mGA in C: A Messy Genetic Algorithm in C

Kalyanmoy Deb and David E. Goldberg
Department of General Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801

IlliGAL Report No. 91008
September 1991

Illinois Genetic Algorithms Laboratory (IlliGAL)
Department of General Engineering
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue
Urbana, IL 61801

Contents

mGA in C: A Messy Genetic Algorithm in C

Kalyanmoy Deb and David E. Goldberg
Department of General Engineering
The University of Illinois at Urbana-Champaign
Urbana, IL 61801

1 Introduction to Messy Genetic Algorithms

This section[1] overviews traditional genetic algorithms and presents a brief description of a messy genetic algorithm. Genetic algorithms (GAs) are search procedures based on the mechanics of natural genetics and selection. Over the years, GAs have been successful in a wide variety of search, optimization, and machine learning problems in science, commerce, and engineering. Despite their empirical success, there have been some objections to their use. The most difficult of these objections is the so-called *linkage problem*. The linkage problem arises because of the coding of the problem parameters. To assure convergence to the global optima, strings in GAs must be coded so that short, highly fit allele combinations—*building blocks*—can combine to form optima. If the linkage between necessary allele combinations is too weak, in certain types of problems called *deceptive* problems (Goldberg, 1989b, 1989c), genetic algorithms will converge to suboptimal solutions. Unfortunately, the necessary linkage associated in a problem are not usually known. Furthermore, it has been shown elsewhere (Goldberg, Korb, & Deb, 1989) that the probability of obtaining a tight linkage in a random coding is very slim. Even though a number of reordering operators have been suggested to recode strings adaptively, these methods have not yet proved sufficiently powerful in empirical studies. Moreover, recent theoretical work (Goldberg & Bridges, 1990) has suggested that unary reordering operators are too slow to be any use in searching for tight linkage.

It is under this backdrop that a different genetic algorithm called a *messy genetic algorithm* (mGA) has been devised and tested (Deb, 1991; Goldberg, Deb, & Korb, 1990, 1991; Goldberg, Korb, & Deb, 1989). mGAs work by searching for tight building blocks and then combining them together to form the optima in a way that respects a version of the schema theorem. In a number of problems of bounded deception, mGAs have repeatedly found the global optima in a time that grows only as a polynomial function of the number of decision variables on a serial machine. The details of messy genetic algorithms are described elsewhere (Deb, 1991; Goldberg, Deb, & Korb, 1990, 1991; Goldberg, Korb, & Deb, 1989). Here, the fundamental differences between traditional and messy GAs are highlighted:

1. mGAs use variable-length strings that may be over- or underspecified with respect to the problem being solved;

2. mGAs use simple *cut* and *splice* operators in place of fixed-length crossover operators;

3. mGAs divide the evolutionary process into two phases: a primordial phase and a juxtapositional phase;

[1] Some portion of this section are excerpted from a paper entitled 'Don't Worry, Be Messy' (Goldberg, Deb, & Korb, 1991).

4. mGAs sometimes use competitive templates to emphasize salient building blocks.

Messy GAs use variable-length strings that may be under- or overspecified with respect to the problem being solved. For example, in the messy string `((2 1) (1 0) (2 0))`, the leftmost element specifies gene `2` and its allele value is `1`. In a three-bit problem, this chromosome is underspecified, because gene `3` is absent, and overspecified, because gene `2` appears twice. This flexibility in coding allows the use of simple genetic operators. But, this flexibility is achieved only at the cost of resolving the dual problem of under- and overspecification in a string. Simple first-come-first-served rules with a left-to-right scan are used to overcome the overspecification problem. Under this scheme, in the above example, the gene `2` takes an allele value `1`, since it appears first in a left-to-right scan. The underspecification problem is somewhat more complex, and its solution is achieved by the use of *competitive templates*. Under this scheme, a string specifying an allele at each gene position is chosen as a template and the unspecified genes in a string are borrowed from the template. In the above example, if the chosen template is `(0 0 0)`, the above string corresponds to the complete 3-bit string `(0 1 0)`, where the third string is taken from the template. How the template is actually selected is discussed in some detail later on.

Like traditional GAs, mGAs use selection and recombination operators on a population and create a new population. In mGA studies, a binary tournament selection method is used. In a binary tournament selection, two strings are chosen at random and the better string is selected. Since, messy strings are of variable length, the usual crossover operator can no longer be used. Two simple operators, cut and splice, are implemented for this purpose. *Cut* severs a string with specified probability $p_c = (\lambda - 1)p_\kappa$ that grows as the string length λ, and *splice* joins two strings together with fixed probability p_s. Together, cut and splice have no more potential disruption than simple single-point crossover and they have roughly the same juxtapositional power (Goldberg, Korb, & Deb, 1989). An analysis of the cut and splice operation (Deb, 1991) indicates that for small values of p_c, and large values of p_s, short strings are recombined almost fully, and large strings are processed as they are with a single-point crossover operator.

Unlike simple GAs, mGAs divide the genetic processing into two distinct phases: a primordial phase and a juxtapositional phase. In the primordial phase, the population is first initialized to contain all possible building blocks of a specified length, where the characteristic length is chosen to encompass possibly deceptive (misleading) building blocks. Thereafter, the proportion of good building blocks is enriched through a number of generations of reproduction without other genetic action. At the same time, the population size is usually reduced by halving the number of individuals in the population at specified intervals. With the proportion of good building blocks so enriched, the juxtapositional phase proceeds with a fixed population size and the invocation of reproduction, cut, splice, and other genetic operators. Cut and splice act to recombine the enriched proportions of building blocks. In empirical tests to date, mGAs have always found globally optimal strings. Furthermore, this convergence has been shown to occur in a time that grows no faster than a polynomial function of the number of decision variables.

