Evolutionary Computation for Single and Multi-Objective Optimization

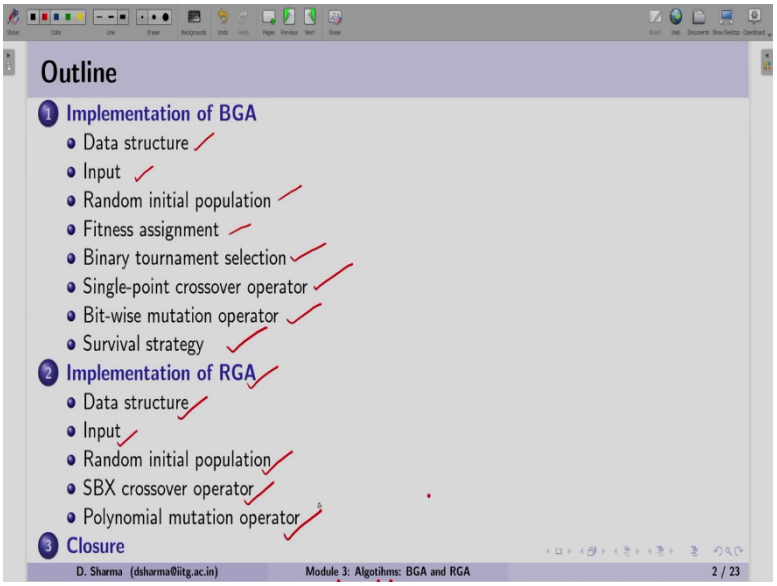
Dr. Deepak Sharma
Department of Mechanical Engineering
Indian Institute of Technology, Guwahati

Module – 03
Lecture – 07
Algorithmic Implementation of BGA and RGA

Welcome to the session on Algorithmic Implementation of Binary coded GA and Real coded GA. So, as of now, we have covered both of these genetic algorithm. So, in the case one where we discussed about BGA or binary coded GA; we used this algorithm for solving a real parameter optimization problem. Similarly, we have this real coded GA; so the operators wherever there is a need, we change those operators and we solve the real parameter optimization problem.

So, as of now we have understood various kinds of operators; we have gone through the hand calculations as well as simulation. So, we know how these algorithm work. So, in this session, we will be targeting that, the if we want to implement those algorithm; then what could be the way. So, I will be discussing one of the implementations in this session. So, this session includes, here first we will start with the binary coded GA.

(Refer Slide Time: 01:47)



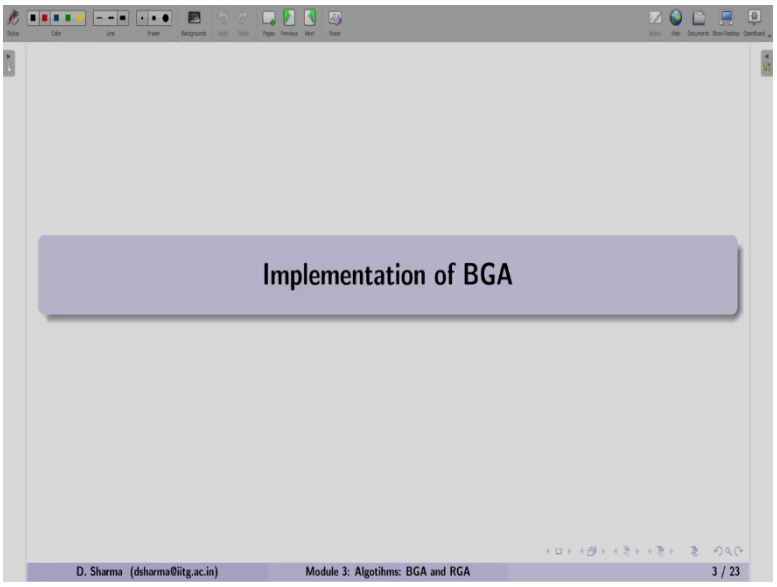
In this case we will first move to the data structure, then the set of input required for a BGA, random initial population, fitness assignment, the binary tournament selection. So, here we

have what we target here, the operators which we have already gone through; we did some hand calculations, so those operators we will discuss. Apart from that, there are various other operators that can also be implemented.

So, we will restrict our discussion to only those operators, which we have done some hand calculations. Thereafter, we will move to the implementation of single point crossover operator, bitwise mutation operator and the survival strategy. Once we are done, then we will be talking about the implementation of RGA.

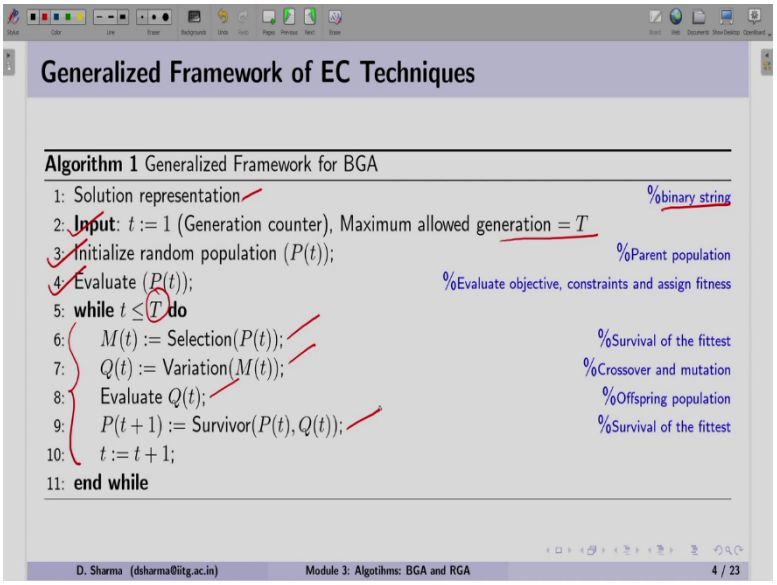
So, again if there is any change in the data structure that we will see; the input to RGA, random initial population and we know that we have to change the crossover operator and mutation operator. So, those operators we will discuss and finally, we will conclude this session.

(Refer Slide Time: 03:07)



So, let us begin with the implementation of binary coded GA, in short we are referring it as BGA.

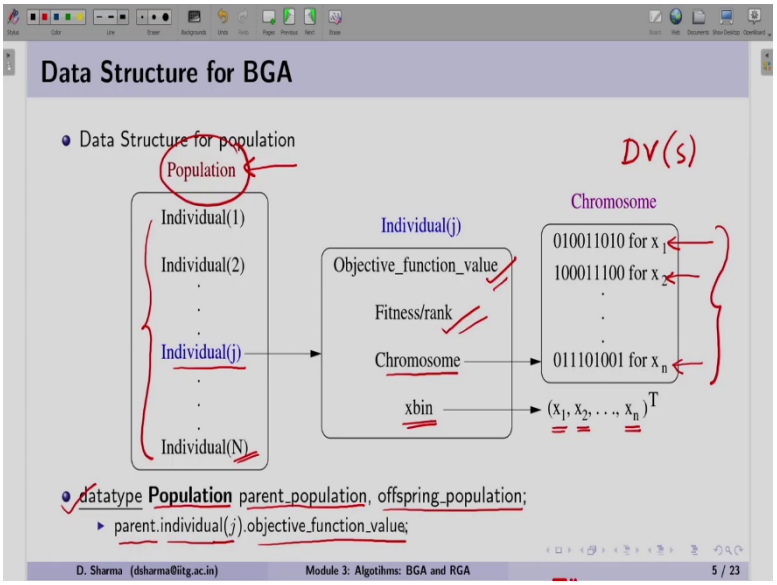
(Refer Slide Time: 03:20)



So, this is the generalized framework we are following and the working principle of both BGA as well as RGA are discussed using this framework. So, in this framework, as we can see in a step 1, we have the solution representation. And once we decided that this is going to be binary for BGA; then we have set of input. So, in this case, for simplicity we generally represent two input; but in the simulation you might have found that, there are certain inputs which are required to run our BGA.

