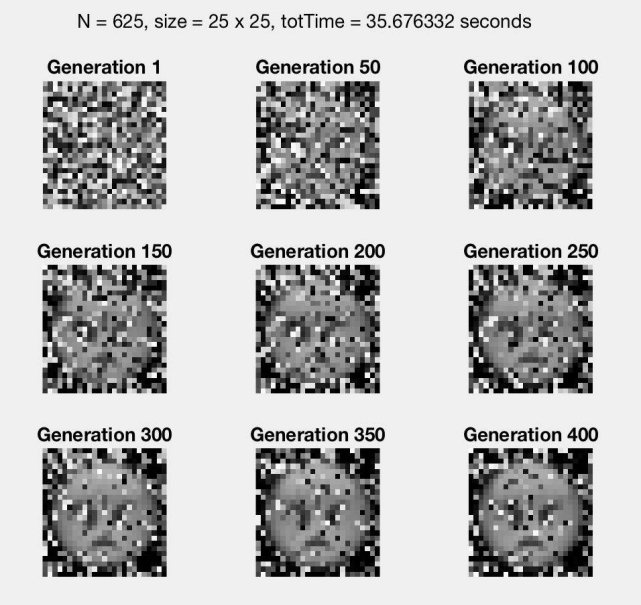
FINAL PROJECT

Due on: December 8^{th}, Sunday 6pm
This Final Project will be Interview Graded

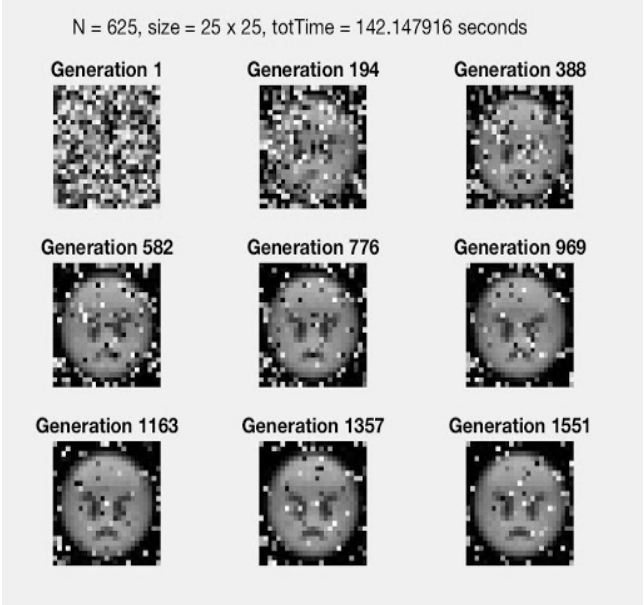
Genetic Algorithms - an Introduction to Artificial Intelligence

The concept of Artificial Intelligence (AI) has been intensely studied by many computer scientists since the early 1950s. However, due to the enormous computing power required to build and run AI algorithms, we have only started seeing serious applications of these technologies in the last decade or so.

Many of our designs for AI systems are inspired directly from nature, such as how neural networks, the widely publicized method used to create Google's ***Alpha Go***, roughly model neurons in human brains. In this project, we will be considering another model of machine learning called '***generative algorithms***'. The concept is derived directly from Charles Darwin's 'survival of the fittest' theory of evolution: we are going to 'breed' a phrase from completely random data over many generations by assessing the 'fitness' of each member of the population. An example of breeding the angry emoji over 400 generations (in black and white) can be seen in Figure 1(a). While the generated image may not be ***perfect***, it is clear to see that we can begin to discern the basic features of the angry emoji after roughly 300 generations. The same image is bred over more generations in Figure 1(b).



(a) An example of breeding the angry emoji over 400 generations



(b) The same image bred over

Figure 1: The generated image

After a certain point there are clearly diminishing returns in running the algorithm any further: The maximum and average fitness can be seen to stagnate in Figure 2.