This accomplishment was only made possible through the invention of a noise-free method of evaluating salient building blocks. This procedure, the method of *competitive templates*, solves the problem of underspecification alluded to earlier by filling in unnamed genes with a locally optimal structure. In this way, only salient building blocks obtain fitness values better than the template value and are thereby enriched during the primordial phase. Since the salient building blocks in a problem may not be known before hand, a *level-wise* mGA has been suggested (Goldberg, Deb, & Korb, 1990) and used (Deb, 1991). In a level-wise mGA, a number of *era*s are used in succession. In the first era, the population is initialized with strings of length one. Since order-one building blocks can not be misleading, a random template is used to fill up the underspecified strings. The best solution found at the end of the first era is then used as the template for the second era. In the

second era, the population is initialized with all possible strings of length two and the best solution found at the end of the first era is then used as the template for the second era, and so on. This idea of using locally optimal solution as the template is implemented in mGA in C.

Together, these four differences, messy strings, messy operators, a two-phase evolutionary process, and competitive templates, distinguish mGAs from their more traditional counterparts.

In traditional fixed-length GAs, strings all have the same set of genes and, though the allele values may differ, all head-to-head comparisons have some meaning, because each string may be viewed as a full solution to the problem at hand. In mGAs, structures need not have any genes in common, and competition between arbitrary substrings is likely to be a case of comparing apples to oranges. Moreover, there are no guarantees that string lengths are matched to building-block size, and this opens the possibility that parasitic bits may tag along with good building blocks, later deactivating other needed substructures. These problems both result because mGAs are trying to solve many subproblems explicitly (and simultaneously), thereby giving an mGA a more ecological flavor than a traditional GA. Two mechanisms have been adopted with empirical success: *genic thresholding* and *length-based tie-breaking*.

Genic thresholding requires that two strings share some threshold θ number of *genes* before they are permitted to compete in tournament selection. The expected number of genes in common between randomly chosen strings of length λ_1 and λ_2 is hypergeometrically distributed. Thus, the expected number in common is $\theta = \frac{\lambda_1 \lambda_2}{\ell}$, and this is the threshold value used in the mGA studies.

Tie-breaking is used to prevent parasitic bits—bits that agree with the competitive template, but currently serve no purpose other than acting as a placeholder—from tagging along with some low-order building block, thereafter preventing the expression of some other building block. An additional $k-1$ *null bits* (explicit placeholders) may be added to the functional genes, or all building blocks at the current level and less may be generated, but the key point is to prefer strings with least length (or least effective length in the case of null bits) when fitness ties occur between building blocks. In this way strings will be preferred that have the least amount of stray genetic material, thereby reducing the threat of blocked expression from parasitic bits. In mGA in C, the tie-breaking is implemented by adding all building blocks at the current level.

In the remainder of this report, important data structures and global variables are described. Brief descriptions of the functions used in the code are given. The input files required to execute the code are discussed. The installing and running procedures of mGA in C are then described. The output files created by the code are briefly discussed. Finally, an example problem is solved using mGA in C and all input and generated output files are presented. A source listing is also included at the end of the report. In this report, a bold lower case character is used to represent a variable and a bold upper case character is used to represent either a defined constant or a defined structure. A $*$ followed by a variable name is used to define an one-dimensional array and two $*$s followed by a variable name is used to define a two-dimensional array.

2 Data Structures and Global Variables

In messy GAs, a gene is represented by both its name and value. Since messy GAs allow variable-length chromosomes, a linked-list structure is used to represent a messy chromosome. The structure `GENE` in figure 1 contains the name of a gene, `genenumber`, its allele value, `allele`, and a pointer to the next gene, `nextgene`. This allows a chromosome to be represented fully by specifying only the address of the first gene. The terminal gene in a chromosome is set to `NIL` (a zero), marking the end of the chromosome. Besides the standard variable types in C, a number of integer types—`INDV_ID`, `GENE_ID`, `ALLELES`, and `BOOLEAN`—are defined but used in different contexts. An individual counter in a population is declared as type `INDV_ID`; a gene is declared as type `GENE_ID`; an allele is declared

```
struct GENE {                    /* a gene */
        GENE_ID      genenumber; /* gene number */
        ALLELES      allele : 1; /* allele value */
        struct GENE   *nextgene; /* pointer to the next gene */
};

struct INDIVIDUAL {              /* an individual */
        struct GENE  *firstgene; /* pointer to the first gene */
        struct GENE   *lastgene; /* pointer to the last gene */
        unsigned     *fullchrom; /* an array of complete chrom */
        unsigned      *fullgene; /* an array of complete genes */
        int            chromlen; /* chromosome length */
        int              genelen; /* expressed gene length */
        double           fitness; /* objective function value */
};
```

Figure 1: Data structures for a gene and an individual

as type `ALLELES`; and a binary variable is declared as type `BOOLEAN`.

An `INDIVIDUAL`'s chromosome is specified by the address of the first gene, `firstgene`. The address of the terminal gene is conveniently called `lastgene`. Since the chromosomes may have different lengths, they can be either under- or overspecified. The under- and overspecification problems are resolved as discussed before and an array, `fullallele`, is formed by specifying an allele at each gene position. Each allele is stored in a bit for efficient memory allocation, and bitwise operators are used to store and extract allelic information. The variable `chromlen` corresponds to the length of a chromosome. The variable `fitness` is an individual's objective function value. The variable `fullgene` is an array of binary variables specifying the presence or absence of each gene in a chromosome. Like `fullallele`, these information are also stored in bits. The variable `genelen` is the number of expressed genes in a chromosome. These two variables are included in the structure to eliminate their repeated evaluation during thresholding.

The constant parameters used in the code are outlined next. A typical value of a parameter is given, followed by a brief description of the parameter.

`NIL`	0	This parameter is defined to be zero. It is used primarily to mark the end of a chromosome.
`UNSINTSIZE`	32	This parameter defines the number of bits required to store a unsigned integer variable. The alleles are stored in bits (rather than in integers) for efficient memory allocation.
`MAX_SIZE`	500	This parameter defines the maximum allowable problem size (number of bits).
`MAX_BYTE`	16	This parameter defines the maximum number of bytes required to store `MAX_SIZE` alleles. If `MAX_SIZE` is not divisible by `UNSINTSIZE`, one extra byte is assigned to store the remaining alleles.