Thereafter we regenerate the initial population, we evaluate the population and we have this particular while loop, which terminates based on the number of generation. Then we have selection, variation; then again we evaluate the offspring population and the survivor stage. So, this particular algorithm 1 is actually telling us that how should be our main function. So, if our main algorithm is this, then rest of the main operators we will be calling inside this main algorithm.

(Refer Slide Time: 04:40)



So, let us start with the data structure here. Now, data structure is important, because we have to store different values as well as, as and when it is needed, we have to extract those values. So, let us see, how the how the data structure of BGA can be made? So, here this is one of the data structures, which we are discussing. So, as you can see in the figure. So, first data structure will be based on the population.

This data structure includes all the individual as you can see. So, the maximum number of individual, we can have N. Now, as of now we are referring these individuals as sometimes solutions and sometimes members. So, inside a population for real coded or a binary coded GA, we may refer as individual or sometimes solution or sometime members. So, all of them will remain, will become the same.

Now, let us take a typical case of an individual j, because all individual will remain the same. So, this individual should save the value of the objective function, which we generally calculate and if there is any specific fitness or rank assignment, that also we should save it. So, as of now, we have not gone through the particular fitness or rank assignment; but we can have this particular option in our data structure. Then we should have a chromosome.

Now, as we know, chromosome stores all the binary string of the variables. Now, as of now, we are discussing only the BGA. So, we are assuming that all the real numbers are represented using binary string. So, here you can see under the chromosome; for x 1 we have a one binary string, similarly we have x 2 we have another binary string, similarly for this variable I have another binary string.

So, these binary string can have the same length or they can have a different length; meaning I can choose 5 bit string for x 1 and I can choose 8 bit string for x 2. So, similarly they can have same binary string length or they can have a different binary string length.

So, if we put together all of them, then we get a chromosome. So, this chromosome we have to store for every individual. As we know, this particular string will give me the decoded value of the string and you remember we write this as decoded value of the string D V s.

Now, using this scaling formula, we can find what is the real number say x 1, x 2 and x n. So, everything will be stored in xbin; so this means that the binary string is decoded and using this scaling formula, we are storing the real numbers. So, let us take one data type as you can see here.

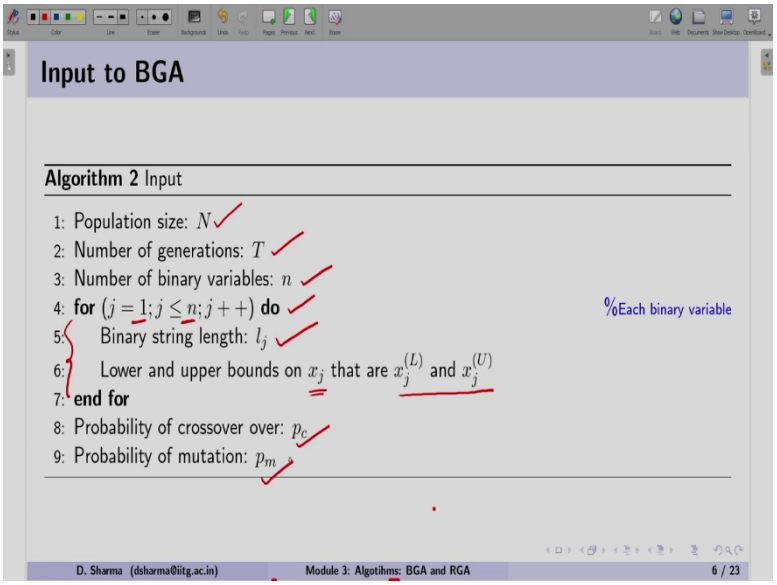
So, the data type we have taken as a population. When we are saying population, you can see from the figure; this particular population we are considering, which have N number of individual and each individual will be having objective function, fitness value chromosome and xbin vector.

Now, this population we can use it as. So, I can assign, we have a parent population, we have a offspring population. So, in binary coded GA, we have only two types of population as of now parent and offspring. So, as soon as we are assigning this population, so this data structure is one of the data type.

How we can use it? So, this is just as representation. So, I can say parent dot j. So, parent dot individual j dot objective function value; this means that, if I am going to call, then the value I can extract as well as if I call this I want to save some value, I can use this data structure.

Now, this data structure is important; because in a one go, we can extract the value as well as copy the value into the individuals, particular into the particular component of the individual.

(Refer Slide Time: 09:23)



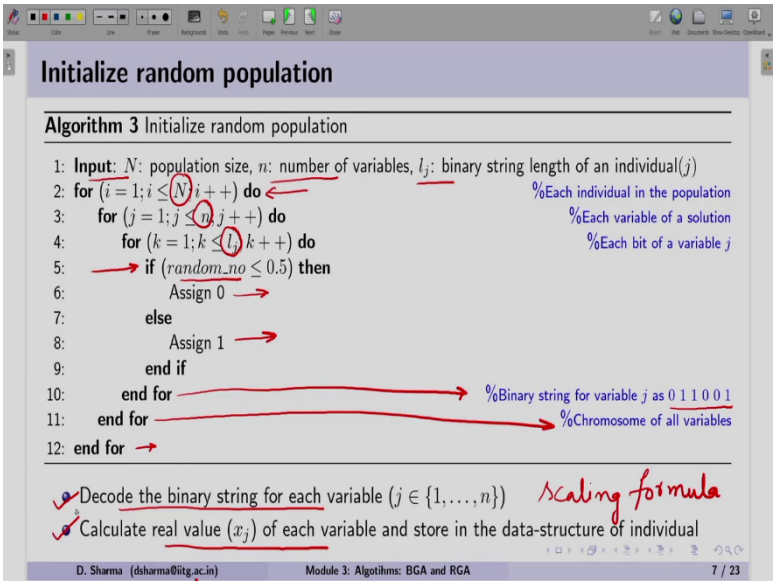
With this explanation on the data structure, we know that first step is the input. So, what could be the input to the BGA? So, I am representing here the important inputs that are required for BGA. Sometimes those input can change; because if any of the, in any of the operator needs some extra input, that we can include it. So, you can see these are some basic or important input that should be given to the algorithm.

So, look at the algorithm number 2. So, the first input here is the population size. And once we decide, then we have to tell what is the number of generation and similarly the number of binary variable. As I told you that, since we are working on binary GA; we are assuming that we have all variables which are binary in nature. Now, as soon as we get to know how many variables; then we have this for loop, which we are running from the first binary variable to the last binary variable.

So, first of all we should save what is the binary string length. So, as I told you that, the binary string length for x 1 variable and x 2 variable can be same or different. So, individually we will be giving the binary string length and for the same variable, we will be giving what is the upper and the lower limit of the variable.

So, inside this for loop, we will be getting the binary string length of the variable j as well as the lower and upper bound of the variable. So, once it is done, then we need probability of crossover and probability of mutation. So, these set of inputs are important for running any binary coded GA.

(Refer Slide Time: 11:19)



Once we have given the input, now our binary coded GA is ready to work. So, in this case we know that, we have to generate the population randomly. So, in algorithm 3, we can see we need certain inputs here. So, population size, number of variable, the binary string length of every individual j. So, here I am showing you the representation, the algorithmic representation here. So, the first for loop says that, we will be running for all population size, all the or for all population members or individuals.

