A GENETIC-OPTIMIZATION C CODE FOR BOUNDED VARIABLE INTEGER PROGRAMS

Atidel Ben Hadj-Alouane
and
James C. Bean
Department of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 94-3

February 1994

A Genetic/Optimization C Code for Bounded Variable Integer Programs*

Atidel Ben Hadj-Alouane
James C. Bean

Department of Industrial and Operations Engineering
University of Michigan, Ann Arbor, MI 48109-2117

Version 1.1

February 7, 1994

*This work was supported in part by the National Science Foundation under Grant DDM-9018515 and DDM-9308432 to the University of Michigan.

Contents

1 Introduction

This report is a documentation and a user manual for the C code, **genip**, developed for solving the following bounded variable integer program (P):

$$\begin{array}{lll} min & \sum_{i=1}^{n} c_i x_i & \\ s.to & \sum_{i=1}^{n} a_{ki} x_i \geq b_k, & k = 1, \ldots, r \\ & 0 \leq x_i \leq u_i, & integer \end{array}$$

The code implements a genetic algorithm applied to the following nonlinear relaxation, (PP_λ), of the above problem:

$$\begin{array}{ll} \min & \sum_{i=1}^{n} c_i x_i + \sum_{k=1}^{r} \lambda_k [\min(0, (\sum_{i=1}^{n} a_{ki} x_i - b_k)]^2 \\ & 0 \leq x_i \leq u_i, \text{ integer} \end{array}$$

where the vector $\lambda \in \Re^r$ is the penalty parameter, which value is periodically adjusted. Computational tests show that **genip** is robust and at least as efficient as OSL's branch-and-bound, especially for large ($\tilde{2}50$ or more variables) dual degenerate problems. See [1] and [2] for more information.

2 User Manual

2.1 Code Location

The source code, **genip.c**, an executable program, **genip**, and other related files are placed in the anonymous FTP machine called *freebie.engin.umich.edu* under the directory

/pub/misc/genopt

To retrieve these files, the user should use the ftp program on a machine with tcp/ip connectivity to the Internet. One should connect to *freebie* with the ftp command, and log in with account name *anonymous* and his/her electronic mail address as the password.

In the following ftp session, the file **genip.c** is retrieved.

```
dice% ftp freebie.engin.umich.edu
Connected to knob2.engin.umich.edu.
220 knob2.engin.umich.edu FTP server (Version 5.60) ready.
Name (freebie.engin.umich.edu:user_id): anonymous
331 Guest login ok, send ident as password.
Password:  <<< Your Email Address Here >>>
230 Guest login ok, access restrictions apply.
ftp> cd /pub/misc/genop
ftp> get genip.c
200 PORT command successful.
150 Opening ASCII mode data connection for genip.c (11763 bytes).
226 Transfer complete.
12253 bytes received in 0.7619 seconds (15.7 Kbytes/s)
ftp> bye
221 Goodbye.
dice%
```

2.2 Compiling and Running

The code is compiled under the AIX Operating System (on IBM Risc/6000) using the xlc (or also cc) compiler. To retrieve To produce an executable program using the optimizer option (O), type

cc genip.c -O -ogenip

The executable **genip** is invoked by the command

genip ⟨*input-file-name*⟩

If ⟨*input-file-name*⟩ is not specified, **genip** prompts the user to enter an input file name. The output is printed to a file called **genip.out**. This file includes summary of some input parameters and solutions.

The input file format is given below

$$
\begin{array}{ccccc}
n & r & & & \\
c_1 & c_2 & \cdots & c_n & \\
u_1 & u_2 & \cdots & u_n & \\
a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\
a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
a_{r1} & a_{r2} & \cdots & a_{rn} & b_r
\end{array}
$$

2.3 Setting Parameters

The maximum problem and population sizes are given. They are used for dimensioning arrays.

