Initial Enhancement : A Technique to Improve the Performance of Genetic Algorithms

Aadhar Gupta

Student, Dept. of Computer Science and Engineering, JSS Academy of Technical Education, NOIDA - 201301, Uttar Pradesh, India

---***---

Abstract - *In a G.A. (Genetic Algorithm) the "number of generations" needed to reach the optimal goal can be dramatically reduced (by a factor of 10 to 20) by just modifying the initial population. Rather than starting with an initial population of "randomly generated solutions", which have a very low fitness, we can get an enhanced initial population with fitter solutions if we follow some simple steps. This reduces the pressure on the procedure followed after the creation of the initial population (traditional procedure: selection-crossover-mutation), which otherwise takes many times more number of generations of solutions to reach the optimal solution. The possibility of premature convergence is also reduced.*

Key Words: Genetic Algorithms, Evolutionary Algorithms, Optimization methods.

1. INTRODUCTION

For the initial population of a G.A. we produce random solutions or members, blindly. We get some solutions that are fitter than the others but they get totally ignored while generating new solutions of the population. If we pay attention over these comparatively fitter solutions and use them as a benchmark for producing new solutions, then these new solutions inherit the good features of the comparatively fitter solutions of the population, with a possibility of improvement. Hence the initial population comes up with much fitter solutions than it otherwise does.

I demonstrate my technique with the help of a simple problem, the target optimal solution is a phrase – "peace and prosperity". All the possible solutions are hence treated as character arrays.

Firstly , let me clarify what I mean by " Traditional G.A." .

Traditional G.A. → "The standard procedure of selection, crossover and mutation. 'Selection'- roulette wheel, 'crossover' - randomly a position in the character array is chosen and all the preceding characters come from one parent while the succeeding characters come from other parent, ' mutation'- modification of the off springs produced with mutation rate of 0. 01 i. e. one character out of a hundred is replaced with a random character."

In a "traditional G.A." we start the search for optimal solution with a population of "randomly generated solutions" and these usually have very low fitness. Therefore when the procedure of selection, crossover and mutation starts we have these low fitness solutions to begin with and it "may" take a lot of generations to reach up to the final optimal solution i.e. the solution of maximum possible fitness.

Now imagine an initial population of solutions with considerably high fitness value. Doesn't this reduce the task of the algorithm? Yes indeed it does, as it gives the algorithm a better start because now it does not have to start with the solutions of minimal fitness.

The impact of such an initial population on G.A. performance is like, a huge part of our journey is covered before even starting, the impact is drastic .

Let's call this modification as "INITIAL ENHANCEMENT" and after adding this procedure to our traditional G.A. it would become " INITIALLY ENHANCED G.A.". The overall number of generations of solutions taken to solve the chosen problem by Traditional G.A. and Initially Enhanced G.A. :-

Traditional G.A. → average 774.35 generations approximately
Initially Enhanced G.A. → around 50 generations or less.

2. WORKING PROCEDURE

We have the target optimal solution phrase – "peace and prosperity". Let the total possible alleles (possible values for one position in the phrase i.e. one character) be 95 (the printable ASCII characters from ASCII value 32 to 126). The total search space being (95)^20 possible phrases (20 Characters and 95 values per character).

The java code for the implementation of my technique INITIAL ENHANCEMENT is provided in the section "3 Source Code " . It will enable the readers to properly understand, analyze and test this technique. In this procedure, instead of straight away starting with the traditional G.A. , we first spend some time and effort to find out fit solutions to start with. We proceed in following steps :-

1) First we create a population that I call " Mutover population " (mutover - mutated over : as we will use mutation over and over to create new solutions from the previously existing solution).