MAX_ORDER	10	This parameter defines the maximum allowable number of genes in a subfunction.
MAX_PARTITIONS	100	This parameter defines the maximum allowable number of different partitions to be investigated. mGA in C allows an user to investigate the growth of strings in a partition (a set of fixed genes in a string) with generation number. The analysis of the growth of competing strings in a partition provides a clearer understanding of the workings of a genetic algorithm.
MAX_PARTITIONSIZE	10	This parameter defines the maximum allowable size of a partition.
PAGEWIDTH	80	This parameter defines the maximum number of characters in a line for reading and writing purposes.

The global variables used in the code are described next. The name of the variable is followed by its type, which is followed by a brief description of the variable.

*oldpop	INDIVIDUAL	This array contains the old population. Each member in this array is of type INDIVIDUAL.
*newpop	INDIVIDUAL	This array contains the new population. Each member in this array is of type INDIVIDUAL.
best_indv	INDIVIDUAL	This variable keeps a record of the best individual found in a population.
max_era	integer	This variable specifies an upper limit on the level of mGA operation.
era	integer	This variable represents the current era or level of operation.
order	integer	The variable order is the size of the strings used to create an initial population.
problem_length	integer	This variable specifies the size of the problem under consideration.
bytesize	integer	This variable specifies the number of bytes required to store problem_length number of bits. If the variable problem_length is divisible by UNSINTSIZE, the variable bytesize has a value equal to problem_length/UNSINTSIZE; otherwise the variable bytesize has a value equal to (1+problem_length/UNSINTSIZE).
*bytelimit	integer	This array contains the number of allocated bits in each byte. Clearly, there are bytesize entries in this array.
*copies	integer	This array contains the number of duplicates of each individual in the initial population for each era.
popsize	integer	This variable specifies the size of the population at any generation.

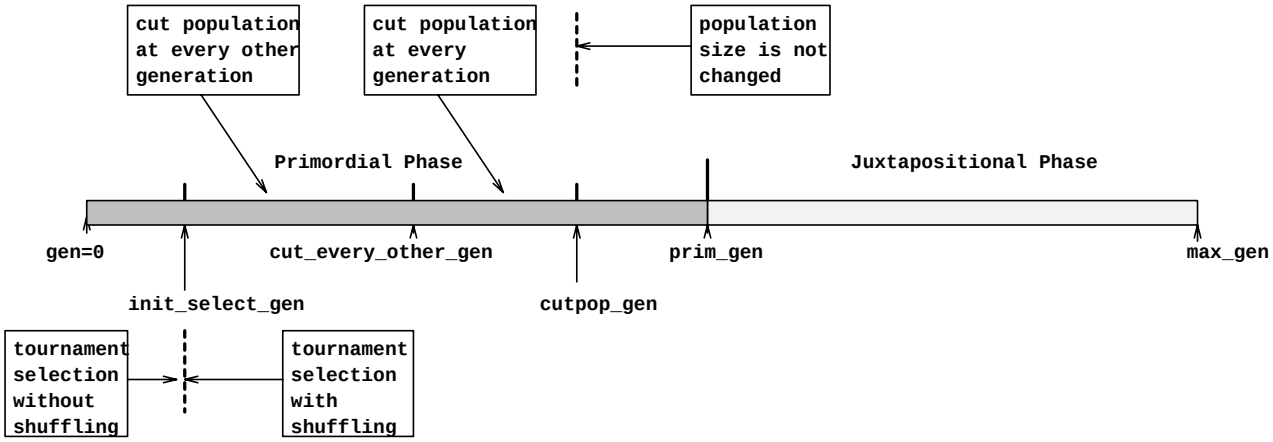

Figure 2: A typical population reduction schedule.

*template	integer	This array of size problem_length contains an allele value at each gene location. The template is used to fill up the underspecified genes in a chromosome.
gen	integer	This variable specifies the current generation counter.
max_gen	integer	This variable specifies the upper limit of the number of generations in an era of mGA operation.

The global variables associated with the mGA operators are described next. During the primordial phase, only selection operator is used to increasingly sample the highly fit strings—building blocks—in the population. As the building blocks are emphasized, the population size is simultaneously reduced.

In a binary tournament selection, two individuals are used for comparison and the better individual is selected. If performed without replacement, only half of the population will be filled when all population members are considered once. To fill up the other half, the population is shuffled and pairs of individuals are drawn again from the shuffled population. In mGA in C, three different variations of binary tournament selection are used. Initially, the tournament selection method is used without any shuffling of the population. In this method, when all population members are used up once, pairs of the individuals from the top of the same list are compared again. This selection operator is used to reduce the stochastic sampling error early on in the population. Thereafter, a tournament selection with shuffling or with shuffling and thresholding mechanism is used depending on the user-specified options.

The reduction of the population from its initial size to the juxtapositional population size is performed in a systematic way. Depending on the problem parameters, the population may be reduced at every generation, or at every other generation, or a combination of both. Figure 2 illustrates a typical schedule of population reduction with generation number.

During the juxtapositional phase, both selection and recombination operators are used to combine the enriched building blocks together to form the optimal solution.

prim_gen	integer	This variable specifies the duration of the primordial phase.