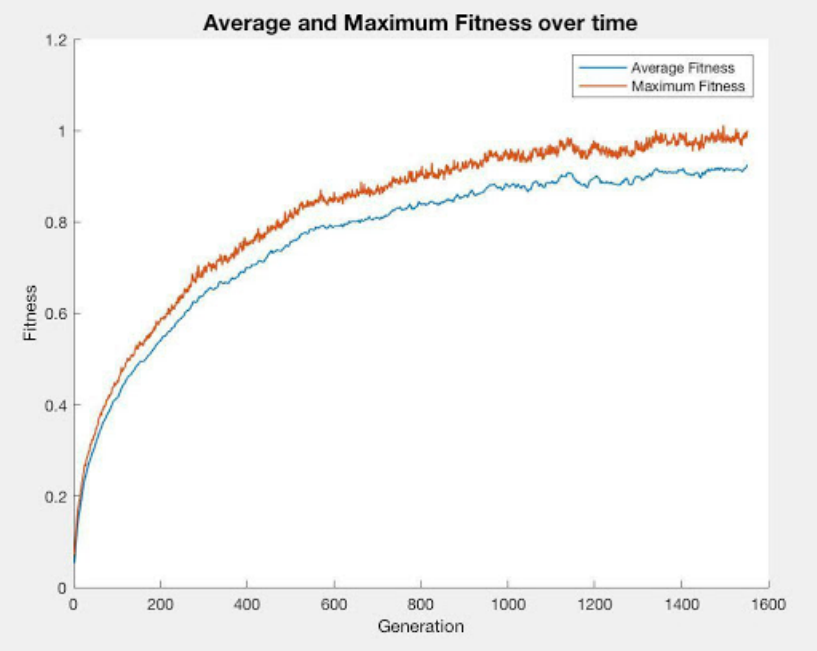


Figure 2: The maximum and average fitness in running the algorithm

Monkeys on a Typewriter

There is an age-old thought experiment pertaining to the nature of randomness and time called the 'Infinite Monkey Theorem'. It states that if you put a monkey on a typewriter and let it start hitting random keys, it will start typing random characters, and maybe occasionally a complete word or two... but you'd certainly have to wait a very long time to get anything ***meaningful*** typed from this process. However, if we gave the monkey an ***infinite*** amount of time, it could type out any given text we're looking for, such as a Shakespeare play.

In the real world, we do not have infinite amounts of time, so we'd instead have to think of creative ways to speed things up. We could start by adding more monkeys on more typewriters, giving rewards to the monkeys when they type complete words, or even limiting the amount of keys certain monkeys had access to.

Since computers essentially have the 'intelligence' of a million monkeys banging on typewriters, we need to consider these sorts of optimization strategies to get the computer to 'think' for us. This is an **open-ended project**, and while you will get many suggestions of how to do things along the way, you may discover your own tweaks to make your program even ***more*** efficient. You are encouraged to experiment at every step, but still stick to the core ideas presented here. More will be discussed on how to make modifications or implement your own ideas in Tasks 1.7.

The basic outline of the generative algorithm we are going to use is as follows:

1) Initialize population members with random DNA
2) Calculate the fitness of each member in population
3) Build a mating pool based on the fitness of each member
4) Select two parents from mating pool and breed child

5) Apply some random mutation to DNA of child

6) Repeat steps 4 and 5 until new population (new generation) has been bred

7) Repeat steps 2 through 6 for each new population until the goal is reached

First, we are going to try and get the computer to generate the phrase 'May your mountains rise'.

Task 1.1: Randomly generate initial population of strings

We are going to write a function ***buildPopulation*** that will initialize 200 random strings (all the same length as the target phrase) containing upper/lowercase letters and spaces. You should store these population members in a single cell, which your function should output. In this function, you should also define the **population size** as a variable, an important quantity that we will come back to often in this project.

To generate random characters, we simply have to generate integers which correspond to text characters in the ASCII table (more informaition: link), and use the ***char()*** typecast to turn them into characters.

Following the biology analogy, our **population** (string array) now contains **organisms** (strings) with **DNA** bits (individual characters) that were randomly generated. We are going to try and match that DNA to the target pnhrase's DNA.