2) Steps to create a mutover population are :-

a) I have used Java programming language and each member/solution is an object. We begin by randomly generating solutions till we get the first member whose fitness is greater than all the solutions produced so far. Lets store this member object in a reference variable "fittest_member_uptil_now".

b) We use this fittest_member_uptil_now as a benchmark for the creation of new solutions/members from now onwards, by passing it as the argument in the constructor of new member objects. Hence we will need two member object constructors in total in our program, one that generates a member by assigning random values, and the other having the argument fittest_member_uptil_now whose functionality is explained ahead (c).

c) We create the new member by applying mutation on the fittest_member_uptil_now. I have used the mutation rate " 0.15 " i.e. 3 out of 20 characters of fittest_member_uptil_now would probably change to give us the new member . (I used 0.15 mutation rate in the constructor because it proved out to be the best among all the mutation rates I tried (Table 1)).

d) As the new solutions are created by modification in fittest_member_uptil_now, this way the newly generated solutions may still have the good qualities of fittest_member_uptil_now (correct character(s) in fittest_member_uptil_now) along with a possibility of new additional good qualities (correct characters that were not present in fittest_member_uptil_now) .

e) We keep analyzing the fitness of newly created members and as soon as we have a member who's fitness value surpasses that of fittest_member_uptil_now, we replace the old member object in fittest_member_uptil_now with this new member and now onwards this member becomes the benchmark, and is passed into constructor for creation of new members of the population.

f) This procedure is followed till we reach the maximum number of members permitted in the mutover population. I set the limit of a mutover population at a thousand members (just like the population limit of a single generation in the traditional G.A. procedure which is executed after the INITIAL ENHANCEMENT). Reason being, I could obtain fairly good fitness value members within this population size and it was easy to compare the performance of traditional G.A. and Initially Enhanced G.A. by means of total generations taken to reach optimal goal. But the total no. of members in a mutover population depends upon the programmer.

g) Now we use the same procedure (a) to f)) to create multiple mutover populations one after the other, and we keep storing the fittest member of each mutover generation in a separate array, let be "fittest_member_of_each_mutover_generation". The two things that determine the performance of this technique are the 'number of mutover populations' and the 'rate of mutation' during generation of new solutions of mutover population using fittest_member_uptil_now in the constructor. One should wisely choose the value of these two elements. In a separate program I varied the mutation rate in the constructor with argument fittest_member_uptil_now, and analysed the effect on the maximum fitness achieved in a single mutover population(Table 1) .

h) We can use any number of mutover populations depending upon how fast are we able to reach the optimal goal i.e. what number of mutover populations yields the least number of overall generations in the entire procedure. (Overall generations → mutover generations + generations produced during traditional G.A. procedure to reach the optimal/final goal) . We can use hit and trial method to determine this. How the number of overall generations taken by the Initially Enhanced G.A. to reach the optimal solution, varies with the number of mutover generations taken, is shown in Table 2 .

3) I have taken 20 mutover populations so it gives me 20 different solutions from 20 different populations, in array "fittest_member_of_each_mutover_generation". Fitness and diversity have been achieved simultaneously. Hence the element required to beat premature convergence , fitness and diversity simultaneously, is gained here.

4) Now we apply the traditional 'selection, crossover, and mutation of offspring' techniques to these 20 solutions stored in the array fittest_member_of_each_mutover_generation to create a whole new population of a thousand solutions. I have included both the off springs produced during a crossover into the population so we have 500 crossovers giving us 1000 new members. The corresponding methods in the source code are "selection_initial( )" and "do_crossover_initial( )".

5) Now we enter this new population of 1000 solutions as the initial population to the traditional GA.

6) This ends our “Initial Enhancement” phase and the other phase, traditional procedure of G.A. starts here, i.e. selection of the fittest members of the population, then crossover, then mutation of the offsprings. This continues over generations till we reach the optimal solution.

