaaaa
    ylabel('Voltage ( $\mu$ V)');
end
```

Note: Here, the package was loaded with the `framed`, `numbered`, `autolinebreaks` and `useliterate` options. **Please see the top of `mcode.sty` for a detailed explanation of these options.**

3) Finally, you can also directly include an external m-file from somewhere on your hard drive (the very code you use in MATLAB, if you want) using the `\lstinputlisting{/SOME/PATH/FILENAME.M}` command. If you only want to include certain lines from that file (for instance to skip a header), you can use `\lstinputlisting[firstline=6, lastline=15]{/SOME/PATH/FILENAME.M}`.

¹Works also in footnotes: `for i=1:3, disp('cool'); end;`