MDL Assignment 3 Report

Genetic Algorithm - Summary

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are an exploitation of random search provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.

Initial Population

The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. In our assignment **we initialized our population with the overfit vector given in "overfit.txt" along with mutations of this vector to for a population of size 50.**

Fitness Function

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. In our assignment a variety of fitness functions were tried but we finally settled on **weighted addition (ax+by) of the errors** as our final fitness function. More detail on the fitness function will be presented later in the report.

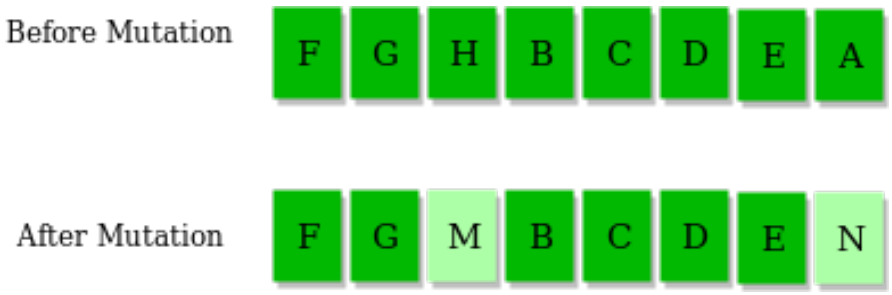
Retention

(Major Change to base genetic algorithm)
The basic genetic algorithm would now proceed with selection, crossover and then mutation. However our genetic algorithm has some major differences in these steps. The additional step of retention is present in our algorithm which **takes the top 10% of our current population, probabilistically mutates them very slightly and passes them onto the next generation's population** without any crossover. This helped us ensure that the **genetic algorithm did not take too long and get lost trying to find a minima**. We also had to ensure that we weren't converging too quickly and therefore, **we only took 10% of the population and also slightly mutated them to ensure diversity in our population.**

Mutation

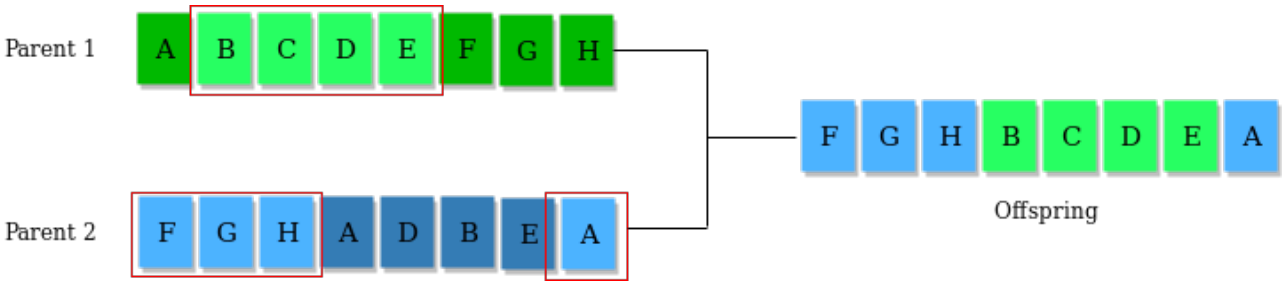
(Major Change to base genetic algorithm)
In general, mutation takes place after crossover. However, in our genetic algorithm, mutation takes place at different places, **once in the retention stage and once during the crossover.** The retention mutation was explained in the previous subtopic. Our

crossover has a **probabilistic chance of picking genes from either of the 2 parents, but it also has the probability of mutating that gene.** This "**Hybrid Probabilistic Crossover"** will be explained in more detail in the coming sections.



Crossover

Crossover represents mating between individuals. Two individuals are selected using from the **top 50% of the population(selection policy)** and crossover sites are chosen **probabilistically**. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring). The **unique element of our crossover is that it also offers a probabilistic chance for mutation** to occur.



Fitness Function

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The **probability that an individual will be selected for reproduction and mutation** is based on its fitness score.

Approach 1

The initial approach was to **only use Validation error as the fitness function**. The aim of the algorithm would be to reduce the validation error. The reason this approach was initially picked was because the given vector was overfit onto the training set and this fitness function focused on bringing validation down. However, we quickly began to realize the downfall of this approach, as it **started overfitting onto validation and train error soared**. This caused us to

```
def cal_fitness(self):
    global requestsleft
    fitness = 0
    err = get_errors('ayf35GN7
    self.valerror=err[1]
    fitness=err[1]
    requestsleft-=1
    return fitness
```

switch our approach to something involving both parameters.

