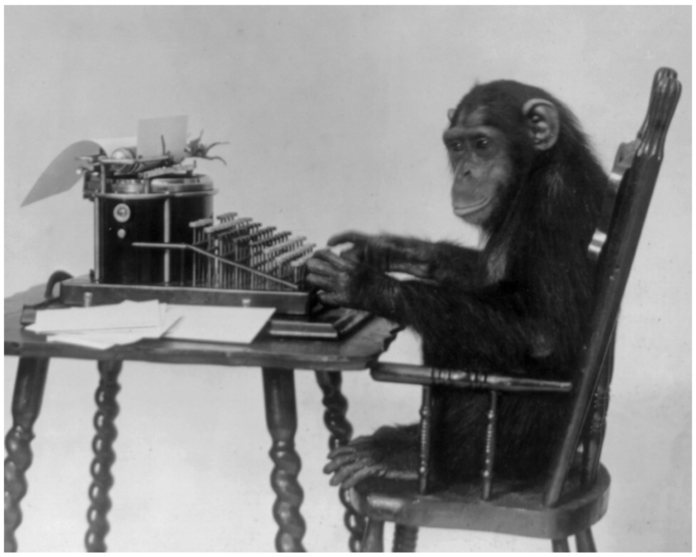
ECTA Homework 1
Genetic Algorithms and Infinite Monkeys

Arun Prabhu, arun.prabhu@smail.inf.h-brs.de
Dharmin Bakaraniya, dharmin.bakaraniya@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`:

```
    %% 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];
    pop       = newPop(1:p.popSize,:);  % Keep population size constant

    % Evaluate new population
    fitness   = feval(p.task, pop);
```

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','SouthEast');
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 LaTeX.
Don't overthink it! Each of these can be done in less than 10 lines!

1. **Tournament Selection**

```
%------------- BEGIN CODE --------------

%% This is 'random' selection of parent pairs, can you do better?
% parentIds = randi(p.popSize, [p.popSize 2]);
parentIds = [];
% iterate over all the individuals in the population
for i = 1:p.popSize
    % Randomly select p.sp individual
    possibleFather = randi(p.popSize, [1 p.sp]);
    [val, index] = max(fitness(possibleFather));
    % Select the one with best fitness as the father
    father = possibleFather(index);
    % Randomly select p.sp individual again
    possibleMother = randi(p.popSize, [1 p.sp]);
    [val, index] = max(fitness(possibleMother));
    % Select the one with best fitness as the mother
    mother = possibleMother(index);
    % Pack them together
    pair = [father mother];
    % Append them to the parentIds list
    parentIds = vertcat(parentIds, pair);
end
end
%------------- END OF CODE --------------
```

2. **Crossover**

```
%------------- BEGIN CODE --------------

%% No crossover happening, can you do better?
% children = pop( parentIds(:,1) ,:);
children = [];
sizeOfParent = size(parentIds);
for i = 1:sizeOfParent(1)
    % create random number to decide whether to crossover or not
    random_number = rand(1);
    if random_number < p.crossProb
        % randomly generate a crossover point where the gene ...
            will be spliced
        crossPoint = randi(p.nGenes);
        parentA = pop(parentIds(i,1),:);
        parentB = pop(parentIds(i,2),:);
        % merge the parents genes at crosspoint
        child = [parentA(1:crossPoint) parentB(1+crossPoint:end)];
    else
        % assign first parent's gene to child without change
        child = pop(parentIds(i,1),:);
    end
    % add child's gene to genepool
    children = vertcat(children, child);
end
%------------- END OF CODE -------------
```

3. **Mutation**

```
%------------- BEGIN CODE --------------

%% No mutation happening, can you do better?
children = children;
sizeOfChildren = size(children);
for i=1:sizeOfChildren(1)
    for j=1:p.nGenes
        % generate a random float(0 to 1) for every gene in a ...
            child
        random_number = rand(1);
        if random_number < p.mutProb
            % Randomly chosen values is assigned to the chosen ...
                gene from the,
            % interval of [0 27] which was used to create the ...
                initial
            % population.
            children(i,j) = randi([0 27]);
        end
    end
end
%------------- END OF CODE --------------
```

4. **Elitism**

```
%------------- BEGIN CODE --------------

%% Here we just keep the first individual as an elite, can you ...
    do better?
% eliteIds = 1;
% Since the population is very small, we are selecting best ...
    performing
% individual.
[val, eliteIds] = max(fitness);
%------------- 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', 'MyImplementation','p > ...
    0.05', 'p < 0.05','Location','SouthEast');
```

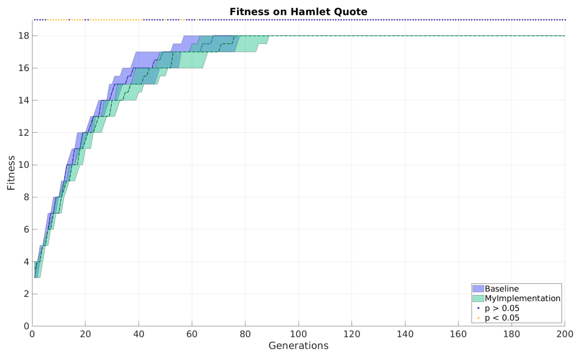


Figure 1: Baseline vs Our Implementation

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.