So, once we have taken say individual i, then we have to run for every variable in this particular individual. Once we decided it, then there is another for loop that will be working for the number of by a number of bits in a binary string. In this case as you can see at this step number 5, we are generating a random number; if it is smaller than 1, we will assign 0, otherwise we will assign 1.

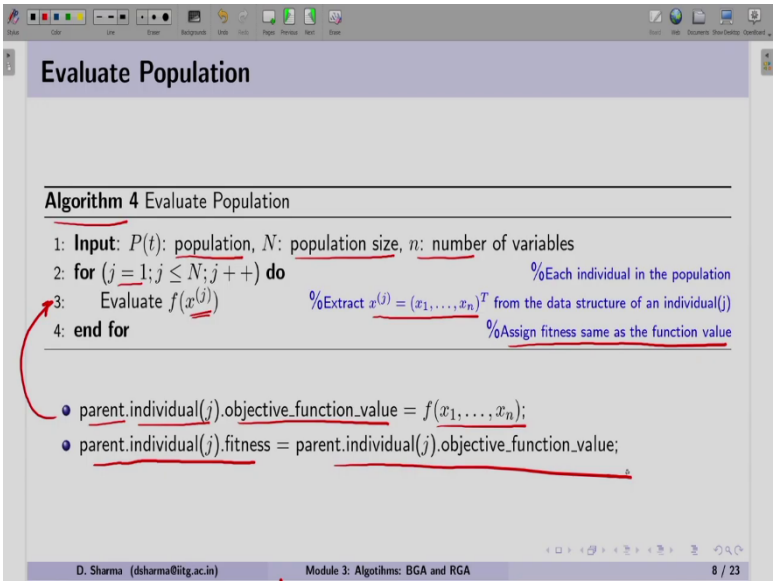
So, in this case what you will realize that, at once we are finishing the for loop; we will get a binary string for a variable j like this. So, in this case, we are randomly generating a set of 0 1 0 1 1 0 for a variable j. And afterwards once it is done, then we will be performing the same thing. So, you can see this for loop which is ending give will give me the chromosome. So, as we understood, chromosome includes all the binary string of the variables, so binary variables. And thereafter we finish it.

So, this particular initialization will give me the chromosome for each individual in the population. Once this is done, we have to decode the binary string. So, since decoding of a binary string is simple. So, here we are assuming that, we can do it by writing a simple code

for decoding the binary string. Once we decode, then we have to calculate the value of the variable. So, as you know that, we can use any scaling formula for using this decoded value to get the value of x j.

So, this is this we will be doing for every variable, for every individual. So, that is why these two points are mentioned here and we can write a very simple code including, which include decoding of the binary string as well as using the scaling formula, we can find what is the real number. So, here that is why we have written in a words, which we can write in a code.

(Refer Slide Time: 14:26)



Afterwards when we get the values of x 1, x 2, x 3 for an individual j; now the next task is we have to evaluate the population. Now, looking at the algorithm number 4, the input is the population and we should know what is the size and the number of variable. In this case if you see this for loop in step number 2, that runs from j equals to; j equals to 1 to j equals to N; this means that, for every member we are running.

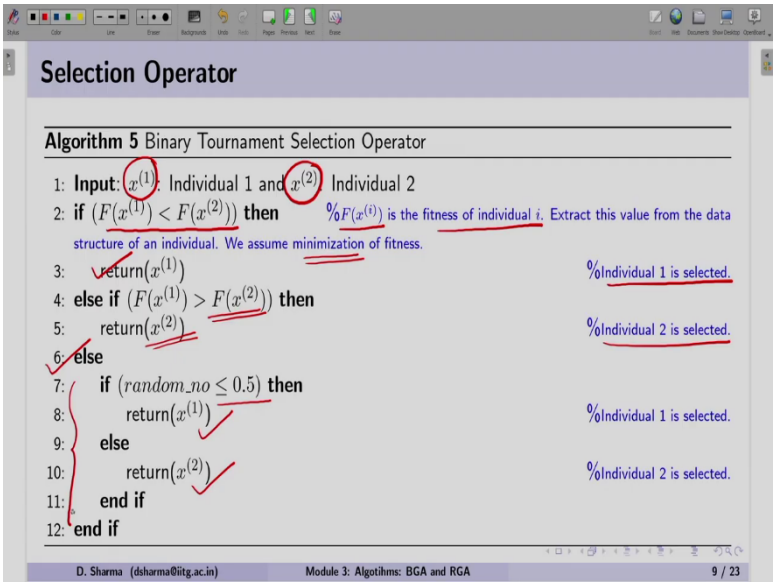
So, in this case we evaluate. So, you can see x j. Now, this x j represents the column vector or the values, the real number real values which we got for this individual j. So, one by one we are actually calculating the function value. Now, here as we have done till now for both binary coded and real coded GA, we assign the fitness same as the function value.

So, in this case, as we can see the parent individual j, objective function value will become the function fitness function value, which we calculated at the step number 2. So, this is exactly the same.

As soon as we calculate we can say that, as in our assumption, the fitness is the same as the objective function value. So, this is what we are considering and this is very simple. And now what the important point which we understood that, the data structure which we generated; as and when we are calculating some value, we are updating that data structure.

So, in the previous; in the previous slide; we find the decoded value, so we updated the data structure. Now, when we are evaluating the population; we are again updating the data structure of say parent population, so that the function value as well as the fitness value can be stored there. Once the evaluation is done, we are in the standard loop of number of generation.

(Refer Slide Time: 16:49)



So, the first step in the standard loop of generation is the selection. We have gone through the tournament selection operator, so we will discuss this operator here. Now, as we know, we take two individuals. So, you can see the individual x 1 and x 2, both of them are taken at randomly. So, this is this process we know.

Now, here I am writing this as a capital F of x I, that represents the fitness of an individual say i. So, in this case what we can do? We can extract this particular value from the data

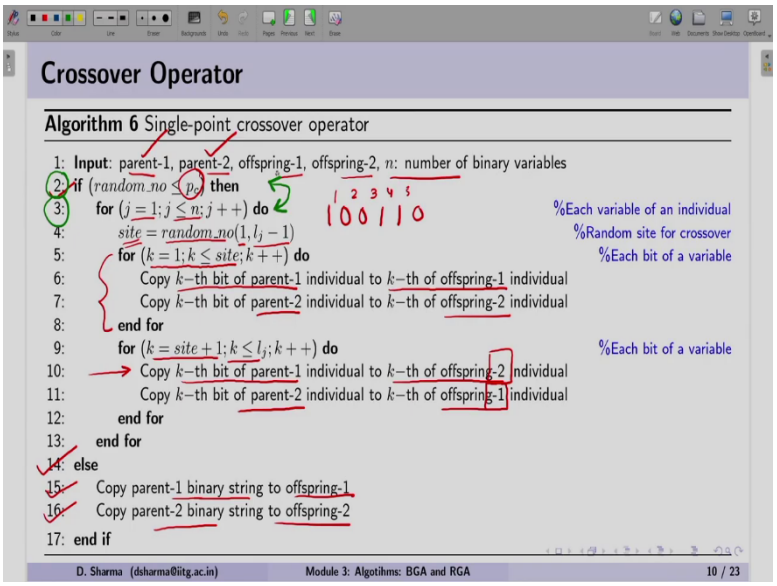
structures. So, as soon as we know which individual, we can extract this value. How this binary tournament operator works? So, in step 2 you can see, we are comparing the two fitness. Suppose the fitness of solution 1 is smaller than the fitness of 2. So, let us assume that, we are solving a minimization problem here.