Approach 2

```
def cal_fitness(self):
    global requestsleft
    fitness = 0
    err = get_errors('ayf35GN7wzsn
    assert len(err) == 2
    requestsleft-=1
    self.trainerror=err[0]
    self.valerror=err[1]
    fitness=err[0]*err[1]
    return fitness
```

The next approach involved us **multiplying both errors to obtain fitness.** This ensured that both errors made a part of the fitness function and avoided overfitting onto one data set. The downfall of this fitness function was observed in crossover. The fitness function played a key part in deciding which parent to pick the gene from in the crossover step. The probability of picking the gene was based upon the value of the fitness function and this **fitness function magnified even small differences in error to massive differences in fitness**(due to multiplication operator) causing the probabilistic model to always pick the gene from the parent with lower fitness which **gravely decreased our diversity** in the population. Therefore, this function was abandoned.

Approach 3 (Current Approach)

```
def cal_fitness(self):
    global requestsleft
    fitness = 0
    err = get_errors('ayf35GN7wzsnh
    assert len(err) == 2
    requestsleft-=1
    self.trainerror=err[0]
    self.valerror=err[1]
    fitness=(a*err[0])+(b*err[1])
    return fitness
```

This approach involves applying a **weighted addition to both errors** to form the fitness function. Initially, the fitness function was picked as a simple addition but we began to realize that this **sometimes skewed and caused overfitting onto a specific dataset**. Therefore, based on past history, we apply these weights to add priority to a certain error over the other if needed. **Both “a” and “b” follow a<=1 and b<=1**. This approach worked best for us as we could easily change the weights and the **algorithm would rebalance and ensure no overfitting was going to occur**. It also ensured a fair crossover which approach 2 wasn’t able to guarantee.

Crossover Function

Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. Crossover is sexual reproduction. Two strings are picked from the mating pool at random to crossover in order to produce superior offspring. Throughout the assignment we tried multiple types of crossover some of which we will list down below.

Selection Operator (Common to all approaches)

The selection operator is the preceding step to the crossover function and is extremely essential to the crossover. Our **selection operator simply picks two vectors randomly from the top 50%** of the population(sorted by fitness) and make them mate to produce 90% of the new population.

```
print("Hybrid Crossover involving mutation",file=f1)
print("",file=f1)
for _ in range(s):
    parent1 = random.choice(population[:int(0.5*POPULATION_SIZE)])
    parent2 = random.choice(population[:int(0.5*POPULATION_SIZE)])
    print("Parent 1: {}\t Fitness: {}".\
    format(parent1.chromosome,parent1.fitness),file=f1)
    print("Parent 2: {}\t Fitness: {}".\
    format(parent2.chromosome,parent2.fitness),file=f1)
    child = parent1.mate(parent2)
    print("Child: {}\t Fitness: {}".\
    format(child.chromosome,child.fitness),file=f1)
    new_generation.append(child)
```

Approach 1 (Hybrid Crossover)

Our initial approach was very similar to a regular crossover in which we take 2 vectors and **give equal probability of the child inheriting a gene from either parent.** However we also decided to make a hybrid between crossover and mutation by adding a **10% chance of actually using a mutation rather than a gene from either parent**. This small change yielded great dividends as we were able to maintain diversity and find new minimums rather than converging.

```
def mate(self, ind):
    child_chromosome = []
    for gp1, gp2 in zip(self.chromosome, ind.chromosome):
        prob = random.random()
        if prob < 0.45:
            child_chromosome.append(gp1)
        elif prob < 0.90:
            child_chromosome.append(gp2)
        else:
            child_chromosome.append(self.mutated_genes())

    return Individual(child_chromosome)
```

Approach 2 (Non-Uniform Probabilistic Discrete Crossover)

Inspired by two different crossovers from https://www.researchgate.net/publication/288749263_CROSSOVER_OPERATORS_IN_GENETIC_ALGORITHMS_A_REVIEW

A uniform discrete crossover uniformly picks a gene in a similar fashion to what was implemented in the first approach. In this approach we tried to use t**he fitness of the vectors to cause a shift in probability of picking a gene.** This ensured that more genes were picked from better chromosomes. The downfall of this technique is that **we were unable to maintain diversity in our population because the fitness function can sometimes skew the crossover,** causing the same vector to repeat multiple times. This was because we were dealing with such large fitness values and although 10^20 and

```
def mate(self, ind):

    child_chromosome = []
    for gp1, gp2 in zip(self.chromosome, ind.chromosome):

        total = self.fitness + ind.fitness
        prob = random.randint(1,total)

        if prob <= int(self.fitness - int(0.05 * total)):
            child_chromosome.append(gp1)
        elif prob <= int(total - int(0.1 * total)):
            child_chromosome.append(gp2)
        else:
            child_chromosome.append(self.mutated_genes())

    return Individual(child_chromosome)
```

10^21 errors weren't too different in our eyes, the computer gave a 10 times more chance to the vector with 10^20 which completely skewed our population. However we continued to take the ideas from this approach into our 3rd approach where we managed to balance the crossover.