MAXn Maximum number of variables (columns)

MAXr Maximum number of constraints (rows)

MAXN Maximum number of solutions in a population

They are specified at the beginning of the code with **#define** statements (see section 3.1). If these limits are violated, **genip** terminates with a descriptive error message. In this case, the user should increase the limit(s) and recompile the code.

The following are more problem-specific parameters, which can be set interactively upon starting a **genip** session.

LB Lower bound for the problem (if available)

TOL If **LB** is available, **genip** stops when a solution within **TOL**% of **LB** is found

MAXG Maximum number of generations created

SEED Seed for the pseudo-random number generator

Many other parameters are already preset in the code.

Nc	Number of copied (or replicated) solutions
Nm	Number of immigrated solutions (mutations)
Nf	Frequency for altering the value of λ
Beta1	Value, β_1, by which λ is multiplied

Although the algorithm is considered fairly robust to these parameters, there may still be instances where adjusting them is beneficial.

2.4 An Example

Consider an integer program with 10 variables and 3 general constraints. The vectors c, u and b, and the matrix A are given below.

$$(c_i) = \begin{pmatrix} 83 & 83 & 124 & 226 & 226 & 277 & 277 & 390 & 390 & 495 \end{pmatrix}$$

$$(u_i) = \begin{pmatrix} 4 & 4 & 10 & 6 & 6 & 8 & 8 & 7 & 7 & 8 \end{pmatrix}$$

$$(a_{ij}) = \begin{pmatrix} 152 & 152 & 314 & 347 & 347 & 626 & 626 & 780 & 780 & 823 \\ 401 & 401 & 520 & 607 & 607 & 7867 & 786 & 918 & 918 & 932 \\ 389 & 389 & 582 & 675 & 675 & 759 & 759 & 867 & 867 & 870 \end{pmatrix} \quad (b_j) = \begin{pmatrix} 18020 \\ 24288 \\ 24137 \end{pmatrix}$$

The corresponding input data file, here called **ip1**, is set up for this problem as follows:

```
10 3
83 83 124 226 226 277 277 390 390 495
4 4 10 6 6 8 8 7 7 8
152 152 314 347 347 626 626 780 780 823 18020
401 401 520 607 607 786 786 918 918 932 24288
389 389 582 675 675 759 759 867 867 870 24137
```

Note that the format of the data file may be such as each data element is in a separate line as long the elements are in the right order.

Before starting to run a problem instance, it is good to find an estimate of its lower bound (the simplest bound may be the solution value of the linear programming relaxation of the instance). If such bound is available, the user should provide it to the code upon starting a genetic algorithm session. In this case, the code uses this value as a stopping criterion, and to initialize the penalty parameter λ. However, if no lower bound is available, a large number of generations should be provided (at least 5000 for large problems). The following are two example sessions for solving the above problem instance. Note that although the first session uses a lower bound, a generation count limit must be provided so that the program does not run indefinitely. Also, a zero tolerance may be provided.

Session 1.

```
dice% genip ip1
** Lower Bound (if not available <RETURN>)? --> 8203
** Tolerance (in %)? --> 0.5
** Max Number of Generations? --> 5000
** Population size (3<=N<=2000)? --> 20
** Seed? --> 2
** Print intermediate solutions? (y/n) --> n
dice%
```

Session 2.

```
dice% genip ip1
** Lower Bound (if not available <RETURN>)? -->
** Number of Generations? --> 5000
** Population size (3<=N<=2000)? --> 20
** Seed? --> 1
** Print intermediate solutions? (y/n) --> n
```

The last line indicates that the user has the option of observing detailed intermediate solutions (evolution of the entire population) on the screen as the program is running. This is sometimes helpful in checking whether parameters are assigned appropriate values (e.g. population size, number of replicated solutions, ...). However, the program becomes slower especially that the user has to hit the ⟨RETURN⟩ key between generations. The following is a sample intermediate output from Session 1. (only the first 10 solutions are shown here).

```
------------------------------ Generation #  2 ----------------------------
```

