International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 2, Issue 5, pp 1-11, July-2018

Creating a Model for Genetic Mutants Using Integer Programming Techniques

[1]Ahmadu A. D, [2]Ahmadu S. Asabe PhD, [3]Philemon Uten Emmoh

[1,3]ICT Center, Federal University, Wukari, Nigeria.

[2]ModibboAdama University of Technology, Yola, Nigeria.

***Abstract* – Improving the quality and functionality of life is becoming an increasing concern globally. This has led to studies to help develop better varieties of life products with improved qualities and reduced defects. This paper uses the techniques of integer programming and genetic algorithm to develop a Genetic Algorithm GA that evolution schemes of species, in which we have a population of individuals, plants, or life element represented by their own chromosome, as time goes by we breed them, mutate some of them and select who shall live and who shall die, and p a better an individual. A programmatic implementation of the schemes is given in C++ language.**

***Keywords*:** Genetic, Algorithm, Chromosome, Evolution, Life, Quality.

I. INTRODUCTION

Linear programming is a well-known technique for solving problems mainly in industries. It is used to achieve better energy use, reduction in costs of manufacturing, scheduling and many others. Linear programming problems are normally expressed as follows in its basic form:

Minimize/maximize (FXi) = C1*X1 + C2*X2 + C3*X3 +…+Cn+Xn

Subject to A1*X1 + A2*X2 + … +AnXn<DB1*X1 +B2*X2 + … +B3*X3<E

Where, Xi is a vector of variables to be solved for.

Ci is a vector of costs for each variable (depending on application), and Ai, Bi are the vector of constraint coefficients , D and E are the constraints itself. Integer programming is an extension to the kind of problem, where we have a special restriction that all variables must be integer. A well-known and used method for solving this kind of problem is the Simplex method that, although it has exponential complexity in theory, in practice it is polynomial. But this method can't deal with the integer constraint (i.e. the variable must be of integer type), so we must use of another method to solve those kind of problems.

II. RELATED WORK

The basic principles of GA were first proposed by Holland [1]. Thereafter, a series of literature [2], and reports, [3], became available. GA is inspired by the mechanism of natural selection, a biological process in which stronger individuals are likely be the winners in a competing environment, Here, GA uses a direct analogy of such natural evolution. It presumes that the potential solution of a problem is an individual and can be represented by a set of parameters. These parameters are regarded as the genes of a chromosome and can be structured by a string of values in binary form. A positive value, generally known as fitness value, is used to reflect the degree of "goodness" of the chromosome for solving the problem, and this value is closely related to its objective value.

Brac*aet al* (1997) offered a computerized approach for the transportation of students to multiple schools located in New York, USA. Their problem includes capacity, distance and time window constraints. In addition, they require that a lower bound on the number of students that form a route should also be respected. The problem consists of 4619 students to be picked up from 838 bus stops and transported to 73 schools. The authors proposed a routing algorithm based on the location-based heuristic for the capacitated VRP. One interesting aspect of this study is the estimation of distances and travel times, which are performed via a geographic information systems-based program (MapInfo) and a regression analysis, respectively. The authors also presented two integer programming formulations, namely a set partition model and an assignment-based model, although these do not explicitly include the capacity and distance constraints and are not utilized in the proposed routing algorithm.

A recent study related to the subject is due to Li and Fu (2002). These authors provided planning techniques for a single SBRP in Hong Kong, China, which consists of trans-porting 86 students located at 54 pick-up points. The problem

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 2, Issue 5, pp 1-11, July-2018

is of a multi-objective nature, including the minimization of the total number of buses used, the total travel time of all the students, the total bus travel time and balancing the loads and travel times between buses. A heuristic algorithm is proposed to solve the problem. The authors also presented a three-index flow-based integer programming formulation, which, however, is not utilized in the solution algorithm.

The OVRP has recently attracted attention from the operations research community. One of the first studies on the OVRP is due to Sariklis and Powell (2000), who proposed a heuristic solution to solve the capacity constrained version. Later on, Tarantilis*et al* (2005) described a single parameter meta-heuristic algorithm for the problem. To the best of our know-ledge, the only exact method to solve the capacitated OVRP is due to Letchford*et al* (2006).

a) Statement of Problem

First of all, we formulate and prepare the variables of a simple example problem that will be used later when the programming action begins. Two Toy Star factories produce a best-seller (bought mainly by our smart Willy Cool Toys Service) toy bomb-doll. But each toy requires an amount of 100kg of gunpowder for a high explosive solution used on it fabrication.