Approach 3 (Uniform Probabilistic Discrete Crossover) (Current)

Trying to fix the problems of the last approach, In this approach we decided to fix a **uniform higher probability if the chromosome had a better fitness value and a uniform lower probability of getting picked if the chromosome has a lower fitness value.** This ensured a much more stable way of picking genes and ensured that even though we give preference to the better chromosome, it is not too biased as that we lose all the diversity in our population. We applied the same principle to our Mutation within the crossover, in which if the **chromosome has better fitness, its genes are the ones that are slightly mutated** and put into the child. T**his helped us ensure diversity.** This is the approach that worked better for us and it is our current approach.

```
def mate(self, ind):

    child_chromosome = []
    iter=0
    for gp1, gp2 in zip(self.chromosome, ind.chromosome):
        prob = random.random()
        if prob < 0.32:
            if self.fitness>ind.fitness:
                child_chromosome.append(gp1)
            else:
                child_chromosome.append(gp2)
        elif prob < 0.80:
            if self.fitness>ind.fitness:
                child_chromosome.append(gp2)
            else:
                child_chromosome.append(gp1)
        else:
            print("Mutation has occured during crossover",file=f1)
            if iter<2:
                child_chromosome.append(self.mutated_genes(4,iter))
            else:
                if self.fitness>ind.fitness:
                    child_chromosome.append(ind.mutated_genes(4,iter))
                else:
                    child_chromosome.append(self.mutated_genes(4,iter))
        iter+=1

    return Individual(child_chromosome)
```

Mutation Function

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low so as to ensure that search doesn't turn too random.

We researched mutations in great depth as we felt like mutations played a huge role in yielding us good results. In a broader sense we tried 2 different kinds of mutations, uniform mutation as well as probabilistic gaussian mutation.

Uniform Mutation

Approach 1

This form of mutation was used initially and is extremely primitive for obvious reasons since we were giving no regard to the current value.

```
if mode==1:
    gene = random.uniform(-10, 10)
    return gene
```

Approach 2

The second approach involved us taking into consideration the value of the **current gene and adding + or - a certain value to the current value**. We soon realized that this method was primitive too because we noticed that although this was effective with the first few genes which had relatively larger numbers numbers, the **higher index genes in the chromosome has minuscule numbers like (1e-7) and adding even 0.1 to such a number** would make too huge a difference to be effective.

```
if mode==8:
    gene = random.uniform(-1, 1)
    lol=self.chromosome[iter]+gene
    ans=0
    if(lol>-10 and lol<10):
        ans=lol
    elif(lol>0 and lol>10):
        ans=lol%10
    elif(lol<0 and lol<-10):
        ans=-(lol%10)
    return ans
```

Approach 3 (Infrequently used in current algorithm)