Task 1.2: Calculate the fitness of each member in the population

In this example, we will simply define fitness as how many characters the current member of the population has correct when compared to our target phrase. For example, if our target phrase is 'Hello World', and we are assessing the fitness of the string 'qK lfdRoPLd', we would compare it as follows:

Target:	H	e	l	**l**	o		W	**o**	r	l	**d**
Test string:	q	k		**l**	f	d	R	**o**	P	L	**d**
DNA match:	0	0	0	**1**	0	0	0	**1**	0	0	**1**

The test string would have a fitness score of 3, due to having 3 DNA matches to the target phrase. A phrase with 100% fitness (i.e., the phrase 'Hello World') would have a fitness score of 11 in this example.

You can see that with a longer target string, the maximum possible fitness goes up. Furthermore, having a fitness of 3 in a phrase like 'Hello World' is much less significant to having a fitness of 3 in a shorter phrase like 'Hello'. Therefore, it is going to be more useful for us to look at the string's fitness as a **percentage**, where 0 has no matches, and 1 is completely matching (also known as **normalizing** the fitness score). In this case the test string has a fitness of approximately 0.273, or 27.3%.

We will now write a function ***calculateFitness*** which calculates the fitness of the entire population in the manner discussed above.

Task 1.3: Build a mating pool

Now that we know the fitness of each member in the population, we are going to build a 'mating pool'. This is essentially a crude way of selecting two parents from the population based on fitness, where fitter members are more likely to be picked.

You are going to write a function called ***buildMatingPool***. The basic idea is that the mating pool will be like a raffle that we can pick parents from. When setting up the mating pool raffle, we will simply give the fitter phrases more tickets to be chosen by.

We could achieve this by multiplying the fitness (which is a value from 0 to 1) by some sort of '**mating factor**' (say 10), and add that many tickets to the raffle. For example, if the population member has a fitness of 0.2, it will get $0.2 \times 10 = 2$ raffle tickets added. If the next member has more matches and has a fitness of 0.8, it will get 8 tickets, and thus will more likely be chosen when parents are selected.

One drawback to this method of building the 'raffle' is that it doesn't enter many tickets at all if the entire population's fitness is low (such as when you first start the algorithm with random text). To account for this, we will again normalize our population's fitness values so they go from 0 to 1, where 1 corresponds to the maximum fitness of the current generation. Now the fittest member will always be given 10 tickets into the raffle, even if it only has a fitness of < 0.001.

Your completed mating pool outputted by your function should only contain **indices** of the population members added to it, **not entire copies** of the string (this will take up a lot of unnecessary memory). It is also important to have your mating factor as a variable that can be easily adjusted later.

Task 1.4: Breed a child from two parents
Note: at this stage you may wish to begin on your source code outlined in Task 1.6 to begin testing some of these components.

We are going to write a function which will breed two of our organisms (strings) by combining their DNA (characters). To do this, we try two methods. In the first method, we will choose a random midpoint in the two phrases, and split the two parents' DNA up accordingly. For example, if we were breeding the phrases 'Heqqq' and 'R llo', and were lucky enough to randomly choose the midpoint 2, the child would look like:

Parent 1 DNA:	**H**	**e**	q	q	q
Parent 2 DNA:	R		**l**	**l**	**o**
DNA from 1 (midpoint = 2):	1	1	0	0	0
DNA from 2 (opposite of 1):	0	0	1	1	1
Child:	H	e	l	l	o

Our second method will be to take a completely random selection of one parent's DNA, followed by taking the remaining DNA from the other. For example, if we were breeding the phrases 'HrllL' and 'pe qo', the result may look like this:

Parent 1 DNA:	**H**	r	**l**	**l**	L
Parent 2 DNA:	p	**e**		q	**o**
DNA from 1 (randomly selected):	1	0	1	1	0
DNA from 2 (opposite of 1):	0	1	0	0	1
Child:	H	e	l	l	o

Again, this would be a very lucky case where the randomly selected DNA for each parents spelled out the word 'Hello'.

Write a function called ***breed*** which takes two parents as an input and returns a child as an output. The function should contain **both** methods of breeding a child, which we will use one at a time (by commenting out the method not being used).

Task 1.5: Cause DNA mutation

The key to any evolutionary process is the fact that DNA can randomly mutate, and sometimes those mutations lead to improvements in an organism that two parents' DNA alone could have never provided.