Sol.#	Solution										Value	Penalty
1	1	3	2	4	5	5	8	6	0	1	9050.200	400.000
2	0	3	6	6	1	4	4	6	4	3	10176.000	0.000
3	0	0	0	2	5	0	8	6	0	8	11101.527	2007053.000
4	0	4	2	6	6	2	2	4	1	8	10310.000	0.000
5	1	3	10	6	0	1	5	5	3	7	11175.000	0.000
6	3	4	7	3	1	5	2	2	1	0	75790.818	140657629.000
7	3	3	4	6	0	4	4	5	6	4	10836.000	0.000
8	1	2	0	4	5	6	7	4	3	0	9907.407	2586814.000
9	3	4	7	3	5	7	2	2	4	8	12050.000	0.000
10	0	3	2	6	5	2	8	6	0	8	12053.000	0.000

```
<RETURN>
```

The output file **genip.out** includes problem data and parameters summary, the final solution and some intermediate solutions. An example output file from Session 1 is given below.

********** A Genetic Algorithm For Bounded Variable **********

********** Integer Programs **********

Version1.1 31 Jan 1994

---PROBLEM DATA----------

10 columns by 3 rows (Input File: ip1)

---PARAMETERS----------

Lower Bound = 8203.000000

Tolerance = 0.500000

Max. Generations = 5000

Population Size = 20

Seed = 2

---STATISTICS----------

Generation #	Value	Penalty
0	14218.000	0.000
20	8401.912	53824.000
40	8215.402	88804.000
60	8435.156	289.000
80	8388.400	3600.000
100	8388.400	3600.000

---BEST SOLUTION FOUND----------

1 3 9 0 0 8 8 2 4 0

Value = 8220.000000

No of generations = 118

CPU Time = 0.590000 seconds

This output file shows some intermediate solutions by printing the value (including original objective, cx, and the penalty $p_\lambda(x)$) and the penalty of the top solution every 20 generations. Notice that the best solution found is feasible to the original problem, i.e it has a zero penalty. If no feasible solution is found after **MAXG** generations, the following message is printed at the end of the file **genip.out**.

```
---BEST SOLUTION FOUND-----------------------------------------------
      No feasible solution after 5000 generations!!
      Adjust GA parameters and rerun genip
      CPU Time          = 200.010000 seconds
 ------------------------------------------------------------------
```

This means that the user should increase the generation count limit (**MAXG**) or change the population size or the seed of the random number generator. If this still fails to give feasible solutions, then other parameters such as the initial value of λ or β_1 should be adjusted. However, in this case, the code must be recompiled.

3 Code Documentation

3.1 Basic Definitions and Data Structure

```
#include <stdio.h>
#include <math.h>
#include <time.h>

/* Max Problem size, modify to solve larger problems */
#define MAXn 1001 /* Max no. of variables (or columns) */
#define MAXr 501 /* Max no. of constraints (or rows) */

/* Max Population size */
#define MAXN 2001

/* Problem data */
int n, r,        /* No. of variables and constraints */
    u[MAXn];     /* Upper bounds on the variables */

float c[MAXn],         /* Objective function coefficients */
      a[MAXr][MAXn],   /* Constraint coefficients */
      b[MAXr];         /* RHS values of the constraints */

/* Solution representation */

struct sol{
       int x[MAXn];        /* Variables (x vector) */
       double value;       /* Solution value including penalty */
       double pen;};       /* Solution penalty */
```

```
struct sol best,          /* Best solution found */
           pop[MAXN],     /* Current population of solution */
           oldpop[MAXN]; /* Previous population of solutions */

/* GA parameters */

int N,        /* Population size */
    Nc, Nm,   /* No. of copied and immigrated solutions */
    count,    /* Index of current generation */
    ctr,      /* Finds sequences of top solns. with zero (nonzero) penalties
    pen1,     /* Zero (one) if top soln. of current generation has zero (nonzero) penalty */
    pen2,     /* Same as pen1 fpr previous generation */
    bestg=0   /* Generation no. where best solution is found */
    lbav      /* Indicates whether a lower bound is available */

float lam,          /* Penalty multiplier λ */
      Beta1, LB, TOL;

char do_out='n';
FILE *outfile;
double c1, c2;      /* Find CPU time */
```