`init_select_gen`	`integer`	The tournament selection without shuffling of the population is used for first few generations. This variable specifies the duration of the tournament selection without any shuffling of the population.
`cutpop_gen`	`integer`	This variable specifies the duration of the population reduction in the primordial phase.
`cut_every_other_gen`	`integer`	This variable specifies the duration for population reduction at every other generation. Beyond this generation, the population is reduced at every generation until `cutpop_gen` generations.
`juxtpopsize`	`integer`	The population size in the juxtapositional phase is stored in this variable. Usually, the population size in the juxtapositional phase is kept constant.
`*shuffle`	`INDV_ID`	This variable contains a shuffled array of the old population members. Two consecutive members from this array are chosen for selection.
`pick`	`INDV_ID`	The variable `pick` is the counter of the population member in the shuffled array to be picked for selection.
`cut_prob`	`double`	This variable specifies the bitwise cut probability to be used in the simulation. To assure proper building block processing, a small value of `cut_prob` is suggested. A value close to $1/(2 * \texttt{problem\_length})$ is used in mGA studies.
`splice_prob`	`double`	This variable specifies the splice probability to be used in the simulation. A high value of `splice_prob` is used in mGA studies.
`allelic_mut_prob`	`double`	The probability of allelic mutation is stored in this variable. If mutated, a `1` is changed to a `0`, and vice versa.
`genic_mut_prob`	`double`	The probability of genic mutation is stored in this variable. If mutated, one gene is changed to any other gene at random.
`prop_bestindv`	`double`	This variable specifies the proportion of the best individual at any generation. The proportion of the best individual at the initial population is used to calculate the duration of the primordial phase. (see `setup_popsize()` for details)
`thres`	`integer`	This variable specifies the *threshold* number, θ, between two individuals. For two individuals with `genelen` equal to λ_1 and λ_2, the threshold number is calculated as $\lceil \frac{\lambda_1 \lambda_2}{\ell} \rceil$, where ℓ is the problem size (Deb, 1991; Goldberg, Deb, and Korb, 1990).

`shuffle_num`	`integer`	This variable specifies the size of a pool of individuals used in thresholding. For every individual in the population, another individual with θ number of common genes between them is searched from a pool of individuals of size, `shuffle_number`, for comparison. If one is found, they are compared, and the better one is selected; otherwise the former individual is selected. In a problem of size ℓ, the `shuffle number` of size ℓ is used in mGA studies (Deb, 1991; Goldberg, Deb, and Korb, 1990).
`avgstrlen`	`double`	The population average chromosome length is stored in this variable.

A structure `STACKTYPE` is defined for two stacks used to store partial strings after cut and splice operations. The members can only be entered and extracted from the top of the stack. The structure is shown in the following:

```
struct STACKTYPE {              /* stack for cut and splice operation */
        struct INDIVIDUAL genefirst;   /* an individual */
        struct STACKTYPE   *nextmem;   /* pointer to the next individual */
};
typedef struct STACKTYPE *STACKPTR;    /* pointer to the structure STACKTYPE */
```

The pointer to the structure `STACKTYPE` is defined as another structure `STACKPTR`. The following two global variables are declared type `STACKPTR`.

`chrom_stack`	`STACKPTR`	This stack contains the strings after cut operation. Two consecutive strings from the top of this stack are picked for splice operation.
`newchrom_stack`	`STACKPTR`	This stack contains the strings after splice operation. The strings from this stack placed into the new population.

The global variables associated with the objective function are described next. In mGA in C, a number of nonoverlapping subfunctions are added together to form the overall objective function. Each subfunction may be described by either specifying all possible allele combinations with their function values (referred as a `table`), or specifying an explicit code for the function evaluation (referred as a `function`). Subfunctions of type `table` and `function` may both be used in a problem. A scale factor for each subfunction may also be assigned so that each subfunction has a nonuniform contribution to the overall objective function value.

`numsubfunc`	`integer`	This variable specifies the number of subfunctions defining the problem.
`*table_id`	`int`	This array specifies the identification number of each `table` or `function` defining a subfunction.
`*taborfunc`	`BOOLEAN`	This array contains binary variables of a `1` or a `0`, specifying whether a subfunction is a `table` or a `function`.
`*str_length`	`integer`	This variable specifies the size of each `table` or `function`.
`*scale`	`float`	This array specifies the scale factor for each subfunction.

**genesets	GENE_ID	This variable contains an array of genes specifying each subfunction.
**chromfitness	double	This variable contains the function value corresponding to each string in a table.
avgfitness	double	This variable specifies the population average objective function value.
minfitness	double	This variable specifies the population minimum objective function value. In the case of a minimization problem, the individual with the minimum fitness is the best individual.
maxfitness	double	This variable specifies the population maximum objective function value. In the case of a maximization problem, the individual with the maximum fitness is the best individual.
templatefitness	double	This variable specifies the objective function value of the template.
function_evaluations	long	This variable keeps a record of the cumulative number of function evaluations at any instant.

mGA in C allows an user to investigate the growth of competing strings in a partition. The global variables associated with the user-defined partitions are described in the following.

numpartition	integer	This variable specifies the number of user-specified partitions.
*partition_len	integer	This array specifies the size of each specified partition.
**partition_genes	integer	This variable specifies an array of genes specifying each partition.

mGA in C allows an user to print a record of the population members at specified generations. The global variables associated with the user-defined generations for population records are described in the following.

*sortpopstatgen integer		This array specifies the generation numbers at which a population record is desired. The generation numbers are arranged in ascending order in this array.
countpopstatgen	integer	This variable keeps a track of the number of generations already considered.
nextpopstatgen	integer	This variable specifies the next specified generation number for printing.

mGA in C allows an user to create a file for plotting purpose. This file contains a number of population statistics versus the cumulative generation number.

gencount	integer	This variable specifies the cumulative generation number since the beginning of the mGA operation.

A number of flags are defined to control the messy GA operation. These variables are of type `BOOLEAN` and can take a value of either a `1` or a `0`.