We will write a function called ***causeMutation*** which will randomly mutate the DNA of the child outputted by ***breed*** by randomly changing some characters. Start by defining a **mutation rate** which determines how often mutation should occur. If a character is selected to be mutated, simply change it to a new random value (A-z, or space). A typical mutation rate is around 4%.

Task 1.6: Bring it all together in a .cpp file!

We now have all of the components of a genetic algorithm. In a .cpp file, we can now have them work together to produce our target phrase, 'To be or not to be'.

In C++ do the following tasks:

1) Define target
2) Generate initial population
3) While the target is not found OR maximum generations has not been reached, do the following:
    a) Calculate fitness
        i) Print the maximum fitness member from the current generation, as well as generation number to the terminal.
    b) Build mating pool
    c) Breed a new population
        i) Randomly select two **unique** parents from mating pool
        ii) Breed child from parents
        iii) Mutate child
        iv) Replace old population members with children until entirely new generation has been bred
    d) Store (write) the values of the fitness scores in each generation into a **.csv** file. (CSV stands for comma separated values). Ideally the csv should be formed in such a way that it is easily readable by MATLAB.

In MATLAB do the following tasks:

1) Read the data that you stored from the .cpp file into a Matrix.
2) Find the average and maximum value of the fitness score of each generation.
3) Plot maximum and average fitness over generations
4) Plot 'genetic diversity' (maximum-average fitness) over generations

```
Population # 1608 Fitness 0.913043 May your mountaens riwe
Population # 1609 Fitness 0.913043 May your mountaens riwe
Population # 1610 Fitness 0.913043 May your mountaens riwe
Population # 1611 Fitness 0.913043 May your mountaens riwe
Population # 1612 Fitness 0.913043 May your mountaens riSe
Population # 1613 Fitness 0.913043 May your mountaens riZe
Population # 1614 Fitness 0.913043 May your mountaens riZe
Population # 1615 Fitness 0.913043 May your mountaens riwe
Population # 1616 Fitness 0.913043 May your mountaens riwe
Population # 1617 Fitness 0.913043 May your mountaens riwe
Population # 1618 Fitness 0.913043 May your mountaens riwe
Population # 1619 Fitness 0.913043 May your mountaens riwe
Population # 1620 Fitness 0.913043 May your mountaens riwe
Population # 1621 Fitness 0.956522 May your mountains riZe
Population # 1622 Fitness 0.956522 May your mountains riwe
Population # 1623 Fitness 0.956522 May your mountains riwe
Population # 1624 Fitness 0.956522 May your mountains riZe
Population # 1625 Fitness 0.956522 May your mountains riwe
Population # 1626 Fitness 0.956522 May your mountains riwe
Population # 1627 Fitness 0.956522 May your mountains riwe
Population # 1628 Fitness 0.956522 May your mountains riwe
Population # 1629 Fitness 0.956522 May your mountains riZe
Population # 1630 Fitness 0.956522 May your mountains riwe
Population # 1631 Fitness 0.956522 May your mountains riZe
Population # 1632 Fitness 0.956522 May your mountains riZe
Population # 1633 Fitness 0.956522 May your mountains riZe
Population # 1634 Fitness 0.956522 May your mountains riwe
Population # 1635 Fitness 0.956522 May your mountains riwe
Population # 1636 Fitness 0.956522 May your mountains riwe
Population # 1637 Fitness 0.956522 May your mountains riwe
Population # 1638 Fitness 0.956522 May your mountains riZe
Population # 1639 Fitness 0.956522 May your mountains riZe
Population # 1640 Fitness 0.956522 May your mountains riwe
Population # 1641 Fitness 0.956522 May your mountains riwe
Population # 1642 Fitness 0.956522 May your mountains riwe
Population # 1643 Fitness 0.956522 May your mountains riwe
Population # 1644 Fitness 0.956522 May your mountains riwe
Population # 1645 Fitness 0.956522 May your mountains riwe
Population # 1646 Fitness 0.956522 May your mountains riwe
Population # 1647 Fitness 0.956522 May your mountains riwe
Population # 1648 Fitness 0.956522 May your mountains riwe
Population # 1649 Fitness 0.956522 May your mountains riwe
Population # 1650 Fitness 0.956522 May your mountains riZe
Population # 1651 Fitness 1 May your mountains rise
rahulaedula95@DESKTOP-D9JJ361:/mnt/c/Users/rahul/1320/project$
```

Figure 3: Sample Generation, Maximum fitness score and the respective string.