A given generation consists of **N** solutions (or individuals). It is declared above as an array of solutions. Each individual is defined as a structure which members are the solution itself, its value with respect to the relaxed problem (i.e. objective function of original problem + penalty value) and the penalty value which indicates the feasibility of the solution.

The best solution found is also declared as a structure except that its penalty value must be zero. The variable **bestg** is the same as the generation count, **count**, when the program stops, in the case where a lower bound is available. In case where only the

generation count limit is provided as a stopping criterion, **bestg** gives the generation index at which the best solution was actually found.

3.2 Input Function

The function **readin** reads problem data, GA parameters and also prints the header of the output file.

```
void readin(fname) /* Read input data and GA parameters */

char fname[];
{
FILE *infile;
int i,j;
char buf[500];


if (fname[0] == '?')
{
  printf("\n\n** Enter input file name --> ");
  gets(fname);
}
if ((infile = fopen(fname, "r")) == NULL)
{
  printf( "ERROR: Cannot open input file: %s\n", fname);
  exit(1);
}
else
{
printf("\n\n** Lower Bound (if not available <RETURN>)? --> ");
gets(buf);
lbav=0;
```

```
if (sscanf(buf, "%f", &LB) == 1)
{
   lbav=1;        /* lower bound available */
   printf("\n\n** Tolerance (in %%)?  --> ");
   gets(buf);
   sscanf(buf,"%f", &TOL);
   printf("\ n \ n** Max Number of Generations?  --> ");
}

else printf("\n\n** Number of Generations? --> ");
gets(buf);
sscanf(buf, "%d", &MAXG);
printf("\n\n** Population size (3<= N <=%d)? --> ", MAXN-1);
gets(buf);
sscanf(buf, "%d", &N);
if (N>MAXN)
{
  printf("ERROR: Population size limit exceeded.");
  printf(" (Data = %d, Limit %d)\n", N, MAXN-1);
  exit(1);
}
else if (N<3)
{
  printf("ERROR: Population size < 3!!\n");
  exit(1);
}
printf("\n\n** Seed? --> ");
gets(buf);
sscanf(buf,"%d", &SEED);
```

```
printf("\n\n** Print intermediate solutions? (y/n) --> ");
gets(buf);
sscanf(buf,"%c", &do_out);
do_out = tolower(do_out);

     /* Read problem data */

fscanf(infile,"%d%d\n",&n, &r);
if (n>MAXn)
{
  printf("ERROR: Column limit exceeded.");
  printf(" (Data = %d, Limit %d)\n", n, MAXn-1);
  exit(1);
}
if (r>=MAXr)
{
  printf("ERROR: row limit exceeded.");
  printf(" (Data = %d, Limit %d)\n", r, MAXr-1);
  exit(1);
}
for (i=1;i<=n;i++) fscanf(infile, "%f", &c[i]);
for (i=1;i<=n;i++) fscanf(infile, "%d", &u[i]);
for (j=1; j<=r; j++)
{
  for (i=1; i<=n; i++) fscanf(infile, "%f", &a[j][i]);
  fscanf(infile, "%f", &b[j]);
}
if (n<=100) Nf = 50;
else Nf = n/2;
```

```
/* Print output file header*/

fprintf(outfile,"\n\n\n\t********** A Genetic Algorithm For Bounded Vari
able **********\n");
fprintf(outfile, "\n\t\t********** Integer Programs **********\n");
fprintf(outfile,"\n\t\t\t    Version1.1 31 Jan 1994");
fprintf(outfile, "\n\n   ---PROBLEM DATA----------------------------------
-----------------\n\n");
fprintf(outfile,"                      %4d columns by %3d rows  (Input File:
 %s) \n\n", n, r, fname);
fprintf(outfile, "   ---PARAMETERS-----------------------------------------
-------------\n\n");
if (lbav)
fprintf(outfile, "\tLower Bound      = %f\n\tTolerance        = %f\n", LB,
TOL);
fprintf(outfile, "\tMax. Generations = %d\n", MAXG);
fprintf(outfile, "\tPopulation Size  = %d\n\tSeed             = %d\n\n", N,
SEED);
fprintf(outfile,"   ---STATISTICS-----------------------------------------
------------\n\n\n");
fprintf(outfile, "\tGeneration #\t\tValue\t\t   Penalty\n");
fclose(infile);
} /* End readin */
```