3. SOURCE CODE

```
/** The code for special constructor for creation of mutover population members : **/

/* Genotype is the class whose objects are the member solutions , genes[ 20] is a character array which carries the solution phrase in the object, fitness( ) is the method that calculates the fitness of the solution phrase { Calculated as ‘2 ^ x ’ , where x = ‘the number of correct characters in solution phrase ’ }. */

Genotype ( Genotype fittest_member_uptil_now )

{

        for(int i=0;i< genes.length ;i++)

{

if(Math.random( )<0.15)

        genes[i]=(char)( (int)( 32 + ( Math.random( ) ) * 95));

 else

        this.genes[i]=fittest_member_uptil_now.genes[i];

}

fitness( );

}

/** The code for methods involved in INITIAL ENHANCEMENT are :- **/

/* population[1000] is an array which stores the reference id of the member solution objects, fittest_member_of_each_mutover_generation[ 20 ] is an array to store the fittest member of each mutover generation/population ; I have used 20 mutover generations so size of array is 20. */

/* Method 1 */

void setup_mutover_populations( )

{

Genotype fittest_member_uptil_now;

float max_fitness_uptil_now;

for ( int gen=0 ; gen < fittest_member_of_each_mutover_generation.length ;gen++)

{

max_fitness_uptil_now=1;

/* as correct characters = 0 initially and 2^0 =1 so minimum fitness possible is 1. */

fittest_member_uptil_now=null;

for(int j=0; j< population.length ;j++ )
```

```
{
if(max_fitness_uptil_now == 1)
        population[j]=new Genotype( );
else
        population[j]=new
Genotype(fittest_member_uptil_now);
if(population[j].fitness > max_fitness_uptil_now )
{
  max_fitness_uptil_now=population[j].fitness;
fittest_member_uptil_now=population[j];
}       }
fittest_member_of_each_mutover_generation[gen] =fittest_member_uptil_now;
System.out.println("mutover generation number -" +gen+"\n fittest_member_current_generation : fitness= " + fittest_member_of_each_mutover_generation[gen].fitness + " Genes= " + String.valueOf ( fittest_member_uptil_now.genes));
System.out.println( );     }
selection_initial( );        }

/* Method 2 */
void selection_initial( )
{
ArrayList<Genotype> mating_pool_initial = new ArrayList<Genotype>( );
int i,k;
float total_fitness=0;
for (i = 0; i < fittest_member_of_each_mutover_generation.length; i++)
total_fitness += fittest_member_of_each_mutover_generation[i]. fitness;
float rand;
float fitness_sum;
for(k=0; k< population.length ; k++)
{
fitness_sum=0;
rand=(float)Math.random( )*total_fitness;
for (i = 0; i < fittest_member_of_each_mutover_generation.length; i++)
{
```

```
fitness_sum += fittest_member_of_each_mutover_generation[i]. fitness;
if(fitness_sum > rand)
{
mating_pool_initial.add ( fittest_member_of_each_mutover_generation[i]);
break;
 }
}
}
do_crossover_initial ( mating_pool_initial ) ;
}

/* Method 3 */
void do_crossover_initial ( ArrayList<Genotype> mating_pool_initial )
{
for (int i=0; i < (population.length)/2; i++)
{
int a = ( int ) ( Math.random( ) * mating_pool_initial.size ( ) );
int b = ( int ) ( Math.random ( ) * mating_pool_initial.size( ));
Genotype parent1 = mating_pool_initial.get(a);
Genotype parent2 = mating_pool_initial.get(b);
if(String.valueOf(parent1.genes).equals( String.valueOf(parent2.genes)))
i--;
else
{
Genotype[] child_array=new Genotype[2];
child_array = parent1. crossover_initial(parent2);
child_array[0].mutate( );
child_array[1].mutate( );
population[i]=child_array[0];
population[( population.length ) /2+i] =child_array[1];
}
}
}
```

4. RESULTS

Table -1: Maximum Fitness Achieved in a Single Mutover Generation for Different Mutation Rates

Mutation Rate / Trial	0. 8	0. 6	0. 4	0. 3	0. 2	0. 15	0. 1	0. 09	0. 07
1	16	64	256	256	128	256	512	256	512
2	16	32	128	128	2048	256	4096	128	512
3	16	32	256	256	1024	1024	1024	1024	256
4	8	32	128	1024	2048	128	512	512	512
5	16	32	512	1024	256	256	256	1024	128
6	16	64	64	512	512	512	2048	1024	128
7	16	32	128	2048	1024	4096	2048	4096	2048
8	16	32	256	128	512	512	512	256	64
9	32	32	128	256	512	1024	1024	512	2048
10	16	64	64	256	256	256	512	512	1024
11	16	64	128	128	512	128	512	1024	128
12	16	128	128	512	512	512	1024	512	512
13	16	32	512	512	128	2048	256	512	32
14	32	32	128	1024	512	256	512	64	256
15	16	64	256	256	512	1024	256	128	512
16	16	64	64	128	512	1024	1024	512	128
17	16	64	256	256	1024	1024	128	512	512
18	16	64	64	256	512	1024	2048	1024	1024
19	16	32	256	128	512	4096	1024	4096	512
20	16	32	128	128	512	512	256	512	512
Maximum Fitness (Average)	17.2	49. 6	192	460. 8	678. 4	998. 4	979.2	912	568

