# Eliminating memory leak in MGA-C program

**Alexander L. Kosorukoff**

IlliGAL Report No. 98015
December 1998

Illinois Genetic Algorithms Laboratory
Department of General Engineering
University of Illinois at Urbana-Champaign
117 Transportation Building
104 South Mathews Avenue
Urbana, IL 61801

# Eliminating memory leak in MGA-C program

Alexander L. Kosorukoff
(sandy@cyberservices.com)

December 1998

## 1 Introduction: Memory leak in MGA-C program

This section overviews the problem of excessive memory consumption in MGA-C program [1] that is caused by several bugs in the sources.

Frequent messages about insufficient memory made me suspect a memory leak in the program sources. I traced the memory allocation, printing the amount of allocated memory at the end of each era. The internal logic of algorithm suggests that this amount should not increase significantly during the run. The result of the trace has shown the opposite. Some dynamic memory wasn't freed after the end of each era via function `freeallmemory()`. The allocated memory accumulated during the run wasting memory resources. After assigning each memory block a marker corresponding to the place of its creator in the source code, I found the places responsible for the lost block creation and the places where these blocks get lost.

One such place is in the file `operators.c`, containing the functions associated with the messy GA operators.

## 2 Bugs found and fixed

In this section the specific causes of the memory leak are discussed and a possible fix is presented. In addition some source modification is suggested that lead to more efficient usage of program stack.

### 2.1 The bug in operators.c

The code `cut_site = rnd(1,(*chr1).chromlen)` in function `cut()` sometimes makes variable `cut_site` equal to `(*chr1).chromlen`. In this case the length of the second part of the cutted chromosome equals zero. This type of chromosome is unsupported neither in the rest of the `cut()`, nor in the `cut_and_splice()`. The possibility of empty chromosome is reflected in the following fragment of `cut_and_splice()` function:

```
        /*  if chr3 is not empty, next element in the stack is chr3  */
        if (chr3flag && chr3.firstgene != NIL)
                push_stack(&chrom_stack,chr3);

        /*  top most element is chr1  */
        push_stack(&chrom_stack,chr1);

        /*  check elements in the stack and splice  */
        do  {
        }   while (test_splice());
```

Empty chromosomes are neither pushed into stack nor freed. Eventually they get lost, and the memory allocated stays such until the end of the program.

There are two ways to correct this bug. Either we should add a deletion of zero-length chromosomes in `cut_and_splice()`, or we should eliminate the possibility of their creation. The second way appear to be simpler and more natural.

The following is suggested modification of file `operators.c` in unified `.diff` format:

```
--- MGA-C/operators.c  Fri Dec 12 16:56:32 1996
+++ MGA-CR/operators.c Fri Jan 31 07:15:28 1997
@@ -160,29 +160,29 @@
 struct INDIVIDUAL chr1,chr2;
 /*  cut and splice operators  */
 {
-       int chr3flag;
+        int chr3flag,chr4flag;
        struct INDIVIDUAL chr3;
        struct INDIVIDUAL chr4;

        /*  if first chromosome needs to be cut  */
-       if (flip(cut_prob * chr1.chromlen))
+        chr4flag=flip(cut_prob * (chr1.chromlen-1));
+        if (chr4flag)
        {
                cut(&chr1,&chr4);
-               /* if chr4 is not empty first element in chrom_stack is chr4 */
-               if (chr4.firstgene != NIL)
                        push_stack(&chrom_stack,chr4);
        }

        chr3.firstgene = NIL;
        /*  if second chromosome needs to be cut  */
```

```
-       if (chr3flag = flip(cut_prob * chr2.chromlen))
+        chr3flag = flip(cut_prob * (chr2.chromlen-1));
+        if (chr3flag)
                cut(&chr2,&chr3);

        /*  next element in the chrom_stack is chr2  */
        push_stack(&chrom_stack,chr2);

        /*  if chr3 is not empty, next element in the stack is chr3  */
-       if (chr3flag && chr3.firstgene != NIL)
+        if (chr3flag)
                push_stack(&chrom_stack,chr3);

        /*  top most element is chr1  */
@@ -230,7 +230,7 @@
        temp = (*chr1).firstgene;

        /* find out cut site  */
-       cut_site = rnd(1,(*chr1).chromlen);
+        cut_site = rnd(1,((*chr1).chromlen-1));

        /*  cut chr1 at cut_site  */
        for (i = 1; i < cut_site; i++)
```