`cutpopflag`	`BOOLEAN`	This flag is set to `1`, if the population size needs to be reduced at any generation during the primordial phase. This flag is set by the code, depending on other user-defined parameters.
`levelmgaflag`	`BOOLEAN`	The flag `levelmgaflag` is set to `1`, if the mGA needs to be run in successive levels. This flag is set by the user.
`maximizationflag`	`BOOLEAN`	If the maximum function value of the given problem is desired, this flag is set to `1`. This flag is set by the user.
`popprintflag`	`BOOLEAN`	This flag is set to `1`, if a record of all population members at any generation is desired. This flag is set by the user.
`partitionflag`	`BOOLEAN`	If partition investigation is desired, this flag is set to `1`. This flag is set by the user.
`plotstatflag`	`BOOLEAN`	This flag is set to `1`, if a plotting file is desired. This flag is set by the user.
`stopmgaflag`	`BOOLEAN`	This flag is set to `1`, if the current era needs to be terminated because of sufficient convergence or attainment to a state with no likely improvement. This flag is set by the code, depending on the convergence of the population at any generation.
`thresholdingflag`	`BOOLEAN`	If thresholding needs to be invoked, this flag is set to `1`. This flag is set by the user.
`tiebreakingflag`	`BOOLEAN`	If tie-breaking needs to be invoked, this flag is set to `1`. This flag is set by the user.
`traceflag`	`BOOLEAN`	This flag is set to `1`, if a trace of the messy GA operation is desired. This flag is set by the user.
`r_initpop_flag`	`BOOLEAN`	The initial population size required to include all building blocks of order k is $2^k \binom{\ell}{k}$, where ℓ is the problem size. Since underspecified strings are evaluated by filling the unspecified genes from a template, a number of strings in this population decodes to the same ℓ-bit string. Thus, this size can be reduced to $\binom{\ell}{k}$, if these duplicate strings are not included in the initial population. If this reduction in the initial population size is desired, this flag is set to `1`. This flag is set by the user.
`extrapopflag`	`BOOLEAN`	This flag is set to `1`, if extra population members are desired in the initial population. This flag is set by the user.

A number of variables are defined to store the names of various input-output files. A file name can be specified with a maximum of thirty characters.

`Inputfilename[30]`	`char`	This variable stores the name of an input file with all important global parameters. The default file name is `parameters`. (see section 4 for details)
`Erafilename[30]`	`char`	This variable stores the name of an input file with all era-dependent global parameters. The default file name is `era`. (see section 4 for details)
`Objfilename[30]`	`char`	This variable stores the name of an input file that contains objective function information. The default file name is `subfunc`. (see section 4 for details)
`Templatefilename[30]`	`char`	This variable stores the name of an input file that contains a user-specified template. The default file name is `template`. (see section 4 for details)
`Partitionfilename[30]`	`char`	This variable stores the name of an input file that contains the user-specified partitions for investigation. The default file name is `partitions`. (see section 4 for details)
`Poprinfilename[30]`	`char`	This variable stores the name of the input file that contains the generation numbers for a population record. (see section 4 for details)
`Outputfilename[30]`	`char`	This variable stores the name of an output file that contains an echo of the important global parameters and a record of the population statistics at each generation. The default file name is `output`. (see section 7 for details)
`Extrafilename[30]`	`char`	This variable stores the name of the input file that contains the set of genes for additional population members to be included in the initial population. The default file name is `extra`. (see section 7 for details)
`Plotfilename[30]`	`char`	This variable stores the name of an output file that contains the population statistics for plotting purpose. The default file name is `plot`. (see section 7 for details)
`Poproutfilename[30]`	`char`	This variable stores the name of an output file that contains a record of all population members at user-specified generations. The default file name is `poprecout`. (see section 7 for details)
`Partoutfilename[30]`	`char`	This variable stores the name of an output file containing the number of user-specified partition members at each generation. The default file name is `partout`. (see section 7 for details)

The global variables associated with the random number generator are described next.

`randomseed`	`double`	This variable specifies the user-specified random seed (a number between 0.0 and 1.0).
`*oldrand`	`double`	This array contains 55 random numbers.

`jrand`	`integer`	This variable keeps a count of the number of random numbers used from the array `oldrand`.

Finally, a pointer `*fin` of type `FILE` is declared as a global variable for the input file stored in the variable `Erafilename`. This allows an easier way to read era-dependent information without opening and closing the file at each era.

3 File descriptions

This section presents a brief description of the functions used in the code. The functions are grouped into separate files according to their role in messy GA operation. There are eleven files: `initial.c`, `generate.c`, `operators.c`, `supports.c`, `functions.c`, `objfunc.c`, `stats.c`, `report.c`, `main.c`, `utility.c`, and `random.c`. The file `initial.c` contains functions to read problem information, to initialize global variables, and to create the initial population. The file `generate.c` contains functions that execute one generation of mGA operation. The file `operators.c` contains functions for all messy genetic operators. The file `supports.c` contains auxiliary functions associated with messy GA operators. The file `functions.c` contains functions associated with user-defined subfunctions. The file `objfunc.c` contains a function for the objective function evaluation. The file `stats.c` contains a function to calculate the population statistics. The file `report.c` contains reporting codes. The file `main.c` is the main routine that controls the overall mGA operation. The file `utility.c` contains all utility functions that are used in other routines, and the file `random.c` contains functions for a random number generator. Each function in each file is briefly described in the following subsections.

3.1 The file `initial.c`

This file contains functions that read all user-specified problem information, initialize global variables, and create an initial population.

`get_input()`	`void`	This function begins by verifying the existence of all input files and by creating all required output files. Thereafter, all user-specified parameters are read and the random number generator is initialized. The template is set up next. Finally, the objective function and other auxiliary information are read from various input files.
`initialize()`	`void`	This function calls routines to create the initial population and to report population statistics. This function also invokes the function that sets up the population reduction schedule during the primordial phase.
`initialize_pop()`	`void`	This function calculates the size of the initial population and allocates memory for the population members. It then invokes the routine `initpop()` to create the initial population.
`extra_pop()`	`void`	This function creates additional population members in the initial population. The genic and allelic information are obtained from a specified input file.