3.3 Genetic Algorithm Functions

There is one main GA function called **genetic** and three other functions referred to as **setup**, **repro** and **eval**. The functions **genetic** first calls **setup** which initializes some parameters and randomly generates an initial population of solutions. Then, it calls iter-

atively **repro** which, given an initial generation, reproduces a new one using the process of elitist reproduction, the Bernoulli crossover and the immigration operators described in [2]. The function **eval** evaluates a solution given its index in the current population, which is defined by the array **pop**. The function **eval** is called by **setup** and **repro**.

```
void eval(m)      /* Evaluate the mth solution in the population */

int m;
{
int i,j,k,l;
double f, fq, px=0.0;

     /* Calculate objective value (without penalty term) */

f = 0.0;
for (i=1;i<=n;i++) f += c[i]*pop[m].x[i];

     /* Calculate penalty value */

for (l=1;l<=r;l++)
{
  fq = 0.0;
  for (i=1;i<=n;i++) fq += a[l][i]*pop[m].x[i];
  if (fq < b[l]) px += (b[l]-fq)*(b[l]-fq);
}

     /* Summarize */

pop[m].pen = px;
pop[m].value = f + lam*px;
} /* End eval */
```

The function **setup** initializes the parameter λ using a lower bound of the problem instance, if provided by the user. Notice that throughout the algorithm, all elements of the vector λ are equal.

```
void setup() /* Set up initial generation and initialize some parameters */
{
int i,j, cnt;
double z;
void eval(), printout();

best.value = 1.0e+31;
Nc = .05*N; if (Nc==0) Nc=1;
Nm = .05*N; if (Nm==0) Nm=1;
lam = 0.0;
Beta1 = 8;
count = 0;
ctr = 1;
z = 0.0;
srand(SEED);

/* Create random solution */

for (i=1;i<=N;i++)
{
  for (j=1;j<=n;j++) pop[i].x[j] = pick(u[j]);
  eval(i);
}
```

```
/* Estimate the value of lam */

if (lbav);
{
  for (i=1;i<=N;i++) if (pop[i].pen != 0.0)
  {
    z += (LB-pop[i].value)/pop[i].pen;
    cnt++;
  }
  if (cnt!=0)  lam = z/cnt;
}
if (cnt==0 || lam <= 1.0E-6) lam=0.0005;
for (i=1;i<=N;i++) pop[i].value += lam*pop[i].pen;
pen1 = !(pop[1].pen==0.0);
if (do_out=='y')
{
  printf("\n\n\n\t********** A Genetic Algorithm For Bounded Variable **
  *********\n");
  printf("\n\t\t************** Integer Programs **************\n");
  printf("\n\t\t\t\t    Version1.1 31 Jan 1994\n\n");

  if (n>20) printf("\n\n\t STRINGS ARE TOO LONG! THEY WILL NOT BE PRINTED.
  \n\n");
  printout();
}
fprintf(outfile, "      \t%5d\t\t%16.3f   %16.3f\n",count, pop[1].value,
pop[1].pen);
} /* End setup */
```