So, in this case, since the fitness of solution 1 is better, so we will return; return means that, we are actually selecting this individual 1 here. If the fitness value of solution 2 is better than the fitness of 1, then we return x 2. So, here in this case we are selecting individual 2.

In case which is very unlikely that, the fitness of both the solutions are same. So, in this case, we can select anything; just as a algorithmic representation we are writing that, suppose if the random number is smaller than 0.5 we return solution 1 as we return solution 2.

Now, as per the discussion what we can see that, a solution can have a multiple copies and in this scenario, we can select anyone. But when we are solving say a multimodal problem in which we have two different solutions; but their function values are same. So, in this case, such kind of loop, which I have shown you from step 7 to 12 will be useful. So, the representation or binary tournament algorithm representation is easy to select one solution out of two.

(Refer Slide Time: 19:23)



Now, once we perform the binary tournament selection operator, we have to perform the crossover. Now, in this crossover operator, we have gone through the single point crossover

operator. So, let us discuss this. Now, in algorithm 6, you have as a input parent 1 and a parent 2.

Now, we are also taking offspring 1 and offspring 2, because we will be storing the value; similarly the number of binary variable. Now, as we know that when we are picking the parent 1 and a parent 2, we generally pick these two solution at random.

Now, in step 2 we have, we checked the random number smaller than the probability of crossover. So, this is what we performed in our hand calculation. Suppose yes, the random number is small; meaning that we have to perform the crossover operator. Now, in step 3, we work for the number of variable.

So, we are taking one variable at a time. So, for a variable say j, we find the site; since its a single point crossover operator, we have to find this site randomly. So, we are creating this random number from 1 to l j minus 1. Why it is say, why it is l j minus 1? So, let me write an example here.

Say binary string is 1 0 0 1 1 0. So, how many sites are possible? So, I can have 1 2 3 4 or 5. So, I can have 5 sites out of 6 bit string and that is why we are writing random number from 1 to l j minus 1. So, once we have decided the site; so the first task is. So, as you remember that, we have to swap the tail.

In this case, the head of the binary string from the site should remain the same. So, that is why we are running a loop as you can see in step number 5; we are running a loop from k equal to 1 to k site and in this case, every kth bit of a parent 1 is copied to the kth bit of offspring 1.

So, this means that, we are copying the head of the string or you can say the left part from the site; we are copying exactly from parent 1 to parent 2. Similarly, we can copy for parent sorry parent 1 to offspring 1; similarly parent 2 to offspring 2. So, we have copied the head part.

Now, we have to swap the tail. So, simple way is, I can start from site plus 1 and I can go to the last bit in a binary string. Now, you have to be, you can see there is a small change here in step number 10 that, kth bit of parent 1 is copied to the kth bit of offspring 2, so that is the important part here.

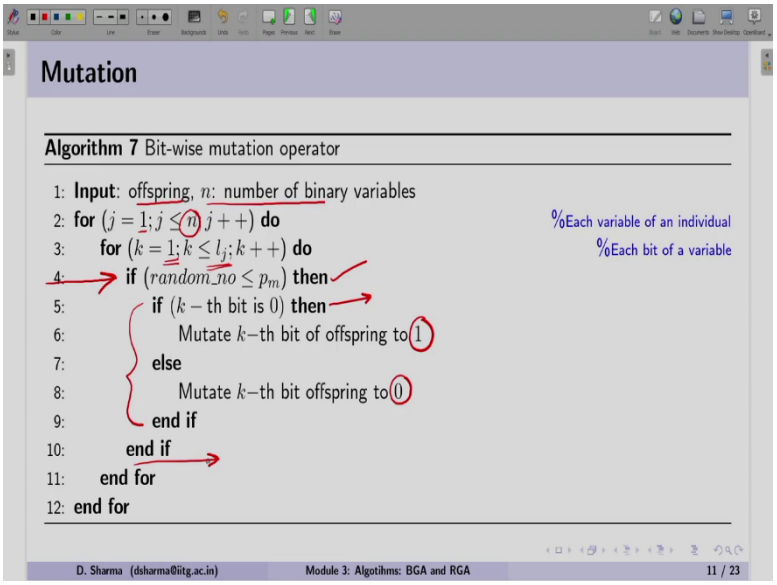
So, we have taken, we have swapped the tail. So, one the bit of 1 is now copied to offspring 2. Similarly, parent 2 kth bit is copied to the offspring 1. So, these are the two important thing which says that, we are swapping the tail. Once it is done, so we have performed the single point crossover operator.

Suppose in the in case the random number is greater than PC, then we have to just copy; as you can see at a step number 15 and 16, we are just copying all the binary string of parent 1 to offspring 1, similarly parent 2 to offspring 2.

Now, there are at many many places or many a times, what we can do? Here is look at the step number 2, now this step number 2 and step number 3; now what we are doing is, as soon as the random number is smaller than the probability of a crossover, then we perform crossover on each on each variable. However, at some places you may find that, you can actually swap these two loops.

Meaning that, for every variable, we are creating a random number and we are deciding whether we have to go with a crossover or not. So, in this case, some variable will be going to the crossover or some not in a one particular set of parent 1 and a parent 2. So, that is second implementation. So, the original implementation which we have followed in our hand calculation, I have shown here; but this swapping of step number 2 and step number 3 are also allowed.

(Refer Slide Time: 24:45)



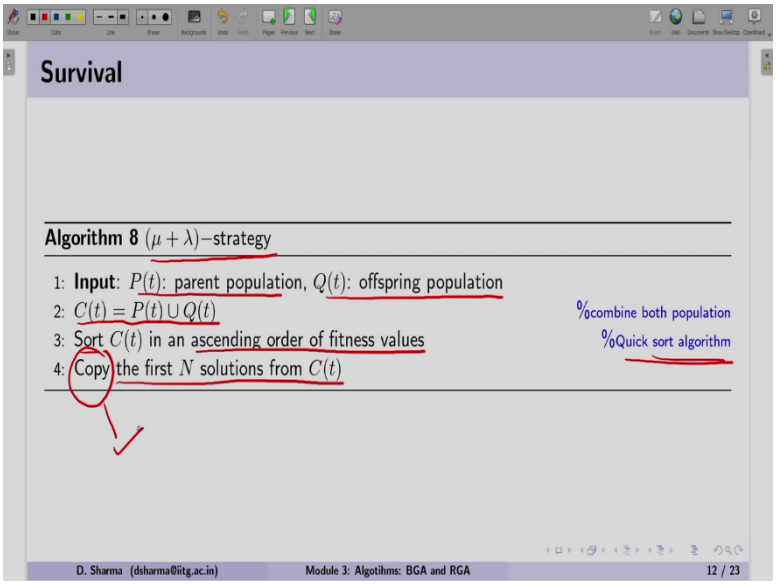
Once we perform this single point crossover operator, now we have come to the mutation operator. So, we as an example, we have taken this bitwise mutation. Now, in this bitwise mutation; what we need an offspring and say number of binary variable.

So, as so since we perform this binary bitwise mutation one by one; so we pick an offspring one by one. Now, look at this step number 2. So, we are running a loop for all the variables; thereafter we run the another loop of k for the number of binary bits we have for variable j.

Now, in step number 4, we are generating a random number. So, that says that, whether we want to perform mutation or not; if yes, then if the bit is 0, we are converting into 1, if it is 1, then we are converting into 0. So, from a step 5 to step 9, these will be mutating the bit. Now, what is the implementation what you can see? That for every bit we are generating a random number and deciding whether we have to perform mutation or not, that you can find it out from step number 4.

If suppose if the random number is greater than probability of mutation, we do not. So, here at this stage, we do not do any kind of a mutation for a one particular bit. So, it is easy to implement this bit wise mutation.