The main modification is the last:

```
        /* find out cut site  */
-       cut_site = rnd(1,(*chr1).chromlen);
+        cut_site = rnd(1,((*chr1).chromlen-1));
```

It assumes that a chunk of a cutted chromosome should be at least 1 bit. Since the number of available cut points is decreased by one we should correct the probability of cut operation in expressions for `chr3flag` and `chr4flag`.

### 2.2 The bugs in utility.c

The second bug is in `freenewpop()` function. When freeing chromosomes of a new population `newpop` the buggy code was used, although the initial program contained a recursive function `delete_chrom()` written for this purpose correctly, which is fully functional. It is more natural to use it here.

I suspect the author had a reason not to use his correct function `delete_chrom()`, in order to avoid messages about stack overflow when deleting lengthy chromosomes. In this case the nonrecursive code for `delete_chrom()` would be advantageous.

If we choose nonrecursive code, instead of

```
void delete_chrom(chrom)
struct GENE *chrom;
/*  delete a chromosome starting at pointer temp  */
{
        if (chrom != NIL)
        {
                delete_chrom(chrom->nextgene);
                free(chrom);
        }
}
```

we can use the following variant

```
void delete_chrom(chrom)
struct GENE *chrom;
/*  delete a chromosome nonrecursively (AK) */
{ struct GENE *p;

  while (chrom != NIL)
    {  p = chrom;
       chrom = chrom->nextgene;
       free(p);
    }

}
```

It is a bit longer but more reliable.

Another bug impairing the memory usage is in the implementation of the function `reallocate_memory()` in the the same source file. When shrinking the arrays of population this function must release the discarded genomes, but it doesn't.

The suggested modification is in `.diff` file below. It assumes that variable `popsize` contain the current size of population. If the new size of population is smaller than discarded population members are released.

```
--- MGA-GCC/utility.c   Fri Dec 12 16:56:32 1996
+++ MGA-GCCR/utility.c  Fri Jan 31 21:16:56 1997
@@ -143,29 +143,34 @@
 void freenewpop()
 {
        register int i;
-       struct GENE *p, *p2;

        for (i = 0; i < popsize; i++) {
-               p = newpop[i].firstgene;
```

```
-               while (p != NIL) {
-                       p2 = p;
-                       free(p);
-                       p = p2->nextgene;
-               }
+                delete_chrom(newpop[i].firstgene);
                free(newpop[i].fullchrom);
                free(newpop[i].fullgene);
        }
 }

 void reallocate_memory(size)
 /*  reallocates storage for arrays, used only in primordial phase  */
 long size;
 {
+        register int i;
+
+        for (i = size+1; i < popsize; i++)
+        { /* delete chromosome, fullchrom, and fullgenes */
+                delete_chrom(oldpop[i].firstgene);
+                delete_chrom(newpop[i].firstgene);
+                free(oldpop[i].fullchrom);
+                free(newpop[i].fullchrom);
+                free(oldpop[i].fullgene);
+                free(newpop[i].fullgene);
+        }
         newpop = (struct INDIVIDUAL *)realloc(newpop,size * \
                                         sizeof(struct INDIVIDUAL));
         oldpop = (struct INDIVIDUAL *)realloc(oldpop,size * \
                                         sizeof(struct INDIVIDUAL));
         shuffle = (int *)realloc(shuffle,size*sizeof(int *));
 }
```

This modification of `reallocate_memory()` function assumes that `popsize` variable value correspond to the current size of population, at least when calling `reallocate_memory()`. In original sources it doesn't.