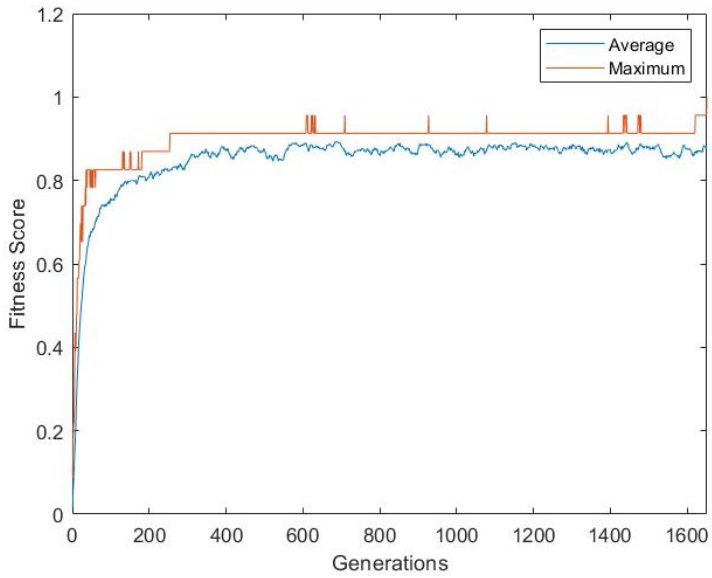


Figure 4: Sample maximum and average fitness in running the algorithm for Breed B

Task 1.7: Experiment!

You will now have the opportunity to test your creation by changing some of the parameters introduced in the making of this program. Note that all of these variables come together in quite a complicated manner to determine how well your evolution progresses. Therefore it is best to only vary **one variable at a time** when trying to discover its effect on overall performance and overall time taken. Try varying the following, taking note of what combination of parameters seem to work best, as well as any observations made along the way (which should probably go **in your report!**):

1) Change the length of the target phrase
2) Change the number of population members
3) Change the mutation rate
4) Change the range of possible characters being considered (include numbers, etc)
5) Try changing between the two breeding methods - which one works better?
6) Change the mating factor - what benefit might we get from increasing this? What's a reasonable value for it?
7) Change the maximum generations - what happens to fitness over time?
8) Which function takes the longest to run? Can you improve its runtime at all?

To make our code much more efficient at selecting the best parents, we can force our population to favor slightly fitter members much more than everyone else. After calculating our fitness, which are numbers from 0 to 1, we can raise it to some power, let's say 3. Now values which are ***close*** in fitness will be much easier to differentiate between, for example:

Original fitness:	0.33	0.30	0.27
fitness^3:	0.036	0.027	0.019

Although the fitness values themselves got smaller, the ***relative difference*** between them became larger. Remembering that the mating pool normalizes fitnesses to the maximum fitness, this will now award 0.33 ***even more*** tickets in the raffle compared to 0.30 or 0.27. In your source code, try raising your entire fitness vector to different values before the mating pool is built (we'll call this the '**exponential factor**'). How high can you make this value before it stops becoming beneficial? How might you want to adjust your mating factor (from Task 1.3) after introducing this exponential factor?

Final thoughts

You may have noticed from Task 1.7 that if you make values such as the population size large enough, your code will run arbitrarily well. This is because if we are trying to breed a 20 character phrase, it won't be too hard to find perfect matches on all of those characters within a random population of 2000+ phrases. In machine learning, this is referred to as 'overfitting' your model: it's performing better than it should because you've given it more resources than it deserves.

Instructions to submit your Assignment

The assignment should contain a fully functional Genetic Algorithm C++ source code along with the MATLAB plotting code. Submit your .cpp and .m files on Moodle under the Final Project tab by the due date along with a brief report of your experiments. You do not need to submit any executable files.

Keep in mind the Honor code and ensure that you do not violate any of the rules it entails.