(Refer Slide Time: 26:31)



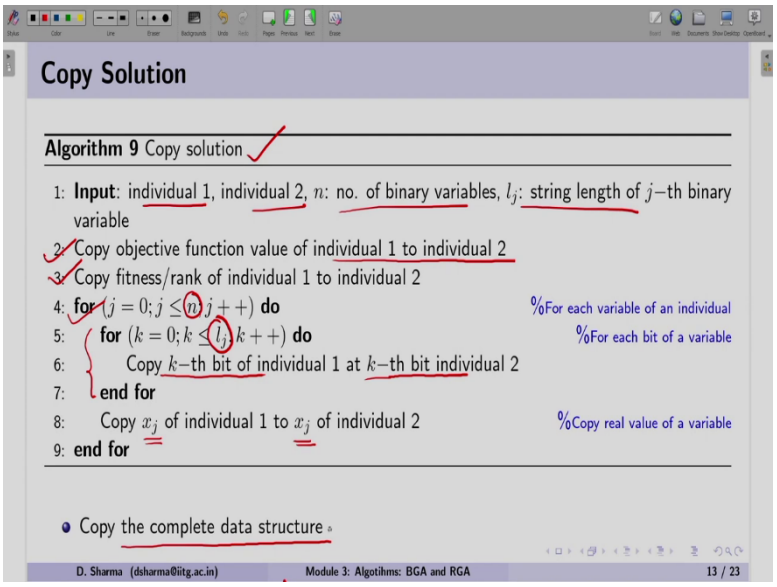
Now, come to the survival stage; this is we know it is the last function or implementation that is required for BGA to work. Now, in this case, you can see algorithm 8 is developed using mu plus lambda strategy.

So, this has been discussed earlier; you can see that for doing this, we need parent population as well as the offspring population after variation operator. So, first step is, we have to copy. So, basically we have to combine the parent population and the offspring population. Once it is done, we can sort this population.

Now, this sorting I can use as I have mentioned here, the quicksort algorithm; there are various algorithm can be used, so that we can or we can write all these solutions in an ascending order of their fitness. Once it is done, we will be copying the first N solution from C t.

So, this is a very simple implementation, where we need only the sorting algorithm and thereafter we copy it. Now, in step 4 when we are copying, this is a important step; why because it says that, the values of an individual should be copied to the another individual.

(Refer Slide Time: 28:08)



So, in order to have more clarity; I am showing you the algorithm 9, which is the copy of this solution. So, let us assume that, we have input 1 and individual 1 and individual 2. So, we are we want to copy the data of individual 1 to individual 2 and we need some, we need other input as number of variables, binary string length and everything we need it.

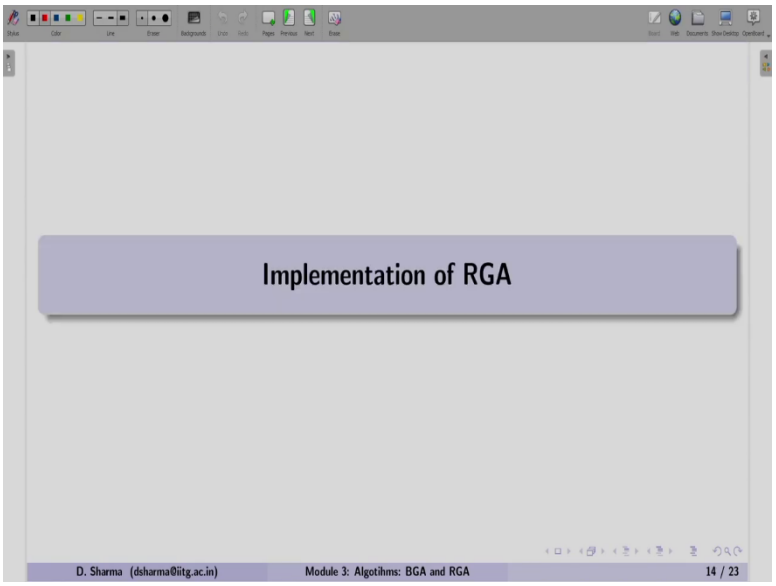
So, first of all what we want is that, we have to copy the objective function from individual 1 to individual 2; thereafter we should also copy the fitness or the rank of individual 1 to individual number 2. Now, as we know as per our data structure, we have a binary string. So,

in this case in step number 4 we can see that, we are running this particular loop for all number of variable; then in step 5, we are running another loop for all binary string.

Now, you looking at the step number 6, we have to copy the k th bit to the of individual one to the at the k th bit of individual 2. So, basically we are copying one by one. And once this is done. So, we have already copied the binary string of solution or individual 1 to individual 2.

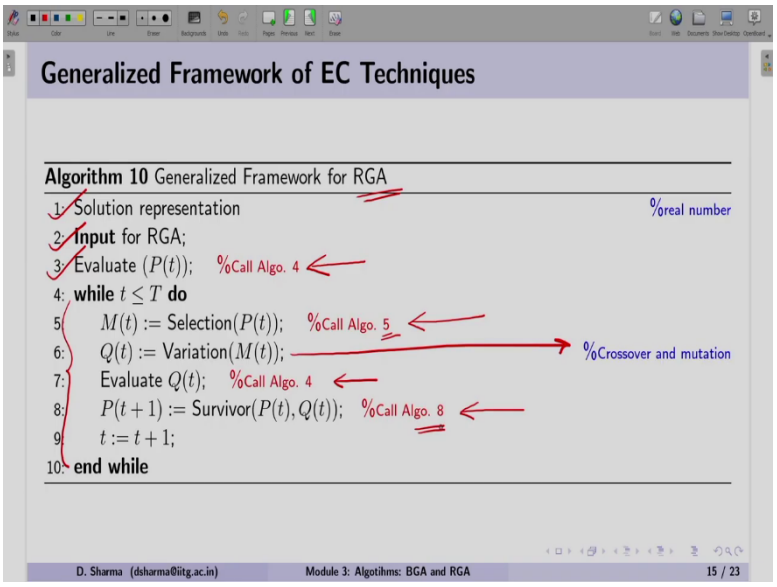
And what is left out is the decoded value. So, this decoded value is also copied into the decoded value of individual 2. So, our main aim is, we have to; we have to copy the complete data structure. Since it is important; so we have included this as a in the algorithmic representation.

(Refer Slide Time: 30:01)



Now, from this we know, we have gone through all the major and important operators that has been written in a algorithmic way for binary coded GA. Now, suppose we want to implement for the real coded GA, then what are the changes we need it.

(Refer Slide Time: 30:23)



Now, here as you can see in algorithm 10, we are targeting RGA. Now, the solution representation in step 1 is the real number; thereafter in step 2, we have certain input, then in step 3 we have to evaluate the population. Now, since the evaluation will remain the same; therefore we are not going to create any function, we just call algorithm 4 which we discussed earlier.