## 2.3 Other files modification

Two files `generate.c` and `initial.c` were slightly modified to make variable `popsize` correspond to the actual population size when calling `reallocate_memory()` function.

```
diff -u mga-c/generate.c mga-c-current/generate.c
--- mga-c/generate.c    Mon Dec 28 21:02:50 1998
+++ mga-c-current/generate.c    Mon Dec 28 21:09:48 1998
@@ -6,6 +6,7 @@
  developed : 1991

     author : Kalyanmoy Deb
+  modified : Alex Kosorukoff
 ================================================*/

 #include "mga.ext"
@@ -64,8 +65,8 @@

                count++;
        }
-       popsize = newpopsize;
        reallocate_memory(popsize);
+       popsize = newpopsize;   /* relocated after reallocate_memory() by AK */

        TRACE("   Primordial exited\n");
 }

diff -u mga-c/initial.c mga-c-current/initial.c
--- mga-c/initial.c     Mon Dec 28 21:06:42 1998
+++ mga-c-current/initial.c     Mon Dec 28 21:06:20 1998
@@ -6,6 +6,7 @@
   developed : 1991

      author : Kalyanmoy Deb
+   modified : Alex Kosorukoff (AK)
 ==========================================================*/

 #include "mga.ext"
@@ -165,6 +166,7 @@
        }
        /*  storage allocation for */
        /*  population array  */
+       popsize=total_size;     /* AK */
        if(!(newpop=(struct INDIVIDUAL *)malloc(total_size * \
                                sizeof(struct INDIVIDUAL))))
        {
```

## 3 Results

The memory trace after these fixes shown that all allocated dynamic memory now is returned to the pool of available memory at the end of each run, there is no leak. Previously wasted memory is now available for more complex tasks.

## References

[1] Kalyanmoy Deb and David E. Goldberg. mGA in C: A messy genetic algorithm in C. Technical Report 91008, Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, Urbana, IL 61801, September 1991.

# 4 The code in C

This section contains the modified source files of MGA-C program. Other sources are available at the IlliGAL Techreport [1].