We have 3 suppliers that produced it, each one with a different price:

S1: N800.00 / ton

S2: N300.00 / ton

S3: N300.00 / ton

To ship the products from supplier to a factory also has a cost:

	To:	A	B
From:	S1	N500.00	N600.00
	S2	N900.00	N800.00
	S3	N600.00	N700.00

The stocking of these powders has a different price in each factory:

A:N900.00

B:N700.00

And we have a limit of how much we can stock, in factory A we can have as much as 550 tons, and in B we can have 700 tons. The suppliers can offer a limited amount of gunpowder, S1 can produce 390 tons, S2 can make 460 tons and S3,370 tons. Each bomb-doll has a cost of N200.00 to produce them (so he can catch that nasty Ruud Runner competitor), we must, then maximize the profit based on these. First of all, we must put this on Linear Programming form: Let's call P(Xi) as the profit function of the doll sells, where Xi is the vector variables, this is calculated by subtracting all the costs from the price of each doll. Since each doll in sold for &320.00 and it costs &20.00 to produce, we start with:

P(D) = 320*D – 200*D = 300*D

Where D is the number of polls produced. But besides production cost we also have the gunpowder costs, the essential part of production. So let's do some naming:

Tij is the amount of gunpowder acquired from supplier i and taken to factory j

Ti is the amount of gunpowder acquired from supplier i

Tj is the amount of gunpowder taken to factory j

T is the total amount of gunpowder bought

So we will have 13 variables as follow:

T1A, T2A, T3A, T1B,T2B, T3B, T1, T2, T3, TA, TB, T,D

From that we can deduce the costs with material, having the prices of each supplier, we have

C1(T1, T2,T3) = 8*T1 + 3*T2 + 5*T3

As in the first cost equation, let's consider the transport of cargo as the second cost, so:

C2(T1A,T2A,T3A,T1B,TB2,T3B) = 5*T1A + (9*T2A + 6*T3A + 8*T2B + 7*T3B

And finally the costs of stocking:

C3(TA,TB) = 9*TA + 7*TB

Completing then the profit equation:

P(T1A, T2A, T3a, T1B, T2B, T3B, T1, T2, T3, TA, TB, T, D) = 300*D – C1 – C2 – C3

= 300*D – (8*T1 + 3*T2 + 5*T3)

- (5*T1A + 9*T2A + 6*T3A + 6*T1B + 8*T2B +7T3B) - (9*TA + 7*TB)

Constrained to:

TA <=550 (factory A can only hold 550 tons)

TB <=700T1 <= 390 (SUPPLIER 1 can only supply 390 tons)

International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 2, Issue 5, pp 1-11, July-2018

T2 <=460T3 <= 370T1A, T2A, T3A, T1B, T2B, T3B, T1, T2, T3, TA, TB, T, D

Integers

T1A, T2A, T3A, T1B, T2B, T3B, T1, T2, T3, TA, TB, T,D>=0

Now we have huge equations with lots of variables, but if we think a little more we can reduce it to six variables, so that:

(the sum of all gunpowder required from all 3 suppliers that we sent to factory A)

TA = T1A +T2A + T3A

TB = T1B + T2B + T3B

T1 = T1A + T1B

T2 = T2A + T2B

T3 = T3A + T3B

T = T1 + T2 + T3 = TA + TB

This reduces to 7 variables: T1A, T2A, T3A, T1B, T2B, T3B, D

But since each doll requires 100KG OF gunpowder, OR 0.1 TON, WE HAVE:

1*T = 10*D => D=T/10

So we must focus only on the six basics variables: T1A, T2A, T3A, T1B, T2B, T3B and our profit equation will be:

P(Xi) = 300*(T/10) – (800*(T1A + T1B) + 300*(T2A + T2B) + 5*(T3A + T3B))

- (5*T1A + 9*T2A + 6*T3A + 6*T1B + 8*T2B + 7*T3B)
- - (9*(T1A + T2A + T3A) = 7*(T1B + T2B + T3B))

b) Genetic algorithm

Now let's talk of the APPROACH TO BE USED IN THIs PROBLEM, the Genetic Algorithm (GA) from now on). GA is a techniques based on evolution schemes of species, in which we have a population of individuals, represented by their own chromosome, as time goes by (and so the 'while loop') we breed them, mutate some of them and select who shall live and who shall die, since the better an individual, greater the chance it has to be chosen for reproduction and for the next generation, we come up with a solution closer to the optimum (hopefully global) with time we get to a new generation.

c) How does it look like?