Table -2: Number of Overall Generations taken to reach the Optimal Solution for Different Number of Mutover Generations taken.

Mutover Generations / Trial no.	50	40	30	20	10	8	5
2	56	49	37	34	28	76	19
3	53	45	40	29	60	120	43
4	60	46	37	32	16	37	44
5	54	45	38	32	20	29	52
6	56	49	40	25	16	94	607
7	55	49	37	27	27	99	161
8	58	50	36	41	37	25	26
9	57	45	42	36	21	38	28
10	55	45	41	27	38	16	172

11	60	49	42	30	30	67	77
12	58	47	36	30	22	65	19
13	55	46	36	31	21	26	55
14	63	49	36	30	57	165	73
15	61	49	42	34	36	24	61
16	55	46	36	35	27	113	101
17	61	49	38	50	31	25	199
18	57	46	47	30	45	88	17
19	56	47	38	26	32	33	507
20	55	46	41	36	26	28	97
Overall Generations taken (Average)	57	47.25	39.3	32.5	30.5	60.9	118.6

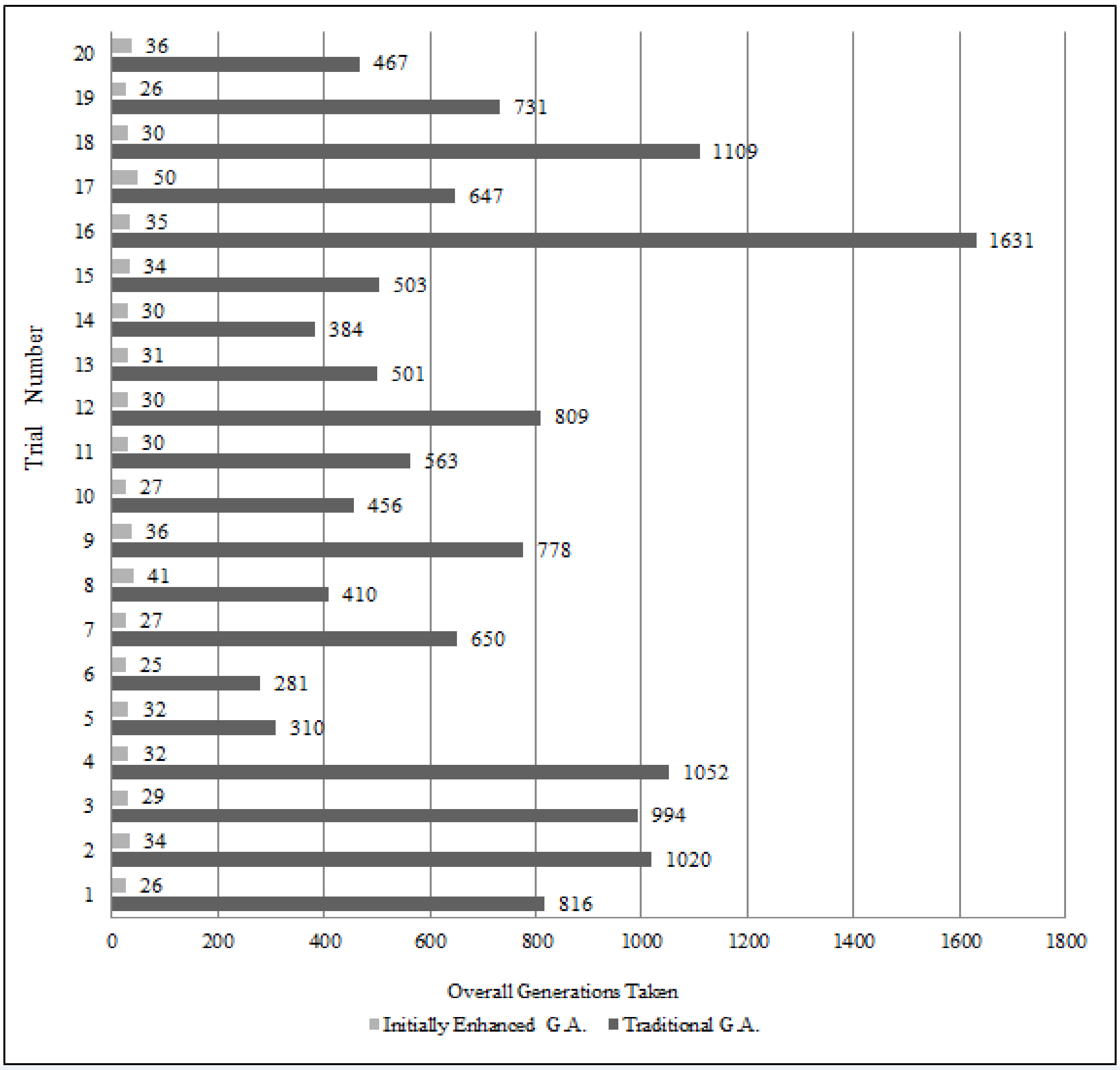

Chart -1: Total overall number of generations taken to reach the optimal solution : Traditional G.A. Vs Initially Enhanced G.A. (Mutover generations = 20, mutover population mutation rate = 0.15)

5. CONCLUSIONS

I have provided a method which improves the beginning of a traditional G.A. The interesting thing is that other methods which improve the performance of G.A. after it has commenced (like better selection techniques, better crossover methods etc.) can still be applied to get even better results.

The only thing I would like the readers to keep in mind while using this technique, Initial Enhancement, is that be very careful in choosing :-

a) the number of mutover populations
b) the rate of mutation of "fittest_member_uptil_now" for generating new members of mutover population.