`storepop(pop, n)`	`void`	This function allocates storage for `n` individuals in the population array, `pop`. All elements of an individual are allocated and initialized.
`initpop(last_member, n)`	`void`	This function creates `n` individuals by initializing genes and alleles of each chromosome. For each genic combination, individuals are created with all possible allele combinations. For a chromosome of size k, each genic combination forms as many as 2^k allelic combinations. If the problem size is ℓ, there are as many as $\binom{\ell}{k}$ different genic combinations possible. Thus, for a problem of size ℓ, the initial population has as many as $2^k \binom{\ell}{k}$ many unique individuals. Each individual is duplicated, if more than one copy is desired. This population size may be reduced to $\binom{\ell}{k}$, if `r_initpop_flag` is set to `1`. In this case, the population is created by mutating the template.
`mut(bit)`	`bit`	This function alters a `bit` to its complement.
`next_genic_comb(list, n)`	`void`	The input to this function is an array of genes of length `n`. This function determines the next possible genic combination. For example, if the input to this function is an array `(1 2 3)`, the output is the array `(1 2 4)`. This routine is called by the function `initpop()`.
`next_allele_comb(list, pos)`	`void`	This function determines the next possible allele combination to that stored in `list`. For example, if the input to the function is `(0 1 0)`, the output is `(0 1 1)`. This routine is called by the function `initpop()`.
`get_template()`	`void`	This function sets up the template for the initial era. First, the variables `bytesize` and `*bytelimit` are assigned according to the specified problem size. Thereafter, if a level-wise mGA is used, a random template is created by invoking the routine `randomtemplate()`; otherwise the user-specified template is read from the input file, `template`.
`randomtemplate(temp)`	`void`	This function creates a random array of `1`s and `0`s of size `problem_length`.
`objfunc_info()`	`void`	This function reads the objective function information. First, the `function`s and `table`s are read. Thereafter, genes specifying each subfunction are read.
`read_subfunction(fp, n, nstruc)`	`void`	The reading of the subfunction information is aided by this function.
`partition_info()`	`void`	This function reads the partition information for analysis. This function is invoked only if the variable `partitionflag` is set to `1`.
`poprec_info()`	`void`	If the flag `popprintflag` is set to `1`, this function reads the generations in which a population history is desired.

`setup_popsize()`	`void`	This function sets up a population reduction schedule in the primordial phase. The population size is reduced depending on the initial population size, juxtapositional population size, and the proportion of the best building blocks in the initial population. During primordial phase, the best individuals grow in a logistic manner. The duration of the primordial phase, `prim_gen`, is calculated as the number of generations required for the best individuals to take up approximately 50% of population. The initial population size is then scheduled to reduce to the juxtapositional size in a systematic way. Depending on the required number of population reductions and allowable generations to achieve these reductions, the population is first reduced once every other generations, thereafter the population is reduced at every generation.

3.2 The file `generate.c`

This file contains functions that create a new population from an old population by messy genetic operators.

`generate()`	`void`	This function performs one generation of messy GA operation. If the generation counter, `gen`, is less than `prim_gen`, the primordial phase is used, otherwise the juxtapositional phase is used. Population statistics of the new population is calculated and reported. The routine terminates by assigning the new population to the old population.
`primordial()`	`void`	This function executes the primordial phase. Only selection is used to create the new population. If the population size is changed, storage for variables `oldpop`, `newpop`, and `shuffle` are reallocated according to the new population size. Since no new individual is created in this phase, individuals in the new population are not evaluated.
`juxtapositional()`	`void`	This function executes the juxtapositional phase. All allocated memory for the new population is released. Two individuals are first selected from the old population, then recombined using cut, splice, and mutation operators, and finally placed into the new population. The objective function value for the new individuals are evaluated. The `fullchrom` and `fullgene` of the new individuals are also assigned. These procedures are continued until the new population is filled.
`change_popsize()`	`long`	This function calculates the population size at each generation in the primordial phase. Figure 2 illustrates a typical population reduction schedule in the primordial phase.

3.3 The file operators.c

This file contains functions that are associated with the messy GA operators. The selection operators are described first. Thereafter, cut, splice, and mutation operators are discussed.

`select_pop()`	`INDV_ID`	This function determines the selection operator to be used at any generation.
`select(first, second)`	`INDV_ID`	This function selects the better of the two individuals indexed by counters, `first` and `second`, in the old population. In the case of an maximization problem, the individual with higher objective function value is chosen. If tie-breaking is in effect, this function chooses the shorter individual (one with fewer genes), if there is a tie in `fitness` values between two individuals.
`init_select()`	`INDV_ID`	This function performs tournament selection without shuffling of the population. This routine is used only for the first `init_select_gen` generations to avoid the stochastic sampling error early on. (see figure 2)
`normal_select()`	`INDV_ID`	This function performs tournament selection with shuffling of the population. In the absence of thresholding, this routine is used. (see figure 2)
`thresholding_select()`	`INDV_ID`	This function performs tournament selection with shuffling and thresholding. Two competing individuals are so chosen that there are threshold number of common genes between them. This allows like individuals to be compared with like. The variable `fullgene` keeps a record of the expressed genes in a chromosome. The number of common genes between two individuals are calculated by taking an intersection of the binary array, `fullgene`, of two individuals. This is performed in the function `matched_genes()`. A statistical analysis shows that the expected number of common genes between two random strings of size λ_1 and λ_2 is $\lceil \frac{\lambda_1 \lambda_2}{\ell} \rceil$, where ℓ is the problem size. If the number of common genes is greater than this threshold number, two individuals are compared, otherwise a new individual is tried.
`matched_genes(gn1, gn2)`	`integer`	This function counts the number of common genes between two individuals by making an intersection between `gn1` and `gn2`.
`cut_and_splice(chr1, chr2)`	`void`	This function performs the cut and splice operation between two individuals `chr1` and `chr2`. The first chromosome is cut into two strings `chr1` and `chr4` with a bitwise cut probability, `cut_prob`. If the string `chr4` exists, it is pushed into a stack, `chrom_stack`. The second chromosome is cut with the same probability into two strings `chr2` and `chr3`. The string `chr2` is pushed next, followed by the string `chr3`. Finally, the string `chr1` is pushed. Thereafter, the function calls `test_splice()` to splice two successive strings picked from the top of the stack, `chrom_stack`.