The first thing to concern about coding an GA is the codification, or chromosome of each individual. If we are dealing with integers, the most efficient way to code it is using an array of bits, making the crossover operation now "active" (this way we can generate new numbers each time we use it, if we had an array of integers we would just swap the numbers among parents, not building new ones) we will get on that later. For now let's face a bigger problem: "how long should my array be?. Remember that we must be sure to fit the values to reach the global maximum. To solve this question, let's take a look at the constraints:

T1A + T2A + T3A <=550

T1B + T2B + T3B <=700

T1A + T1B <=390

T2A + T2B <=460

T3A + T3B <=370

If we take a variable and "zero" the others we can take a maximum allowed value for this. Let's take the T1A variable as an example:

T1A + 0 + 0 <=550

T1A + 0 <=390

From this we can see that T1A can be no more than 390, now, as we're going to treat it like a binary number, let's see how many digits we must use for it:390 in binary is 110000110 which has 9 bits, so our first variable will occupy the first 9 indexes of the array. Calculating the bits for the others variables we get:

T1A = T1B = T2A = T2B = T3A = T3B = 9 bits

So we need array of 9*6 bits total. An individual can be represented only by its chromosome, but to avoid some wasteful calculations we put together two other information: fitness, which gives each individual a "score, and prob, which represents the probability to be chosen for breeding of to live. So our first code starts here:

```
/*BITS USED BY EACH VARIABLE*/
#define T1A_BITS 9
#define T1B_BITS 9
#define T2A_bits 9
#define T2B_BITS 9
#define T3A_BITS 9
#define T3B_BITS 9
/*Total bits */
```


International Research Journal of Innovations in Engineering and Technology (IRJIET)
ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

```
#define BITS T1A_BITS + T1B_BITS + T2A_BITS +
T2B_BITS + T3A_BITS + T3B_BITS
/* How we represent each indidivujdla, which its
chromosome, its fitness, and its probability */
Type def structcrom
 {
Char cromo[BITS];
Long int fitness;
int pro;
} crom;
```

III. METHODS

Well, having defined how each individual is represented we must create a function that will show us what does this representation means (i.e. transform a bit array into an array of 6 integers number). This function will have the chromosome and a pointed to an array of size n (number of variables) passed as parameters, so we can decode the bits as decimal numbers. We will have something like this:

```
----------galib.c----------
void decode (cromindv, intvars[]){
   IntI,j;
   /*initialize the variables */
   Vars[0] = 0;
   Vars[1] = 0;
   Vars[2] = 0;
   Vars[3] = 0;
   Vars[4] = 0;
   Vars[5] = 0;

/************************************************
   *Let's walk the first 9 bits (T1A) from left to right: *
   *2^8 + 2^7 + … + 2^0
   * so, for i = 0 to 9 we use the formula: bit*2^(BITS_1-i)*

************************************************
   For (i=0; i <T1A_BITS;i++){
   Vars[0] +=indv.cromo[i]*(int)pow(2, T1A_BITS -1-i);
   }
/************************************************
   *here 'i' already starts as 9 from the above loop, *
   *let's make different this time, let's initialize *
*j to BITS-1 (8) and decrement it until it's less than 0, *
* and keep incrementing i to keep going all variables *
************************************************
for(j=T1B_BITS-1;j>=0;1++,--j){
vars[1] += indv.cromo[i]*(int)pow(2,j)
}
/* and so on …*/
for{j=T2A_BITS-1,j>=0;1++,--j){
vars[2] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T2B_BITS-1,j>=0;1++,--j){
vars[3] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T3A_BITS-1,j>=0;1++,--j){
vars[4] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T3B_BITS-1,j>=0;1++,--j){
vars[5] +=indv.cromo[i]*(int)pow(2,j);
}
}
```