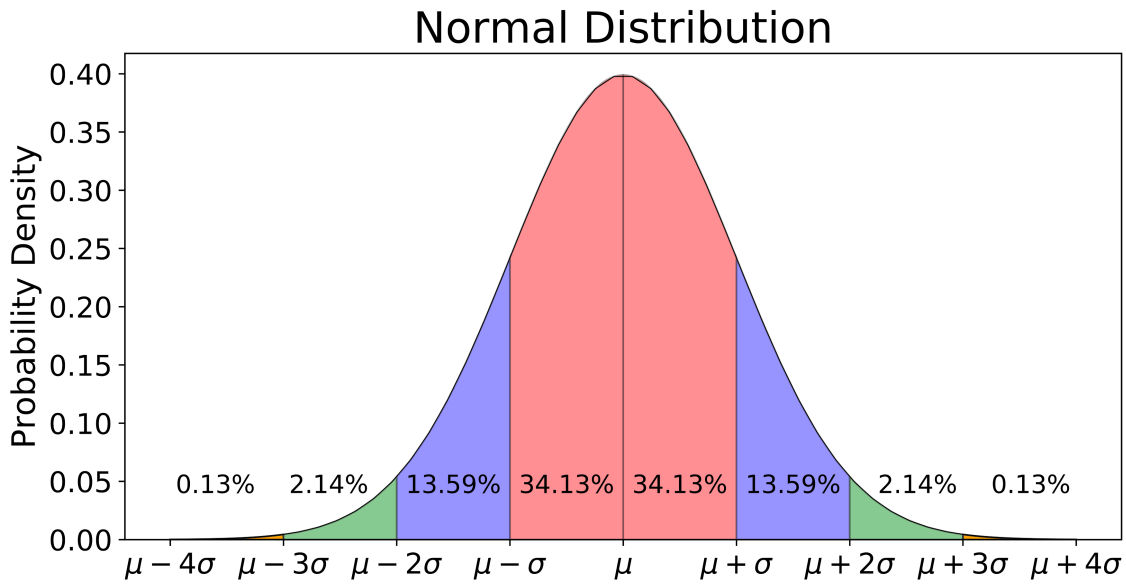
The third approach was much more effective since we **multiplied a number close to 1 with the currently existing number**. This helped ensure that the mutation wasn't too wild and we achieved decent results with this technique. T**wo major drawback with this technique was that positive numbers would remain positive and negative numbers would remain negative and a 0 would remain a 0.** Another drawback we noticed was that sometimes the change was too **big and we wanted something more closer to 1 more often**. However we still use this technique when we try to reach new minima's because it is slightly wilder than the following section.

```
if mode==9:
    gene = random.uniform(0.9, 1.1)
    lol=self.chromosome[iter]*gene
    ans=0
    if(lol>-10 and lol<10):
        ans=lol
    elif(lol>0 and lol>10):
        ans=lol%10
    elif(lol<0 and lol<-10):
        ans=-(lol%10)
    return ans
```

Probabilistic Gaussian Mutation

We realized that we wanted something that would usually pick a number close to 1 to multiply but occasionally pick a larger number to ensure diversity. The gaussian/normal

distribution was perfect in this regard and is probably the greatest catalyst in our algorithm for good results.



The distribution helped us to cause really good diversity in our dataset, yet also fine tune and converge onto minima's when required.

Approach 1 (Frequently Used in Current Algorithm)

```
if mode==5:
    mu,sigma=0,0.05
    gene = np.random.normal(mu, sigma, 1)
    lol=self.chromosome[iter]*(1+gene[0])
    ans=0
    if(lol>-10 and lol<10):
        ans=lol
    elif(lol>0 and lol>10):
        ans=lol%10
    elif(lol<0 and lol<-10):
        ans=-(lol%10)
    return ans
```

This approach is a direct continuation of the last approach of uniform mutation except for the key difference that we generate **the random number using a normal distribution** from the numpy package. The **distribution helped us to create really good diversity in our dataset, yet also fine tune and converge onto minima's when required.** Using normal mutations was probably the **greatest and most creative part of our algorithm** as we noticed the biggest impact in results using this. For **bigger mutations all we needed to do was increase sigma**(variance of normal distribution) and vice-versa for smaller mutations.

Approach 2 (Infrequently used in current algorithm)

Another very similar approach we used to the algorithm was using a **probabilistic way of picking sigma**(variance) to ensure even **richer variety in our population**. We used this approach occasionally when we needed more **random, stronger mutations**. This was used to **help us get out of local minima's and explore better local minima's and hopefully find the global minima**.

```
if mode==7:
    mu=0
    rngmut=random.randint(1,5)
    sigma=0.01
    if rngmut==1:
        sigma=0.1
    elif rngmut==2:
        sigma=0.01
    elif rngmut==3:
        sigma=0.001
    elif rngmut==4:
        sigma=0.5
    elif rngmut==5:
        sigma=1
    elif rngmut==6:
        sigma=1.5
```

Hyperparameters

- Population Size: The choice was made to choose a population size of 50 after initial populations of 100 were found to have too many wasted chromosomes, and later this number was brought down to 30 because we noticed that 30 vectors ensured it was large enough to maintain a rich diversity in the population and was small enough that we can process more generations with the smaller population size as we had to stick to the requests constraint.

- Retention Size: As our main algorithm ensured that the top n% of population was only slightly mutated and retained to the next generation, we choose n as 10. This was because 10% ensured that we kept the crux of the good vectors while also trying to maintain a good diversity as the rest of the population would be generated using our crossover.