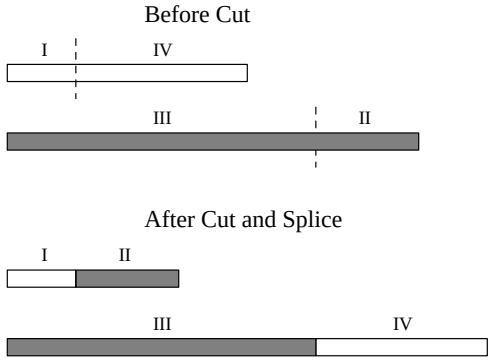

Figure 3: A messy cut and splice operation.

`test_splice()`	`BOOLEAN`	This function sends a `1` if the stack, `chrom_stack`, is nonempty. Two strings are picked from the top of the stack `chrom_stack` and spliced with a probability, `splice_prob`. The cut and splice operation is illustrated in figure 3.
`cut(chr1, chr2)`	`void`	This function severs an individual specified by the variable `chr1` into two individuals at a random site. The resulting individuals are passed through the argument as `chr1` and `chr2` respectively. The cut is performed by assigning the `nextgene` of the gene at cutsite to `NIL` and by assigning the `firstgene` of `chr2` to the address of the gene following cutsite. The size of each chromosome is modified and storages for `fullchrom` and `fullgene` of `chr2` are allocated.
`splice(chr1, chr2)`	`void`	This function splices two individuals `chr1` and `chr2` into one individual `chr1`. The splice is performed by assigning the `lastgene` of `chr1` to the `firstgene` of `chr2`. The chromosome size of `chr1` is modified and all allocated memory for `chr2` are released.
`mutation(chr)`	`void`	This function coordinates both genic and allelic mutation on the individual `chr`. An allelic mutation is performed with a probability, `allelic_mut_prob`, and a genic mutation is performed by invoking a routine `genic_mutation()` with a probability, `genic_mut_prob`.
`genic_mutation(num)`	`GENE_ID`	This function changes a gene to another gene at random. This routine is called by the function `mutation()`.

3.4 The file `supports.c`

This file contains a number of functions that are primarily used to support the functions used in file `operators.c`.

`length_chrom(chr)`	`integer`	This function calculates the length of a chromosome `chr` by counting the number of genes in the chromosome.

`fill_chrom(chr)`	`void`	This function calculates `fullchrom` and `fullgene` for a chromosome `chr`. The array `fullchrom` contains an allele value at each gene position. The problem of overspecified genes is resolved by using first-come-first-served rules with a left-to-right scan on the chromosome, and the underspecification problem is resolved by borrowing the unspecified genes from the template. The array `fullgene` is either a `1` or a `0` depending on whether the gene is present or absent in the original chromosome.
`copy_chrom(chr1, chr2)`	`void`	This function copies an individual `chr1` to a variable, `chr2` of type `INDIVIDUAL`. Each element of the structure, `INDIVIDUAL`, is copied.
`isempty(stack)`	`BOOLEAN`	This function checks if the `stack` is empty.
`pop_stack(stack, chr)`	`BOOLEAN`	This function gets the top member from the `stack`. If the `stack` is nonempty, a `1` is sent; otherwise a `0` is sent.
`push_stack(stack, chr)`	`void`	This function pushes an individual `chr` into the `stack`.
`pickallele(id, rawchr)`	`ALLELES`	This function picks the allele value of the gene indexed by `id` from the array, `rawchr`. Since alleles are stored in bits, bitwise operators are used.
`shuffle_pop()`	`void`	This function calculates a shuffled array of population members.
`assign_beststring_to_template()`	`void`	This function assigns the best string found in an era to the template for the next era.

3.5 The file `functions.c`

This file contains all functions that are used as subfunctions. The user defines the objective function(s) in this file.

`get_func_val(id, chr, len)`	`double`	This routine controls the evaluation of each subfunction. The routine is set to handle upto ten different subfunctions. More subfunctions can be used by extending the number of `case`s in the `switch` environment. (refer to the code on page 82)
`file1-10(chr, len)`	`double`	These functions are user defined objective functions. The variable `chr` contains an array of alleles representing a subfunction and the variable `len` is the size of the array. Liepins and Vose's (1990) deceptive function (`file0()`) and David Ackley's (1987) test bed (`file1()-file5()`) are included in this file.

3.6 The file `objfunc.c`

This file contains a function that evaluates the overall objective function value of an individual.

`objfunc(rawchr)`	`double`	This function calculates the objective function value of an individual by summing all its subfunction values. First, the allele combinations for each subfunction is gathered and the appropriate function is called for evaluation. If the subfunction is a `table`, the subfunction value is calculated from the user-specified tables; otherwise the subfunction value is calculated from the user-specified functions given in the file `functions.c`. Each subfunction value so obtained is multiplied by the user-specified scale factor and added.

3.7 The file `stats.c`

This file contains a function that calculates the population statistics at any generation and evaluates a termination condition for the current era.

`statistics(pop)`	`void`	This function calculates a number of population statistics—average, minimum, and maximum objective function value, cumulative number of function evaluations, and the population average string length. It also assigns the best individual in the current population to the variable, `best_indv`. Finally, the termination condition is checked. If the proportion of the best individual in the population is greater than a user-specified factor (given in input file `parameters`), the `stopmgaflag` is set to `1`, and the era is terminated. An era may also be terminated, if the initial population contains no individual better than the template. This reduces the function evaluations in an era where no likely improvement of the individuals is anticipated.

3.8 The file `report.c`

This file contains functions that writes various output files.

`reportpop(pop)`	`void`	This function controls the reporting of all output files.
`problem_rep()`	`void`	This function performs the reporting of all important problem information at the top of the output file specified by the variable `Outputfilename`.
`general_rep()`	`void`	This function writes the population statistics in the file specified by the variable `Outputfilename`.
`plot_rep()`	`void`	This function writes the population statistics for plotting purpose. The statistics are written in the file specified by the variable `Plotfilename`.
`fwritechrom(fp, chr, skipchar)`	`void`	This function writes a messy chromosome, `chr`, into a file with a file-pointer `fp`. The `skipchar` is the number of characters to be skipped in the beginning of a line for clarity.