Now we have the binary representation and its decoded values for an individual, we must now calculate its fitness so we can eval one by one in a population. First of all, let's calculate the function we want to maximize, the objective function:

```
----------insert into galib.c-----------
Long intobject(intvals[]){
Long intobjt;
Int D_30, T, TA, TB, T1, T2, T3;
/*Let's calculate some intermediate values */
TA = vals[0] + vals[1] + vals[2];
TB = vals[3] + vals[4] + vals[5];
T1 = vals[0] + vals[3];
T2 = vals[1] + vals[4];
T3 = vals[2] + vals[5];
T = TA + TB;
 /* D*300 = (T/10)*300 = 30*T so now we don't need
to deal with float point numbers */
D_30 = 30*T;
/*now let's apply the objective function to it */
Objt = D_30
```


ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

```
Objt -= (8*T1 + 3*T2 + 5*T3);
Objt - = (5*vals[0] + 9*vals[1] + 6*vals[2] + 6*vals[3] +
8*vals[4] + 7*vals[5]);
Objt-=(9*TA + 7*TB);
Return objt;
}
```

Well, having defined how each individual is represented we must create a function that will show us what does this representation means (i.e. transform a bit array into an array of 6 integers number). This function will have the chromosome and a pointed to an array of size n (number of variables) passed as parameters, so we can decode the bits as decimal numbers. We will have something like this:

----------galib.c----------

```
void decode (cromindv, intvars [ ]){
IntI,j;
/*initialize the variables */
Vars[0] = 0;
Vars[1] = 0;
Vars[2] = 0;
Vars[3] = 0;
Vars[4] = 0;
Vars[5] = 0;
/***********************************************
*Let's walk the first 9 bits (T1A) from left to right: *
*2^8 + 2^7 + … + 2^0
* so, for i = 0 to 9 we use the formula: bit*2^(BITS_1-i)*
***********************************************
*** /
For (i=0; i <T1A_BITS;i++){
Vars[0] +=indv.cromo[i]*(int)pow(2, T1A_BITS -1-i);
}
/***********************************************
**
*here 'i' already starts as 9 from the above loop, *
*let's make different this time, let's initialize *
*j to BITS-1 (8) and decrement it until it's less than 0, *
* and keep incrementing i to keep going all variables *
***********************************************
for(j=T1B_BITS-1;j>=0;1++,--j){
vars[1] += indv.cromo[i]*(int)pow(2,j)
}
/* and so on …*/
for{j=T2A_BITS-1,j>=0;1++,--j){
vars[2] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T2B_BITS-1,j>=0;1++,--j){
vars[3] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T3A_BITS-1,j>=0;1++,--j){
vars[4] +=indv.cromo[i]*(int)pow(2,j);
}
for{j=T3B_BITS-1,j>=0;1++,--j){
vars[5] +=indv.cromo[i]*(int)pow(2,j);
}
}
```

Now we have the binary representation and its decoded values for an individual, we must now calculate its fitness so we can eval one by one in a population. First of all, let's calculate the function we want to maximize, the objective function:

----------insert into galib.c----------

```
Long intobject(intvals[]){
Long intobjt;
Int D_30, T, TA, TB, T1, T2, T3;
Let's calculate some intermediate values */
TA = vals[0] + vals[1] + vals[2];
TB = vals[3] + vals[4] + vals[5];
   T1 = vals[0] + vals[3];
T2 = vals[1] + vals[4];
T3 = vals[2] + vals[5];
= TA + TB;
 /* D*300 = (T/10)*300 = 30*T so now we don't need
to deal with float point numbers */
D_30 = 30*T;
/*now let's apply the objective function to it */
Objt = D_30
Objt -= (8*T1 + 3*T2 + 5*T3);
Objt - = (5*vals[0] + 9*vals[1] + 6*vals[2] + 6*vals[3] +
8*vals[4] + 7*vals[5]);
Objt-=(9*TA + 7*TB);
Return objt;
```


International Research Journal of Innovations in Engineering and Technology (IRJIET)

ISSN (online): 2581-3048

Volume 2, Issue 5, pp 1-11, July-2018

}

---------- Cut here ----------

IV.RESULTS AND DISCUSSIONS

a) Constraints

Wait a moment, you must be thinking, if we evaluate each individual through just the objective function, how we can guarantee that it will respect the constraints? Well, you're right; we must avoid the solutions that break the constraints so we don't get a wrong answer. To control it we have four options:

- Correcting the individuals that do not satisfy the constraints
- Eliminating the individuals that do not satisfy the constraints
- Adding a penalty function to diminish the individuals that do not satisfy the constraints
- Make the crossover; mutation and encoding take care of this.