- Mating Pool Size: The mating pool size was initially kept at a ratio of 50% of the population size. This seemed to work well for us since we observed that most vectors after the 50% point were not such great vectors. We stuck with this size all throughout our assignment.

- Values for mutation: The various values used in the mutation functions were obtained after trial and error on the same generation multiple times to see which mutations produced the best results. Upon obtaining values that seemed to work, these values were maintained. We used these various techniques whenever required. For example if we felt like we hit a local minima, we attempted to use a more aggressive mutation policy and when our aim was to fine-tune we stuck to smaller mutations.

- Fitness Function values: The values used in the fitness function ("a" and "b") were obtained through multiple adjustments, all the while observing the resulting generation. Upon observing good generations being generated, the values were kept constant until once again the skewed results were observed. Rebalancing of "a" and "b" were done whenever required.

Heuristics

All heuristics and various different approaches were explained throughout this document within each subsection. The ones which are in **current use in the algorithm have the tag (Current)** next to them and the rest of them are past approaches. Given below are few other **optimizations/kinks** we have applied throughout the algorithm

- We realized that the **first 3 indices** of the vectors were in general much **larger numbers than the other 8.** This helped us **customise the mutations** we applied to the algorithm and we applied more **aggressive larger mutations on the first 3 indices** and **much smaller fine-tuned mutations to the larger indices.**
- At the midway point we tried one more kind of mutation which was aggressively **multiplying the number with a random number from (0,2)** to try and displace our local minima. We **strongly believe this took us to much better local minima's** then the one we would have converged to if we did not use this technique.
- Initially we chose our parents at random, not as consecutive pairs but since we wanted every parent to get used in crossover we switched to the current model.
- Our retention policy allowed us to take **10% of the best population** to ensure we retain good vectors and **slightly mutate them to also ensure diversity in our population**.
- Our unique two step mutation meant that we always have a r**eally good balance of diversity and low error to ensure that we didn't just converge at any ordinary local minima**, but hopefully a very good minima.
- Our **probabilistic crossover** meant that we gave a **larger priority to the better vectors but not so much that we jeopardise the diversity of the population**, which would lead to overfitting and early converging.
- We tried different variations of the fitness function, mating pool size and mutation range as discussed before, each combination playing its own role depending on the current situation

Statistical Analysis

Train and Validation Error

Initial train and Validation error -

- Train Error - 79569
- Validation Error - 3625792
- Total Error - 3705361

Final Train error and Validation error we were able to achieve -

- Train Error - 195000

- Validation Error - 190000
- Total Error - 385000 (385k)

Vector that will achieve the best results

Having spent ample amount of time with the dataset and the errors, I strongly believe that **a vector having 300k training error and 300k validation error will achieve the best test error**. Although, I have submitted much better vectors with **less than 200k on both training and validation,** I believe that these have **slightly overfit** onto these datasets as the days have gone by. Even with the initial leaderboard, the one that did better on the leaderboard had a train and validation error of around **400k giving a test error of around 450k.** As I have continued to mutate and change my techniques, I reached a **test error with 187k but which I strongly believed overfit onto the test set.** My optimum vector was the one that **gave 300k on all 3 errors** and I feel strongly that it will **behave similarly on the new test dataset**. However, luck is always a factor in machine learning and it is equally possible that a garbage vector with **1 million train error performed better than any of the better vectors.** That is why towards the later end of this assignment, I made sure that I submit more vectors to ensure I have a higher chance of succeeding.

More statistical information

- **Number of generations/iterations to first minima:** To reach the first minima it took about 200 generations of the algorithm yielding a total error of 1.3 million (1.3*10^6) with a train error of 660k and validation of 640k
- **Number of iterations to second minima after crossover changes and mutation changes:** 400 Iterations to reach the second minima, at approximately total error of 1 million (1∗106) with 480k training and 520k validation error.
- **Number of iterations to reach third minima:** 150 Iterations to reach the third minima, at approximately total error of 3.85 ∗ 105 (385k) with 195k training and 190k validation error.

Diagrams

The diagrams are located in the **zip file as iteration1.html, iteration2.html and iteration3.html.** They are located in html files because we used data visualisation software to code them in javascript.

Trace

The trace is contained in trace.txt in the zip file which has 10 generations/iterations. In the initial population we began with a **fitness value of 1.65 million** (total error) and towards the end of our 10th generation **we achieve a vector with 541k fitness (total error)** These iterations display our algorithm in its prime with all the new optimizations discussed throughout this document.