I have used hit and trial method to demonstrate the behavior of my technique for different values of a) and b) (mentioned above) . As we observe in Table 1, the maximum fitness achieved appears to be highest around Mutation Rate = '0.1 to 0.15' . It increases as we go from 0.8 to 0.1 and then falls sharply. Similarly as shown in Table 2, the overall generations taken to reach final goal , are least (optimal) when number of mutover populations is in the interval '10 to 20'.Values greater or lesser than this range appear to result in greater number of overall generations taken to reach the goal. A wrong selection of value for a) or b) may render this technique ineffective.

REFERENCES

1. D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning," Addison-Wesley Professional, January 1989.

2. J. H. Holland, "Adaptation in Natural and Artificial Systems," Ann Arbor:The University of Michigan Press, 1975.

3. R.L. Haupt, S.E. Haupt, "Practical Genetic Algorithms," 2nd ed., John Wiley & Sons, New Jersey, 2004.

4. M. Melanie, "An Introduction to Genetic Algorithms," MIT Press, Massachusetts, 1999.

5. S.N. Sivanandam, S.N. Deepa, "Introduction to Genetic Algorithms," SpringerVerlag, Berlin, Heidelberg, 2008.

6. J. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection," MIT press, 1992.

7. R. Belew and M. Mitchell, "Adaptive Individuals in Evolving Populations, Santa Fe Institute Studies in the Sciences of Complexity Volume XXVI, Addison Wesley, 1996.

8. M. Mitchell, J. Holland, S. Forrest, "When Will a Genetic Algorithm Outperform Hill Climbing?" in Advances in Neural Information Processing Systems 6, 1993.

9. J. Koza , "Genetic Programming II : Automatic Discovery of Reusable Programs" ,MIT press, 1994.

10. J. Baldwin, "A new factor in evolution", American Naturalist 30: 441- 451, 1896.

11. L. Davis, "Handbook of Genetic Algorithms," Van Nostrand Reinhold ,1991.