## 4.1 operators.c

```
/*=================================================
       file : operators.c

    purpose : messy GA operators

  developed : 1991

     author : Kalyanmoy Deb
     bugfix : Alex Kosorukoff
=================================================*/

#include "mga.ext"


INDV_ID select_pop()
/*  determine which selection operator  */
{
if (gen < init_select_gen)
return (init_select());
else
{
if (thresholdingflag)
return(thresholding_select());
else
return(normal_select());
}
}


INDV_ID select(first, second)
INDV_ID first, second;
/*  selects the better of first and second  */
{
if (oldpop[first].fitness == oldpop[second].fitness) {
if (tiebreakingflag)
return((oldpop[first].chromlen < \
oldpop[second].chromlen) ? first : second);
```

```
else
return(flip(0.5) ? first : second);
}
else if (oldpop[first].fitness > oldpop[second].fitness)
{
return((maximizationflag) ? first : second);
}
else
{
return((maximizationflag) ? second : first);
}
}


INDV_ID init_select()
/*  tournament selection operator without shuffling  */
{
INDV_ID first,second;

if (pick >= popsize-1)
pick = 0;
first = pick;
second = pick + 1;
pick += 2;
return (select(first, second));
}


INDV_ID normal_select()
/*  tournament selection operator with shuffling  */
{
INDV_ID first,second;

if (pick >= popsize-1)
{
pick = 0;
shuffle_pop();
}
first = shuffle[pick];
second = shuffle[pick+1];
pick += 2;
return (select(first, second));
}
```

```
INDV_ID thresholding_select()
/*  tournament selection operator with thresholding  */
{
register int i,j,k;
INDV_ID first, second, fittest, secondid;
int num1, num2, count, temp, threshold;
BOOLEAN flag;
unsigned *rawgn1, *rawgn2;

if (pick >= popsize-1)
{
pick = 0;
shuffle_pop();
}
/*  choose the first individual, default is first  */
fittest = first = shuffle[pick];
rawgn1 = oldpop[first].fullgene;
num1 = oldpop[first].genelen;

for (k=1,flag=0; (k<=shuffle_num && !flag); k++)
{
/*  choose the second individual  */
secondid = pick + k;
/*  if required, scroll to the begining of the array */
if (k >= (popsize - pick))
secondid = k - (popsize - pick);
second = shuffle[secondid];
rawgn2 = oldpop[second].fullgene;
num2 =  oldpop[second].genelen;

/*  count the number of matched genes  */
count = matched_genes(rawgn1, rawgn2);

/*  calculate the threshold value  */
threshold = (num1 * num2);
threshold /= problem_length;

if (count > threshold)
/*  if count > threshold, found a match, perform selection */
{
flag = 1;
temp = shuffle[secondid];
shuffle[secondid] = shuffle[pick + 1];
```

```
shuffle[pick + 1] = temp;
fittest = select(first, second);
pick++;
}
}
pick++;
return (fittest);
}

int matched_genes(rawgn1,rawgn2)
unsigned *rawgn1;
unsigned *rawgn2;
/*  count the number of matched genes  */
{
register int i, j;
int num = 0;
unsigned rawgn, mask = 1;

for (i = 0; i < bytesize; i++)
{
/*  get the template of common genes */
rawgn = (rawgn1[i] & rawgn2[i]);

/*  count the number of 1's in the template  */
for (j = 0; j < bytelimit[i]; j++)
{
if (((rawgn >> j) & mask) == 1)
num++;
}
}
return(num);
}


void cut_and_splice(chr1,chr2)
struct INDIVIDUAL chr1,chr2;
/*  cut and splice operators  */
{
int chr3flag, chr4flag;       /* chr4flag appended -- AK */
struct INDIVIDUAL chr3;
struct INDIVIDUAL chr4;

/*  if first chromosome needs to be cut  */