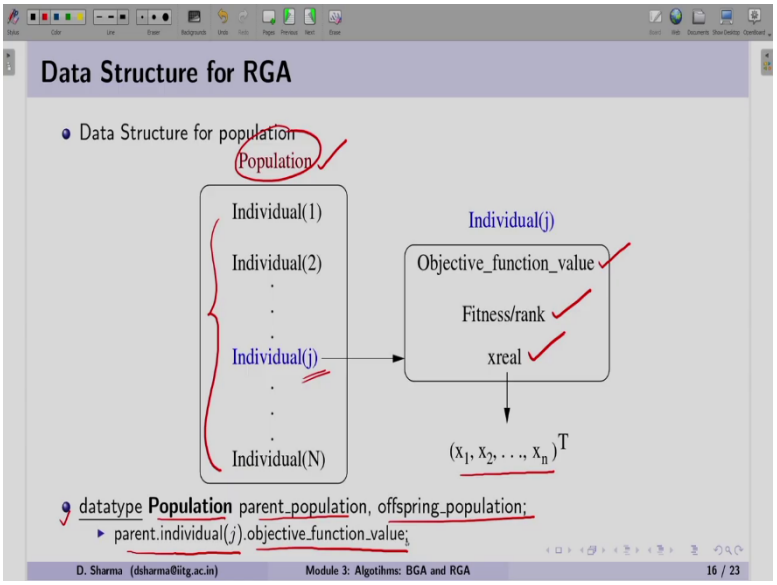
Thereafter, we are in the standard loop of number of generation here and then in step 5, we have to perform the selection. Now, if we are using the binary tournament operator; then we do not have to create any new function, because we already discussed how the binary tournament selection operator work. So, the algorithm 4, we algorithm 5 now in this case; we can use it as it is and that can be used with the RGA as well.

Step 6 is the crossover and a mutation operator and we know that, this we have to change it; because the crossover and mutation operator for real variable, real variable is different. Now, again at step number 5, we have to evaluate the offspring population. So, again we will be calling the algorithm 4, which we have discussed earlier and then in step number 8, we have to perform the survival stage.

So, if we are using mu plus lambda strategy, we can directly call our algorithm number 8, so that our sorted population based on the fitness can be copied. So, this is the way you can see that, if any code or a function is written; we have to just use it as it is. Now, in this case

algorithm 4, algorithm 5 and algorithm 8 can be used as it is what we have discussed with binary coded GA.

(Refer Slide Time: 32:38)

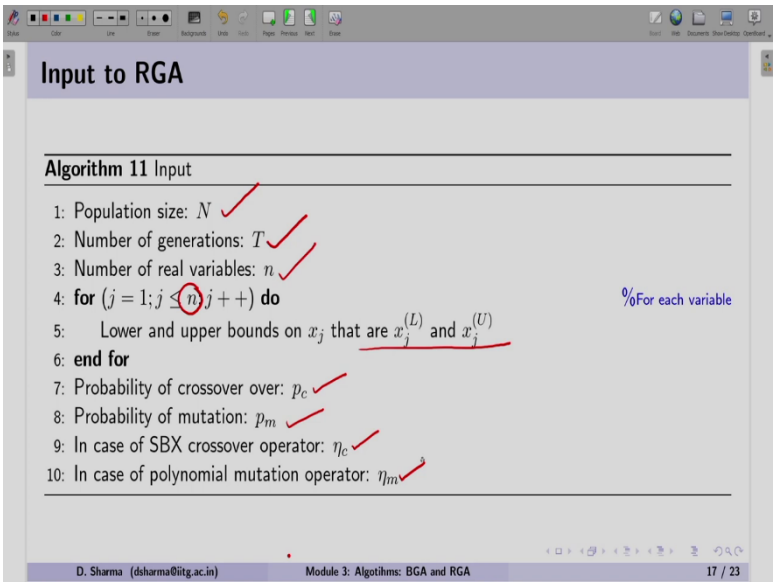


Now, let us start with the data structure of RRGA. Now, the population data structure remains the same, where we have solutions from 1 to N. Now, let us take a ones individual say j. Now, in this case we will be storing the value of objective function, the fitness value and the real number. Now, we do not need any chromosome or decoding. So, we are storing the x real value. So, x real means that, we are saving this column vector of N variable.

Now, here we are assuming that all variables are real. So, the data structure here, the same definition of data structure called population can be used as you can see on the top, which consist of N pair N solutions. So, we can say, we have parent population and we have offspring population.

And in a similar way, we can extract the value; say for example, parent dot individual j dot objective function value. So, we can extract, similarly if we want to copy; we can call similar kind of similar kind of a command for the data structure, so that we can store the value.

(Refer Slide Time: 34:00)

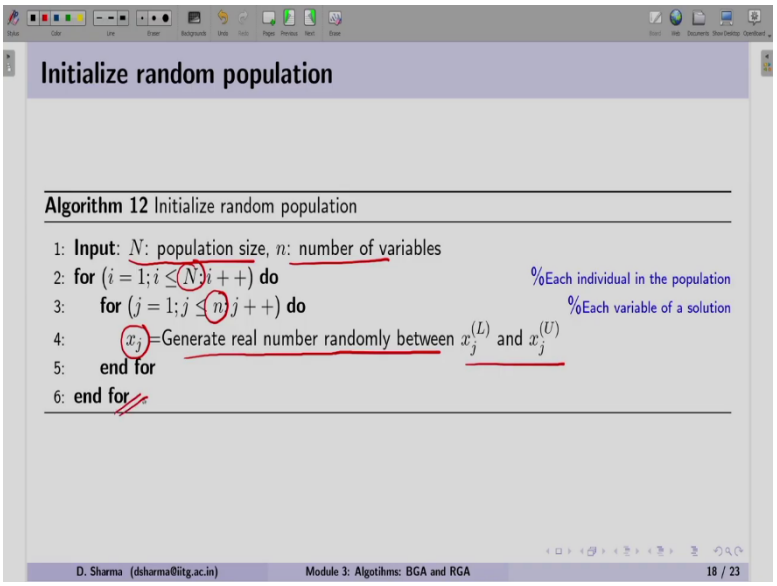


Now, let us see the input to RGA. Now, there are certain inputs that, remains that remain the same for both BGA and RGA. Now, we need a population size that remains the same; number of a generation will also remain the same and the number of variable. So, last time it was binary variable, now this time number of real variables. Now, the small change which you can see that in step number 4, we are running a loop for number of real variable.

Since we already since the variable is represented in a real number; so we only have to provide the lower and upper bound of the variable j. Thereafter, again we have to assign probability of crossover as p c; probability of mutation as a p m.

And now, since the crossover and mutation operators are change and we have already worked on SBX crossover operator and polynomial mutation, so let us take this these two operators. So, in this case eta c is required for crossover operator and eta m is required for polynomial mutation operator.

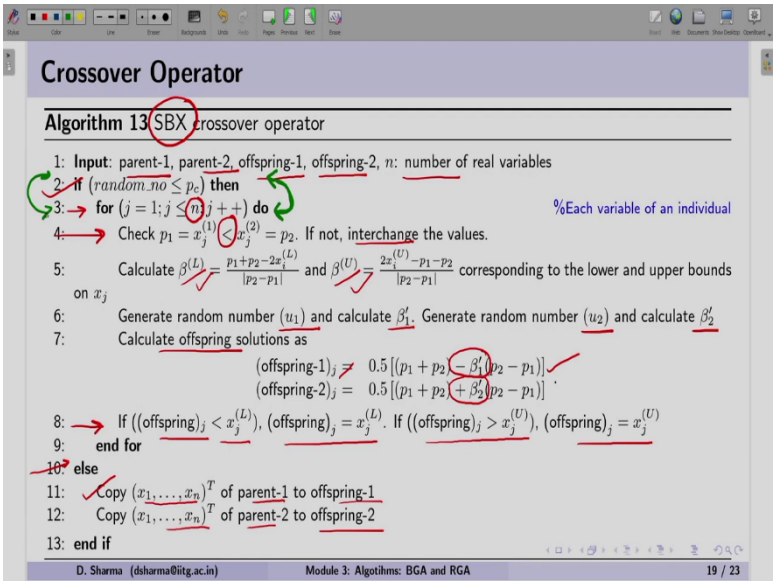
(Refer Slide Time: 35:21)



Now, let us start with initialize random population; it is because we have to generate the number. So, the input to this algorithm 12 is population size and the number of real variables. So, we are running a loop for each member or each individual of the population; then we are running another loop for the number of real variable.