The function **repro** first sorts the solutions of the current generation in increasing order of their objective function values using the *bubble sort*. Second, it copies the **Nc** top solutions to the new generation. If necessary, the value of **lam** is adjusted. Then, the Bernoulli crossover and the immigration operators are performed.

```
void repro() /* Reproduce solutions for the next generation */

{
void eval();
int pick();
int i,j,ii[MAXN],sorted,k,s,p,stop;
float ff[MAXN],z;

/* Initialize */

for (i=1;i<=N;i++)
{
  for(j=1;j<=n;j++) oldpop[i].x[j] = pop[i].x[j];
  oldpop[i].value = pop[i].value;
  oldpop[i].pen = pop[i].pen;
  ii[i] = i;
  ff[i] = oldpop[i].value;
}

/* Bubble sort population */

sorted = 0;
```

```
while(1-sorted)
{
  sorted = 1;
  for(i=1;i<N;i++) if (ff[i] > ff[i+1])
  {
    z = ff[i];j = ii[i];
    ff[i] = ff[i+1]; ii[i] = ii[i+1];
    ff[i+1] = z; ii[i+1] = j;
    sorted = 0;
  }
} /* End while loop */

/* Copy top "Nc" solutions */

for(i=1;i<=Nc;i++)  pop[i] = oldpop[ii[i]];

/* If needed, decrease/increase the value of lam */

if (ctr == Nf)
{
  if (pen1==0)
  {
    lam /= (0.7*Beta1); /* 0.7*mult is Beta2 */
    for(i=1;i<=Nc;i++) pop[i].value += lam*(1-0.7*Beta1)*pop[i].pen;
  }
  else
  {
    lam *= Beta1;
    for(i=1;i<=Nc;i++) pop[i].value += (lam-lam/Beta1)*pop[i].pen;
  }
} /* End if */
```

```
/* Mate (N-Nc-Nm) random pairs */

i = Nc;
stop = N - Nm;
while (i < stop)
{
  i++;

  /* pick two mates from the population */

  j = pick(N);
  k = pick(N);

  /* Bernoulli crossover with prob=0.7 */

  for(s=1;s<=n;s++)
  {
    p = pick(100);
    if (p<=70)
    {
      pop[i].x[s]=oldpop[j].x[s];
      pop[i+1].x[s]=oldpop[k].x[s];
    }
    else
    {
      pop[i].x[s]=oldpop[k].x[s];
      pop[i+1].x[s]=oldpop[j].x[s];
    }
  } /* End for */
```

```
      /* evaluate offspring and keep the best */

   eval(i); eval(i+1);
   if (pop[i+1].value < pop[i].value)
   {
     for (s=1;s<=n;s++) pop[i].x[s] = pop[i+1].x[s];
     pop[i].value = pop[i+1].value;
     pop[i].pen = pop[i+1].pen;
   }
} /* End while */

/* Create mutations ("Nm" random solutions) */

for (i=1;i<=Nm;i++)
{
  k=N-Nm+i;
  for (j=1;j<=n;j++) pop[k].x[j] = pick(u[j]);
  eval(k);
}
} /* End repro */
```