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.

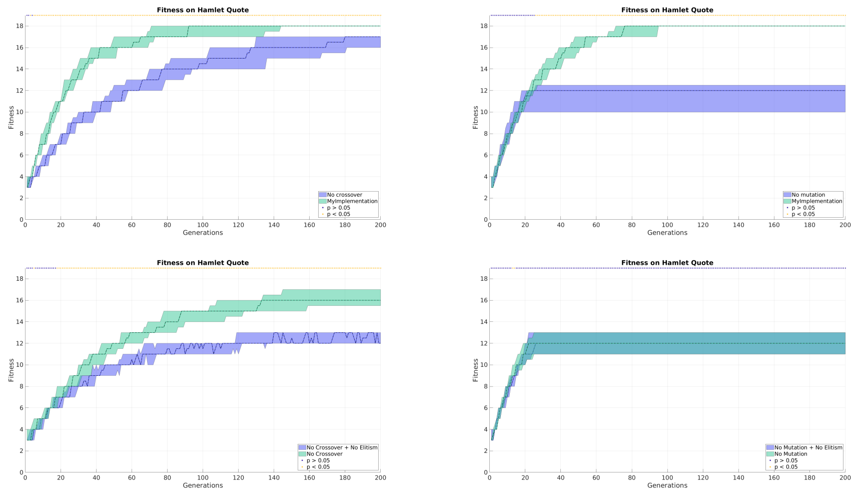


Figure 2: **GA performance when operators are removed**
Top Left: No Crossover, *Top Right:* No Mutation, *Bottom Left:* No Crossover vs. No Crossover and No Elitism, *Bottom Right:* No Mutation and No Elitism

4.2.2 Analyzing the Results

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

   (a) Crossover
      - The main purpose of crossover is to spread and maintain favorable genes across generation hence producing better solutions over time.
      - As it can be seen in the top left graph above, the fitness of the population achieved using crossover is better than the fitness of the population achieved without crossover. This fitness is also achieved in a shorter time.

   (b) Mutation
      - Mutation is a way of introducing new information into genes of the population. This enables achieving better solutions than just crossover.
      - As it can be seen in the top right graph above, the fitness of the population achieved using mutation is better than the fitness of the population achieved without mutation.

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 the solve the problem every time?**

   - The reason why "only crossover" typically does not solve the problem is random generation of intial individuals in the population.
   - As "only crossover" does not result in individuals which are outside the bounds set by the genes in the intial population.
   - Yes, an experiment which solves the problem with "only crossover" can be devised.
   - For example, in One-Max, if we create the initial population such that it has all the possible values for each gene then crossover alone can solve the problem, given enough generations and if the crossover point is not constant.

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

(a) Elitism with crossover

- Elitism helps to reach the best attainable fitness just a little faster. To ensure "no elitism", we added the eliteId at the end instead of at the beginning, since the last 1 individual is being excluded (refer monkeyGaReplica.m line:103).
- As seen from the graph on the bottom right, with only crossover vs. crossover and elitism, median fitness of the population was reached just 4 generation faster.

(b) Elitism with mutation

- Elitism helps to retain the progress made by the parent generation, so even if the children generation mutates towards worse fitness, the whole population does not suffer.
- As seen from the graph on the bottom left, with only mutation vs. mutation and elitism, we can see that elitism helps to achieve better fitness of the population.

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  = 10000;              % 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 ' (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?
`In 807.1651 seconds, the algorithm achieved 98.4094% correctness.`

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'])
```

- Taking 10 trials on single evaluation time measurement, we got an average single evaluation time of $8.76299 \cdot 10^{-7}$ seconds.
- Possibilities for a single character : 28
- Number of characters : 1446
- Number of distinct possibilities : $28^{1446} = 3.895 \cdot 10^{2092}$
- Possible exploration time : $3.895 \cdot 10^{2092} \cdot 8.76299 \cdot 10^{-7} = 3.4131 \cdot 10^{2086}$ seconds.
- Possible average time : $\frac{1}{2}\cdot$ total exploration time $= 1.7 \cdot 10^{2086}$ seconds.

How comparable are these methods? Is random search comparable to evolutionary search?
`No. These methods are not even close to comparable.`

- `GA took 807.1651 seconds to achieve correctness of 98.4094%.`
- `Brute force(smart monkey with MATLAB skills) would take` $3.4131 \cdot 10^{2086}$ `seconds (more than the age of universe` 10^{13}`).`
- `GA (1) - Smart Monkey (0)`

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 LaTeXin it:  πx²
    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 % blaaaaaaaa
    ylabel('Voltage (μV)');
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}`.

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