So, let us take the variable x j and this value will be we what we will do here is, we generate a random number between the lower and the upper bound. So, in this case, all the solutions or the variables that will be generated within the bound and afterwards we can finish this initialization process.

(Refer Slide Time: 36:10)



Now, let us come to the crossover operator, now as we remember that; we have to give the input to RGA, thereafter we have to evaluate it. Now, this evaluation we are not discussing, because that evaluation is the same as what we discussed with the implementation of BGA, that is binary coded GA.

Thereafter in a standard loop, we had binary tournament selection; since binary tournament selection will remain the same; because it depends on the fitness value, so therefore, the functioning will remain the same and we are we have not included here. We can call the same function, which we have discussed with the binary coded GA.

Now, we have come to the crossover operator. Now, in this crossover operator, we are discussing the SBX crossover operator; because we have already done some hand calculations here. So, input to this is we have a parent 1, then we have a parent 2. So, we remember that we pick 2 solutions randomly p 1 and a p 2 and then we also have offspring 1 and offspring 2; because we save the new solution or a new value into offspring 1 and a 2 and the number of variables.

Now, in step 2 as you can see that, we create a random number; if random number is smaller than probability of a crossover, then we will perform the crossover. Now, looking at the step 3, assuming that yes we are performing the crossover; we have a for loop on the number of variables.

Now step 4 is important; because this is which we have discussed earlier, then that the parent 1 should be smaller than parent 2. So, this condition you can see here; if it is there, it is fine; if it is not, then we have to interchange the value.

Once it is done. So, here we are calculating beta L and beta U. Now, this representation or this SBX operator is the same operator, which we discussed for bounded variable; because we have lower and upper bound, we want our SBX operator should generate the value in between.

So, that is why the same set of equations are used here to calculate or to find the offspring. So, in step 5, we will calculate the beta that is corresponding to the lower limit of on the variable j and beta u will be corresponding to the upper limit of the variable j.

Once we find out the beta values, we generate the random numbers say u 1 for beta one prime and u 2 for beta 2 prime. So, in step 6 we get the data what we needed and thereafter, we calculate the offspring in a step number 7. So, it is simple; why because, we already know the formula for that. So, offspring 1 jth variable can be represented in this term. Now, important point is, now we are using minus beta 1 prime and for the offspring 2, we are using plus beta 2 prime.

So, that will help us to generate offspring between lower and upper bound. But as a on the safer side in step number 8, in case if the variable is still out of the bound; we are including one condition. In this condition, if the j th component of offspring is smaller than the lower bound; then we are putting this on the lower bound. Similarly, if the j th component of offspring is greater than the upper bound, then we are taking this on the upper bound.

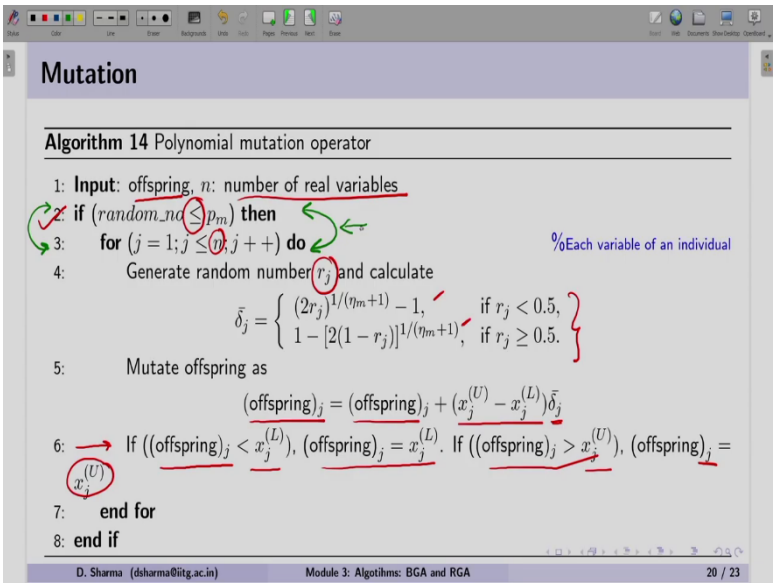
So, that will make sure that our variables are under lower and upper bound. Once this for loop is done, then we are at the step number 10. Now, this 10 says that, in case the random number is more than probability of crossover; so we should not perform it. In this case, in step number 11 as you can see; we can copy x 1 to x n basically a column vector of parent 1 to offspring 1, similarly the column vector of parent 2 is copied to the offspring 2.

Now, here you can realize there may be some other implementation, which can be which can be using these two condition interchangeably. So, I am using this step 2 and step 3 and I am putting this arrow it is; because if suppose we take step 3 above and a step 2 below that says

that, we are creating a random number for each variable and deciding whether we have to perform the crossover operator or not.

So, in our hand calculation, we discuss that as soon as the random number is smaller than probability of crossover; then we will be performing crossover on each variable. But the another implementation as I suggested, if we are swapping this step number 2 and step number 3; which says that, we want to generate the random number for each variable of a solution and then accordingly we will be performing the crossover. So, the small change can be done for having two kinds of variation in SBX operator

(Refer Slide Time: 42:15)



Now, coming to the polynomial mutation. Now, in this polynomial mutation, since we have worked on hand calculation; so let us see how it works. For this we have an input of offspring. So, as you know we need only one solution and the number of variables. So, in step 2 it is the same condition that, we have to generate and check whether it is a smaller than equal to probability of mutation; if yes, we will be performing mutation on every variable.

So, in step number 3, you can check that the for loop is executed for all the variable; thereafter we generate in step 4, we generate a random number say r j and then according to the r j. So, this particular formula we already know for the polynomial mutation and based on the value of the random number r j, we are going to use either the formula number 1 or formula number 2.

And using this delta i value, we will be mutating the offspring j with respect to its previous value plus the difference between the lower and upper bound multiplied by the delta j value. So, this is the same equation what we have written for mutation.

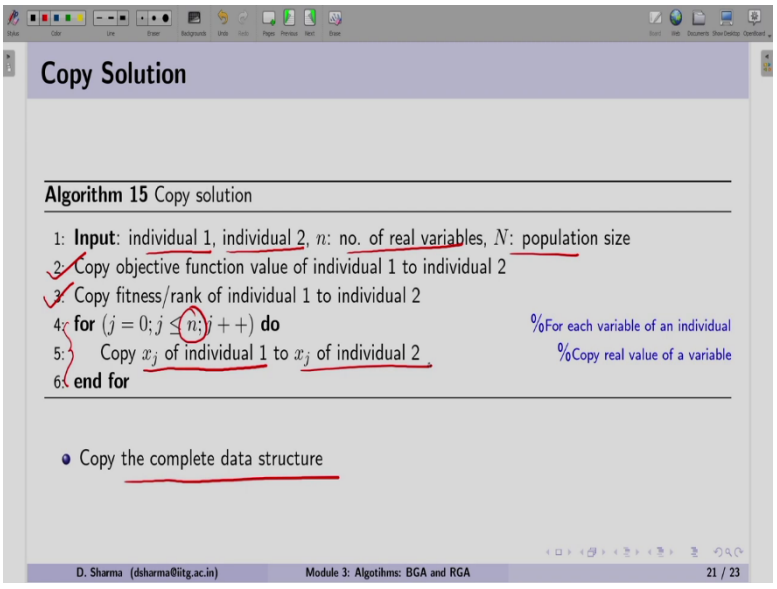
So, that will change or that will mutate an offspring using polynomial mutation. As I check in step number 6, we are we are making sure that offspring should be generated within the bound. So, in this case the offspring, so j th component of the offspring; if it is smaller than the lower bound, then we are putting it on the lower bound.