chr4flag = flip(cut_prob * (chr1.chromlen - 1));   /* AK */
```

```
if (chr4flag)
{
cut(&chr1,&chr4);
/* if chr4 is not empty first element in chrom_stack is chr4 */
if (chr4.firstgene != NIL)
push_stack(&chrom_stack,chr4);
}

chr3.firstgene = NIL;
/*  if second chromosome needs to be cut  */
        chr3flag = flip(cut_prob * (chr2.chromlen - 1));   /* AK */
if (chr3flag)
cut(&chr2,&chr3);

/*  next element in the chrom_stack is chr2  */
push_stack(&chrom_stack,chr2);

/*  if chr3 is not empty, next element in the stack is chr3  */
if (chr3flag) /* modified by AK */
/* the zero-length chromosome check  */
/* is no longer needed here */
push_stack(&chrom_stack,chr3);

/*  top most element is chr1  */
push_stack(&chrom_stack,chr1);

/*  check elements in the stack and splice  */
do  {
}   while (test_splice());
}


BOOLEAN test_splice()
/*  splice two members of the stack, send 1 if there is a member left,
    0 otherwise  */
{
struct INDIVIDUAL chr1,chr2;

/*  get the first member  */
if (pop_stack(&chrom_stack,&chr1))
{
if (flip(splice_prob))
{
/*  check for another member in the stack  */
```

```
/*  splice if yes  */
if (pop_stack(&chrom_stack,&chr2))
splice(&chr1,&chr2);
}
push_stack(&newchrom_stack,chr1);
return(1);
}
else
return(0);
}


void cut(chr1,chr2)
struct INDIVIDUAL *chr1,*chr2;
/*   cut chr1 into two pieces: chr1, chr2   */
{
register int i;
struct GENE *temp;
int cut_site;
char msg[PAGEWIDTH];

temp = (*chr1).firstgene;

/* find out cut site  */
cut_site = rnd(1,(*chr1).chromlen-1);   /* modified -- AK */

/*  cut chr1 at cut_site  */
for (i = 1; i < cut_site; i++)
temp = temp->nextgene;

/*   assign other members of chr1 and chr2  */
(*chr2).firstgene = temp->nextgene;
(*chr2).lastgene = (*chr1).lastgene;
(*chr1).lastgene = temp;
temp->nextgene = NIL;
(*chr2).chromlen = (*chr1).chromlen - cut_site;
(*chr1).chromlen = cut_site;

/*  allocate memory for chr2 fullchrom and fullgene  */
if(!((*chr2).fullchrom = (unsigned *)malloc(bytesize*sizeof(unsigned)))) {
sprintf(msg, "Insufficient memory for fullchrom\n");
print_error(msg);
}
if(!((*chr2).fullgene = (unsigned *)malloc(bytesize*sizeof(unsigned)))) {
```

```
sprintf(msg, "Insufficient memory for fullgene\n");
print_error(msg);
}
}


void splice(chr1,chr2)
struct INDIVIDUAL *chr1,*chr2;
/*   splice chr1 and chr2 to chr1  */
{
/*  chr2 is appended behind chr1  */
(*chr1).lastgene->nextgene = (*chr2).firstgene;
(*chr1).lastgene = (*chr2).lastgene;
(*chr1).chromlen = (*chr1).chromlen + (*chr2).chromlen;

/*  destroy chr2 fullchrom and fullgene */
free((*chr2).fullchrom);
free((*chr2).fullgene);
}


void mutation(chr)
struct INDIVIDUAL *chr;
/*  if mutation probabalities are greater than zero, perform mutation  */
{
register int i;
  struct GENE *temp;

if ((allelic_mut_prob > 0.0) || (genic_mut_prob > 0.0)) {
temp = (*chr).firstgene;
while (temp != NIL)
{
if (flip(allelic_mut_prob))
/*   perform allelic mutation  */
{
if (temp->allele == 1)
temp->allele = 0;
else
temp->allele = 1;
allelicmutation++;
}
if (flip(genic_mut_prob))
/*  perform genic mutation  */
{
```

```
temp->genenumber = \
genic_mutation(temp->genenumber);
genicmutation++;
}
temp = temp->nextgene;
}
}
}


GENE_ID genic_mutation(num)
GENE_ID num;
/*  change the genenumber to other  */
{
GENE_ID n;

do {
n = rnd(0, problem_length-1);
}  while (n == num);
return(n);
}
```