For this project let's focus on the penalty function, for it preserves those individuals who aren't satisfactory but can generate a child that is. There should be some experiments to decide which one is more effective, and even that can vary from problem to problem. The penalty function will be defined as "how much the value has passed from the constraint" multiplied by a constant rate to make the individual worse. So the more you get above the constraint less will be your fitness. Once we have 5 constraints, we will have 5 penalty functions, defined as:

------insert into galib.c----------

```
Long int penalty (intvals[ ]){
Long int TA, TB, T1, T2, T3, P1, P2, P3, P4, P5, P;
TA = vals[0] + vals[1] + vals[2];
TB = vals[3] + vals[4] + vals[5];
T1 = vals[0] + vals[3];
T2 = vals[1] + vals[4];
T3 = vals[2] + vals[5];
/*********************************************
*if (TA – 550) > 0 (the amount passed from constraint) *
* the value returned will be: penalty rate * this difference *
*else it will be 0, for it respected the constraint *
*********************************************/
P1 = ((TA – 550) > 0)? (RATE1*(TA -550)):0;
P2 = ((TB – 700) > 0)? (RATE2*(TB -700)):0;
P3 = ((T1 – 390) > 0)? (RATE3*(T1 -390)):0;
P4 = ((T2 – 460) > 0)? (RATE4*(T2-460)):0;
P5 = ((T3 – 370) > 0)? (RATE5*(TA -370)):0;
P = P1 + P2 + P3 + P4 + P5;
Return P;
}
```

The RATEn constraints are defined in galib.h:

-----insert into galib.h----------

```
/* the rate for penalizing for eaxch constraint unsatisfied */
#define RATE1 20
#define RATE2 20
#define RATE3 20
#define RATE4 20
#define RATE5 20
```

----------cut here ----------

b) Determining Fitness

Now that we have the objective function value and the penalties for not respecting the constraints we can say how good an individual is, or in GA, its fitness:

-------insert into galib.c----------

```
Void evaluate(crom*cromo){
Intvals[6];
Long intobjt;
Int P;
/* let's get the ral values of all variables */
Decode((*cromo),vals);
/* now let's apply the objective function to it */
objt=object(vals);
/* its penalty for not respecting our constraints*/
P = penalty(vals);
/*****************************************
    *let's guarantee that the fitness will be always *
    * positive, because we don't want the objective *
    *Function to give us negative results *
    *(or it wouldn't be profit, it'd be outlay) *
```


ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

```
* so if its penalty is too much much, let's zero *
* the result and taking it out of further selection *
(it will have a probability of 0% to be chosen) *
**************************************?
Cromo->fitness = (objt – P>0)?(objt-P):0;
}
```

c) Crossover and Mutation:

Now let's think about two basics operators in GA: the crossover and mutation. For the crossover, as the individual is represented by an unique binary string in its crom, let's just use its simplest form, single point crossover: the procedure is very simple, had chosen one random point, let's slice each individual into two parts and exchange those parts to generate two new individuals, the function will get as a parameter two crom structs representing the parents and two pointers to chromosome structs representing the two children.

```
--------insert into galib.c---------
Void crossover(crom par1, crom par2, crom par3, *son1,
crom*son2){
Int point,I;
/*Let's get a point between 0 and BITS*/
Point = random(0,BITS);
/**************************************
*the first part we copy the genes*
*from parent 1 to child 1 *
*and from parent 2 to child 2 *
*******************************/
For(i=0;i<point;++i){
Son1->cromo[i] = par1.cromo[i];
Son2->cromo[i] = par2.cromo[i];
}
/*****************************
*here we'll "cross" the chromosome, *
   *now copying from parent 2 child 1 *
    * and from parent 1 to child 2 *
    ***********************************
        For(i=0;i<point;++i){
        Son1->cromo[i] = par2.cromo[i];
        Son2->cromo[i] = par1.cromo[i];
        }
    }
  ----------cut here-----------
```

And for the mutation we select a single point and then exchange its bit:

```
---- insert into galib.c---
    Void mutation(crom*cromo){
    intpoint,i;
    /*let's get a point between 0 and BITS*/
    Point = random(0,BITS);
    /*just invert the bit in the point chosen (bless the binary
    system*/
    Cromo->cromo[point] = !cromo->cromo[point];
      }
  ------------insert into
```

d) Selection and Dropping of Mutants and Surviving Chances:

Now let's work with the one function that we'll make use of to help select who will reproduce and later, who will stay alive to the next generation. This works by calculating the probability of an individual to be selected based on its fitness and the sum of all fitness of population. So first let's make a function to calculating this probability:

```
----------insert into galib.c----------------
    Void probability(crom*pop, intsize_pop){
    Int i;
    Double sum = 0.0, pro;
    Intpro_sum=0;
    /*let's calculate the sum of all fitness */
    For(i=0;i&1t;size_pop;++i){
    Sum +=pop[1].fitness;
  }
  /*******************
  *now for each one we decide *
  *its fitness by the sum of *
  *all and multiply by 100      *
  *resulting in its percentage *
  *************************/
  For(i=0;i&1t;size_pop;++i){
  Prob = (double)(100*pop[1].fitness) / sum;
```


International Research Journal of Innovations in Engineering and Technology (IRJIET)
ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

```
Pop[i].prob = iround(prob);
/*just in case we want to see if we have
Really 100%(and rarely we do)*/
Prob_sum +=iround(prob);
}
}
```

The i round function uses an near-even algorithm for rounding numbers, so we can get something near 100% as the sum of all probabilities. You can see this function in the source code package.

e) Selecting Our Survivors:

This select function will get as parameters, the population to make the selection, the amount of individuals to select, and pointer for a cromstruct where the chosen ones will be stored, we also use a trick to slow down evolution when there's an individual much better than the others, we spin the roulette several times trying to select someone not selected yet. So here is the code:

```
----------insert into galib.c----------
Void select(crom *pop, intsize_pop, int selections, crom
*result){
intI,j,k,choice,theone,tries=0;
char *h_pop;
/*let's use this dynamic array to avoid choosing 2 same
individuals */
h_pop = (char*) malloc(sizeof(char)*size_pop);
For(i=0;i&1t;size_selections;++i){
tries = 0;
do {
j=0;
theone = 0;
/*0 to 100 percent */
Choice = random(1,100);
/*sum the probabilities until we get the percentage
randomly chosen */
While(theone&1t;choice && j &1t; size_pop)
Theone += pop[j++].prob;
/*get back to the chosen one */
--j;
/*******************************
*after the loop, j will store the *
*Value of the chosen one, but in *
* Case we have passed thru the limit… *
*******************************/
j=j%size_pop;
if(j&1t;0) j = 0;
/*******************************
*loop until  we chosen someone *
*not chosen before, or we have *
Tried more than 20 times *
*******************************/
} while(h_pop[j] &&tries++ &1t;20);
/*this one is now chosen */
h_pop[j]=1;
/*******************************
*do the copy dance *
*******************************
For(i=0;i&1t;BITS;++k){
Result[i].cromo[k] = pop[j].cromo[k];
/*******************************
*only the fitness will be copied *
*for the probability will be different *
*******************************/
Result[i].fitness = pop[j].fitness;
}
/*let's not waste memory */
Free(h_pop);
}
----------cut here----------
```

f) Achieving the Solution:

Well, all the parts of a simple GA system is ready to run, let's put all pieces together:

```
-------insert into galib.c----------
Void ga-solve(){
inti,j,k,gen;
intvals[6];
/*************************************
*declaration of the population, their children,        *
*an auxiliary variable and the best individual ever*
*************************************************/
```


ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

```
Crompop[POP}, childfren{CHILDREN},
temp[POP+CHILDREN], best;
/*****************************************
*in the beginning the best individual *
*is the 0 one, so… no need to set *
*any more parameters since we'll just *
*use the fitness for comparison    *
*****************************************/
Best.fitness = 0;
Best_turn.fitness = 0;
/*Let'sgerenate the initial population at random
(or pseudo-random as you wish) */
For(i=0;i&1t;POP;++i){
For(i=0;i&1t;BITS;++j){
pop[i].cromo[j] = random(0,2);
}
}
gen = 0;
/*repeat the reproduction steps until the max
*number of generations is reached */
While(gen++&1t;GENS){
/*first, let's see how good they are…*/
for(i=0;i&1t;POP;++i){
evaluate(&pop[i]);
/*…and what is the chance of each one */
Probability(pop,POP);
/*****************************************
*and two by two, my human zoo shall reproduce *
*until the number desired ofm children is reached *
*****************************************/
For(i=0;i&1t;CHILDREN;i+=2){
Select(pop,POP,2,temp);
Crossover(temp[0],temp[1],&children[i],&children[i+1]);
}
/*DO our children are good enough? */
For(i=0;i&1t;CHILDREN;++i){
Evaluate(&children[i]);
/*Let's do some mutation experiments
to our population buhuahuahua */
i= random(0,POP);
mutation(&pop[i]);
/* Let's see how good our mutant is */
        evaluate(&pop[i]);
/*let's gather all together the oldies and the newbies */
/*first the oldies */
for(i=0;i&1t;POP;++i){
    temp[i].fitness = pop[i].fitness;
for(j=0;j&1t;BITS;++j){
    temp[i].cromo[j] = pop[i].cromo[j];
}
/*let's elect the best of this generation */
for(i=1;k=0;i&1t;POP+CHILDREN;++i)
if(temp[i].fitness &gt; temp[k].fitness){
decode(temp[i],vals);
/*we are looking for someone who
respect the constraints */
if(!penalty(vals))
            k = I;
        }
Decode(temp[k],vals);
/*let's store it for later */
if(temp[k].fitness &gt; best.fitness&& !penalty(vals)){
    for(i=0;i&1t;BITS;++i)
        best,cromo[i]=temp[k].cromo[i];
    best.fitness = temp[k].fitness;
}
/*Now the penalty can begin, who will lve and who shall
die? */
Probabilikty(temp,POP+CHILDREN);
Select(temp,POP+CHILDTREN,POP,pop);
}
/*End of this Generation */
/*And the best individual ever was… */
printf ("The best ever fitness was: %d\n"best.fitness);
Decode(best,vals);
printf("Values:%d%d%d%d%d%d\n",
vals[0], vals[1],vals[2],vals[3],vals[4],vals[5]);
printf("Objective Function:%d\n",object(vals));
printf("Penalty: %d\n\n"9");
```


ISSN (online): 2581-3048
Volume 2, Issue 5, pp 1-11, July-2018

V.CONCLUSION

As you might have noticed, the solution almost never converges to the global optimum solution, but it gets close. You should also notice that the GA system has a complexity of O(1), since no matter the kind of problem, the time will be almost constant, and what defines the time is the number of generations that it will loop for, and not the size of the problem.

REFERENCES

[1] Holland, J. H. "Adaption in Natural and Artificial Systems" *Cambridge, MA: MIT* Press, 1975.
[2] Van, N. R. "Handbook of Genetic Algorithms*"*. New York:, 1991.
[3] Srinivas, M and Patnaik, L. M. "Genetic algorithms: A survey," *Computer*, pp. 17-26, June 1994.
[4] Braca J., Bramel J., Posner B. and Simchi-Levi D. "A computerized approach to the New York City school bus routing problem. *IIE Trans* vol. 29, pp. 693-702. 1997.
[5] Li L. and Fu Z. "The school bus routing problem", A case Study. *J Opl Res, Soc* vol. 53, pp. 552-558, 2002.
[6] Letchford N., Lysgaard J. and Eglese R. "A branch-and-cut algorithm for the capacitated open vehicle at routine problem" 2002.
[7] Sariklis, D. and Powell, S. "A heuristic method for the open vehicle routine problem. *J Opl Res Soc* Vol. 51, pp. 564-573, 2000.
[8] Tarantilis, C.D., Ioannou, G., Kiranoudis C.T. and Prastacos G.P. "solving the open vehicle routing problem via a single parameter metaheuristic algorithm", *J Opl Res Soc* vol.56, pp. 588-596, 2005.

AUTHOR'SBIOGRAPHIES

AhmaduDauda Ally, ICT Center, Federal University, Wukari
Qualification: Bsc(Hons) Computer Science, Msc, Information Technology, MBA

Ahmadu Sandra Asabe, Department of Computer Science. ModibboAdama University of Technology, Yola.
Qualification: Bsc(Hons), Msc, PhD Computer Science

Philemon Uten Emmoh is a Senior System Analyst/ Programmer Engr. with the ICT Dept. of Fed. University Wukari, Taraba State, Nigeria.
Qualifications: B.Sc. & M.Sc in Computer Science

ISSN (online): 2581-3048

Volume 2, Issue 5, pp 1-11, July-2018

Citation of this article:

Ahmadu A. D, Ahmadu S. Asabe PhD, Phelimmon Emmon, "Creating a Model for Genetic Mutants Using Integer Programming Techniques", *International Research Journal of Innovations in Engineering and Technology (IRJIET)*, Volume 2, Issue 5, pp 1-11, July 2018.

****