If the j th component of the string is on the upper bound, in this case we are putting this on the upper bound. So, this small check we included, so that we can make sure that even after mutation; the solution is within the limits or the range.

Now, again if we discuss the swapping; so I am considering step number 2 and step number 3. So, if we are going to swap these two step. So, currently what we are doing that, as soon as the random number is smaller than probability; we will be performing mutation for every variable.

However, if we swap it between 2 and a 3 step that indicates that, for every variable we will generate a random number and then decide whether we want to perform a mutation for say variable j or not. So, those 2 implementations are easy and we can include any one of them.

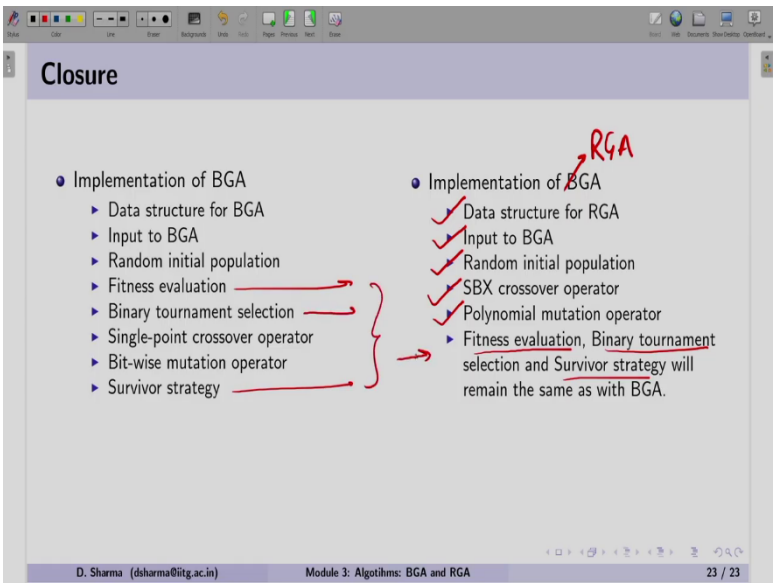
(Refer Slide Time: 45:20)



Now, coming to the copy of this solution; as you can see in algorithm 15, our main objective is to copy the complete data structure as it is mentioned in at the last of this slide. So, again what we need is the individual 1 and the data of individual 1 will be copied to individual 2 and we need for example, number of real variable and the population size.

So, first point is, let us copy the objective function value of individual 1 to individual 2. Similarly, thereafter we can copy the fitness of individual 1 to individual 2; the in step from step 4 to 6 as we can see, there is a for loop on the number of variables. So, for every variable say j th variable, we are copying the x j; so basically a full column vector, we are copying to the individual 2, so x j to individual 2. So, please note that, we are copying each and every, each and every variable of individual 1 to individual 2.

(Refer Slide Time: 46:40)



So, now let us come to the end of this implementation. In this particular session, we have gone through the implementation of a BGA first, where we discussed the data structure which is an important part of an algorithm. So, before we start implementing this algorithm, we should know what kind of data structure we should use it. We then we discussed about what kind of inputs we needed, so that we can call that function and every input should get for running the BGA.

We generated the random population in terms of binary string and finally, making the chromosome. Fitness evaluation was easy; because every individual one by one, we calculate the objective function and the same objective function value will become the fitness as of

now. We perform the binary tournament selection of picking the two solutions randomly. So, we compare the fitness values and accordingly we return one solution.

Single point crossover operator, it is the simplest operator, where we find the site; in the site we will be copying the head of the binary string as from individual or from parent 1 to the offspring 1. And the tail as we swap the tail; so the from site to the last bit, we are interchanging those bits from parent 1 to offspring 2, similarly from parent 2 to offspring 1. So, it was a easy implementation; thereafter we have a bitwise mutation.

Now, in bitwise mutation, as soon as the probability is satisfied, the random number is smaller than the probability of mutation; we check that every bit will be will be mutated from 0 to 1 or 1 to 0. And thereafter, we had a survival stage, which is mu plus lambda. So, in that case, we have to combine it; we have to perform sorting and then we have to copy. So, the copy also we have we have gone through; because the main objective is we have to copy the data structure as it is from individual 1 to individual 2.

Thereafter we discussed r j. So, this should be. So, the in this case, this is RGA. So, the implementation of RGA includes the data structure of RGA and you can see that; we do not need chromosome, decoding and the scaling function. So, we are representing the variable as it is in the real number.

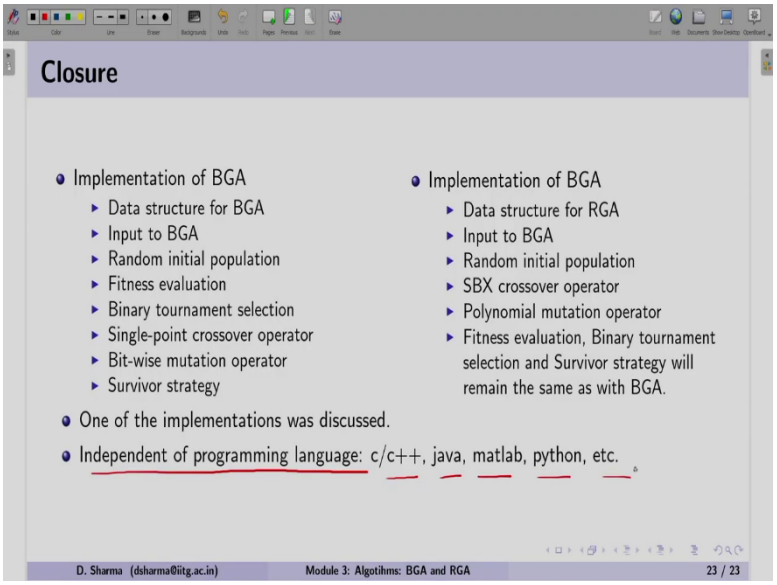
Thereafter, the input has been changed; it is only because the variables are represented in the real number, second the crossover and mutation operator need other parameters that should be fixed by user.

Random population was generated according to the lower and upper bound of the variable, which was not required in binary GA; because the scaling function will take care of it. But in real number that can go beyond, so that is why we generate the random number between lower and upper bound. In case of crossover operator, we performed the SBX and that the implementation was shown for the bounded SBX, where the SBX operator should generate a new offspring solution within the bound.

However, on the safer side, we included the way, if suppose it is going out of bound; thereafter we discuss polynomial mutation. In this polynomial mutation, we used the, we calculate the delta i value and used it to mutate a individual or an offspring. An important point we found that, the fitness evaluation or the binary tournament selection or the survival

stage, we do not have to change; because the same functions that has been used with binary coded GA can be used as it is with the real coded GA.

(Refer Slide Time: 51:01)



At the end what we have discussed is one of the implementation. So, that is why if you search, there will be various other kinds of implementation available in the literature. So, this is one of the implementation and that is also you can use it, you can write your own binary coded or a real coded genetic algorithm code for you.

Here important point is that, all the algorithm representation which are shown in this session, they are independent of any programming language. It is because, it depends on the user which choice or which programming language they want.

So, therefore, the algorithm representation is starting from the generalized formulation or framework, followed by the other functions or implementation we have discussed; for example, binary tournament selection operator, crossover mutation, all of them can be easily implemented using either for example, I have mentioned as C, C ++, we can use java, MATLAB, python etcetera.

So, those languages we can use it and since these implementations are independent, we can easily make our own code and solve our problem. So, with these details on implementing the binary coded and real coded in an algorithmic way, I conclude this session.

Thank you.