## 4.2 utility.c

```
/*====================================================

    file    : utility.c

    purpose :  utility funations used in other files

  developed : 1991

     author : Kalyanmoy Deb
     bugfix : Alex Kosorukoff

=====================================================*/

#include "mga.ext"


void gen_file(file, filevar, name)
char *file, *filevar, *name;
/*  name input-output files */
{
char newname[30];
int limit = 30;

sprintf(filevar, "%s", name);
/* find out if a different file name is asked for */
printf("\n%s file name (%s) = ",file,filevar);
get_string(stdin,newname,limit);
if (strlen(newname) != 0)
strcpy(filevar,newname);
}


void get_string(txt, s, n)
FILE *txt;
char *s;
int n;
/*  read a string */
{
register int c;
while ((c = getc(txt)) != EOF && --n > 0) {
if (c != '\n')
*s++ = c;
```

```
else
break;
}
*s = '\0';
}


void check_input_file(filename)
char *filename;
/*  check an input file */
{
FILE *fp, *fopen();
char msg[80];

if ((fp = fopen(filename,"r")) == NIL)
{
sprintf(msg, "Input file: can not open %s",filename);
print_error(msg);
exit(1);
}
fclose(fp);
}


void scratch_file(filename)
char *filename;
/* scratch a file */
{
FILE *fp, *fopen();

fp = fopen(filename,"w");
fclose(fp);
}


void print_error(msg)
char *msg;
/*  send error message and exit  */
{
fprintf(stderr, "\n***** %s\n", msg);
exit(1);
}
```

```
void sortnum(num,list)
int num;
int *list;
/*  sort an array of integers in ascending order  */
{
register int i,j;
int temp;

for (i = 0; i < num; i++)
{
for (j=i+1; j<num; j++)
{
if (list[i] > list[j])
{
temp = list[i];
list[i] = list[j];
list[j] = temp;
}
}
}
}


void freeallmemory()
/*  free all memory stored  */
{
register int i;

for (i = 0; i < popsize; i++)
{ /* delete chromosome, fullchrom, and fullgenes */
delete_chrom(oldpop[i].firstgene);
delete_chrom(newpop[i].firstgene);
free(oldpop[i].fullchrom);
free(newpop[i].fullchrom);
free(oldpop[i].fullgene);
free(newpop[i].fullgene);
}
free(newpop);
free(oldpop);

free(shuffle);
delete_chrom(best_indv.firstgene);
free(best_indv.fullchrom);
free(best_indv.fullgene);
```

```
}


void delete_chrom(chrom)
struct GENE *chrom;
/*  delete a chromosome nonrecursively (AK)  */
{ struct GENE *p;

while (chrom != NIL)
{
p = chrom;
chrom = chrom->nextgene;
free(p);
}
}


void freenewpop()
{
register int i;
struct GENE *p, *p2;

for (i = 0; i < popsize; i++) {
delete_chrom(newpop[i].firstgene);  /* reduced -- AK */
free(newpop[i].fullchrom);
free(newpop[i].fullgene);
}
}

void reallocate_memory(size)
/*  reallocates storage for arrays, used only in primordial phase  */
long size;
{
register int i;

for (i = size+1; i < popsize; i++)
{ /* delete chromosome, fullchrom, and fullgenes */
delete_chrom(oldpop[i].firstgene);
delete_chrom(newpop[i].firstgene);
free(oldpop[i].fullchrom);
free(newpop[i].fullchrom);
free(oldpop[i].fullgene);
free(newpop[i].fullgene);
        }
```

```
newpop = (struct INDIVIDUAL *)realloc(newpop,size * \
sizeof(struct INDIVIDUAL));
oldpop = (struct INDIVIDUAL *)realloc(oldpop,size * \
sizeof(struct INDIVIDUAL));
shuffle = (int *)realloc(shuffle,size*sizeof(int *));
}


int round(num)
double num;
/*  round off the number  */
{
int a1;
double a2;

a1 = num;
a2 = num - a1;
return ((a2 > 0.5) ? a1+1 : a1);
}


int maximum(num, list)
int num;
int *list;
/*  calculates the maximum value in an array  */
{
register int i;
int max;

for (i = 1, max = list[0]; i < num; i++) {
if (list[i] > max)
max = list[i];
}
return(max);
}


void repchar (out, ch, repcount)
FILE *out;
char *ch;
int repcount;
/* Repeatedly write a character to an output device */
{
register int j;
```

```
for (j = 1; j <= repcount; j++)
fprintf(out, "%s", ch);
}

void page(out)
FILE *out;
/* Issue form feed to device or file */
{
repchar(out,"\f",1);
}


void skip(out, skipcount)
FILE *out;
int skipcount;
/* Skip skipcount lines on device out */
{
repchar(out,"\n",skipcount);
}


void skip_space(out, skipcount)
FILE *out;
int skipcount;
/*  skip skipcount characters  */
{
repchar(out," ",skipcount);
}


long fact(n)
int n;
/*   calculates factorial of n  */
{
register int i;
long prod;

for (i = 1, prod = 1; i <= n; i++)
prod *= i;
return(prod);
}
```

```
long choose(n,k)
int n;
int k;
/*   calculates n choose k   */
{
register int i;
long prod;

for (i = n, prod = 1; i > n-k; i--)
prod *= i;
return(prod / fact(k));
}

void reset_list(list,nlist)
ALLELES *list;
int nlist;
/*   resets the elements of list to zero  */
{
register int i;

for (i = 0; i < nlist; i++)
list[i] = 0;
}


long power(a,b)
int a;
int b;
/*  calculates a^b  */
{
register int i;
long prod;

for (i = 0, prod = 1; i < b; i++)
prod *= a;
return(prod);
}

BOOLEAN parity(num)
int num;
/* whether num is even */
{
BOOLEAN par;
```

```
if ((num % 2) == 0) par = 1;
else par = 0;
return(par);
}

int ones(chr,len)
int chr[];
int len;
/* count the number of 1's */
{
register int i;
int num = 0;

for (i = 0; i < len; i++)
if (chr[i] == 1)  num++;
return(num);
}

double decode(chr,len)
int chr[];
int len;
/* calculate the decoded value of a binary string */
{
register int i;
int powerof2;
double val = 0.0;

powerof2 = power(2,len-1);
for (i = 0; i < len; i++) {
if (chr[i] == 1)
val += powerof2;
powerof2 /= 2;
}
return(val);
}

double map(min, max, x, len)
double min, max, x;
int len;
{
int dec_max;

dec_max = power(2, len) - 1;
return (min + ((max - min) * x) / dec_max);
```

```
}
```