The function **genetic** runs while the stopping criterion is not met (i.e **cont**=1). It also checks whether there are **Nf** successive generations each with the first top solution feasible (or infeasible) to the original problem.

```
void genetic() /* Run genetic algorithm: repeat the reproduction process
                  a fixed number of times (use MAXG) or until a lower
                  bound is reached within a percentage given by TOL */

{
void setup(), repro();
int test, cont;

setup();
do
{
  count++;
  repro();
  if (do_out=='y') printout();
  if ((count%20) == 0)
  {
    fprintf(outfile, "      \t%5d\t\t%16.3f   %16.3f\n",count, pop[1].value,
    pop[1].pen);
    fflush(outfile);
  }

  /* Check if the top solution, pop[1], had successively zero
     (or nonzero) penalty values in the last Nf generation */

  pen2 = !(pop[1].pen==0.0);
  if (!(pen1^pen2))
  {
    if (ctr < Nf) ctr++; else ctr = 1;
  }
```

```
  else
  {
   pen1 = pen2; ctr = 1;
  }

/*    Update best solution found so far */

  if (pop[1].pen == 0.0 && pop[1].value<best.value)
  {
    best = pop[1];
    bestg= count;
  }
  cont = (count < MAXG);

/*    Check whether lower bound (if available) is reached */

  if (lbav)
  {
    test = (((pop[1].value-LB)*100/fabs(LB)) > TOL);
    cont = ((test || pop[1].pen!=0.0) && cont);
  }
} while (cont); /* End do loop */
} /* End genetic */
```

3.4 Main Function and Utilities

The function **printout** prints intermediate solutions (entire generations) to the screen.

```
void printout() /* Print intermediate solutions to the screen */

{
int i, j;
char buf[500];

if (n<=20)
{
   printf("\n ------------------------------ Generation #%3d --------
   ---------------------\n\n\n", count);
   printf("  Sol.#\t\t\tSolution\t\tValue\t\tPenalty\n\n");
}
else
{
   printf("\n        ------------------------ Generation #%3d ---------
   -----------\n\n\n", count);
   printf("\t Sol.#\t\t\tValue\t\t\tPenalty\n\n");
}

for (i=1;i<=N;i++)
{
  printf("  %4d\t ", i);
  if (n<=20) for (j=1;j<=n;j++) printf("%3d",pop[i].x[j]);
  printf("\t%16.3f   %16.3f \n", pop[i].value, pop[i].pen);
}
printf("\n <RETURN>\n");
```

```
gets(buf);
} /* End printout */
```

The function **pick** randomly generates integer numbers using the standard function **rand**.

```
int pick(n) /* Generate a pseudo-random integer number between 0 and n */
int n;
{
float p;
int p1;

p = rand();
p1 = p*(n+1)/32767.0;
return p1;
} /* End pick */
```

The **main** function calls **readin** and **genetic**, and prints the final solution.

```
void main(argc, argv)
int argc;
char *argv[];
{
char fname[30];
void readin(), genetic();
int i,j;

outfile = fopen("genip.out", "w");
if (argc == 2) strcpy(fname, *++argv);
else fname[0] = '?';
```

```
readin(fname);
c1 = (double) clock();
genetic();
c2 = (double) clock();
fprintf(outfile, "\n   ---BEST SOLUTION FOUND----------------------
------------- -----------\n\n");
if (bestg==0)  /* No feasible solution is found */
{
  fprintf(outfile, "\n\tNo feasible solution after %5d generations!!\n",
  count);
  fprintf(outfile, "\n\tAdjust GA parameters and rerun genip\n\n");
}
else
{
  for (j=1;j<=n;j++) fprintf(outfile, "\t\t%d \n",best.x[j]);
  fprintf(outfile, "\n\tValue             = %f\n", best.value);
  fprintf(outfile, "\tNo of generations = %d\n", bestg);
}
fprintf(outfile, "\tCPU Time          = %f seconds\n\n", (c2 - c1)/(1.0*
CLOCKS_PER_SEC));
fprintf(outfile, "   ------------------------------------------------
-----------------\n");
fclose(outfile);
} /* End main */
```

References

[1] J. C. Bean and A. Ben Hadj-Alouane. A dual genetic algorithm for bounded integer programs. Technical Report 92-50, Department of Industrial and Operations Engineering, University of Michigan, 1992. To appear in *RAIRO Recherche Opérationelle.* (invited submission to special issue on GAs and OR).

[2] A. Ben Hadj-Alouane and J. C. Bean. A genetic algorithm for the multiple–choice integer program. Technical Report 92-50, Department of Industrial and Operations Engineering, University of Michigan, 1992.