`fwritefullchrom(fp, chr)`	`void`	This function writes a `fullchrom` (an array of `1`s and `0`s) of an individual in a file with a file-pointer `fp`.
`initreport()`	`void`	This function echos the input data and writes the basic parameters used in the messy GA operation.
`writepop(pop)`	`void`	This function copies all individuals in a population, `pop`, into the output file specified by the variable `Popfilename`.
`partitionprop(pop)`	`void`	This function calculates the proportion of all partitions specified by the user in the file specified by the variable `Partoutfilename`.

3.9 The file `main.c`

This file contains the main routine that controls the overall mGA operation.

`main()`	`void`	This function controls the overall messy GA operation. First, the function that reads all input parameters is called. At each era, the function `generate()` is called until the maximum specified generation is reached or enough convergence is achieved. The best individual found in the current era is assigned as the template for the next era. The current era is then terminated and the next era is started. This is continued until the maximum specified era is completed. But, if only a particular era of mGA is desired, the simulation resumes with the specified era and terminates after completing that era.

3.10 The file `utility.c`

This file contains a number of utility functions that are used in other files.

`gen_file(file, filevar, name)`	`void`	This function assigns a `file` specified by the variable, `filevar`, its `name`. For example, `gen_file("Objective function", Objfilename, "subfunc")` assigns the `objective function` file specified by the variable, `Objfilename`, a name, `subfunc`.
`get_string(fp, s, n)`	`void`	This function reads a string `s` of size `n` from a file with a file-pointer `fp`.
`check_input_file(filename)`	`void`	This function checks if the specified input file, `filename`, exists. If the file does not exist, an error message is sent and the execution is terminated.
`scratch_file(filename)`	`void`	This function creates an output file.
`print_error(msg)`	`void`	This function writes an error message on the console.
`sortnum(num, list)`	`void`	This function sorts the numbers in the array, `list`, in ascending order.

`maximum(num, array)`	`integer`	This function calculates the maximum value in an `array` of integers.
`round(num)`	`integer`	This function rounds off the number `num`.
`power(a, b)`	`long`	This function calculates $\mathtt{a}^{\mathtt{b}}$, where `a` and `b` are both integers.
`fact(n)`	`long`	This function calculates the factorial of a number `n`.
`choose(n,k)`	`long`	This function calculates $\binom{\mathtt{n}}{\mathtt{k}}$, the number of combinations of `n` objects taken `k` at a time.
`ones(chr, len)`	`integer`	This function calculates the number of `1`s in a binary string, `chr`, of length `len`.
`decode(chr, len)`	`double`	This function calculates the decoded value of a binary string, `chr`, of length `len`.
`map(min, max, x, len)`	`double`	This function linearly maps any number, `x`, in the range $(0, 2^{\ell}-1)$ to a range (`min`, `max`).
`parity(num)`	`BOOLEAN`	This function checks whether the number, `num`, is odd or even. A `1` is sent if the number is even, otherwise a `0` is sent.
`reallocate_memory(size)`	`void`	This function reallocates memory for the variables, `oldpop`, `newpop`, and `shuffle`. These variables are updated according to the modified population size in the primordial phase.
`freenewpop()`	`void`	This function deletes all memory allocated for the variable `newpop`.
`freeallmemory()`	`void`	This function deletes memory allocated for all era-dependent variables. At the end of each era, this routine is called to free all era-dependent variables used in the current era and to make rooms for the new allocations in the next era.
`delete_chrom(chrom)`	`void`	This function recursively deletes all genes in a chromosome.
`reset_list(list)`	`void`	This function resets all elements of the `list` to zero.
`repchar(out, ch, repcount)`	`void`	This function repeatedly writes a character, `ch`, to an output devise.
`page(out)`	`void`	This function issues a form feed to an output device.
`skip(out, skipcount)`	`void`	This function skips `skipcount` lines on an output device.
`skip_space(out, skipcount)`	`void`	This function skips `skipcount` characters in a line.

3.11 The file `random.c`

This file contains a random number generator. Random numbers are generated according to the subtractive method described in Knuth (1981).

`advance_random()`	`void`	This function creates a batch of 55 random numbers.
`warmup_random(seed)`	`void`	The random number generator is initialized in this function. A `seed` is used to create an array of random numbers. This function calls `advance_random()` to create a set of 55 random numbers.
`randomize()`	`void`	This function gets a seed number and initiates the random number generator.
`random()`	`double`	This function creates a random number between 0.0 and 1.0.
`rnd(low, high)`	`integer`	This function generates a random number between integers, `low` and `high`.
`flip(prob)`	`BOOLEAN`	This function flips a coin with a probability, `prob`. If the flipping is true, a `1` is sent; otherwise, a `0` is sent.

4 Input files

This section describes various input files used in mGA in C. There are altogether seven input files—`parameters`, `era`, `subfunc`, `template`, `partitions`, `poprecin`, and `extra`. The file `parameters` contains information about the global parameters used in the code. The file `era` contains global parameters that change at each era of mGA operation. The file `subfunc` contains objective function information. The file `template` contains a user defined template and is only required if a particular era of mGA is desired. The file `partitions` contains user defined partitions for investigation. The file `poprecin` contains user defined generations for population records and the file `extra` contains information to create additional population members in the initial population. In the following, each of the above files is briefly described.

4.1 The file `parameters`

This file contains user-specified global parameters. A typical value for each parameter is shown in parenthesis. The user needs to specify the desired value at the right side of the equal (`=`) sign.

```
Maximization (1) =
Level wise mGA (1) =
Problem size (15) =
Maximum era (3) =
Probability of cut (0.02) =
Probability of splice (1.0) =
Probability of allelic mutation (0.0) =
Probability of genic mutation (0.0) =
Thresholding (0) =
Tiebreaking (0) =
Reduced initial population (1) =
```

```
Extra population members (0) =
Stopping criteria factor (1.0) =
Partition file (0) =
Plotting file (0) =
Population record file (0) =
Trace (1) =
Copies (10 1 1) =
```

The entry in the first line is a `1` if the problem needs to be maximized, otherwise a `0` is entered. The next line contains information about the type of mGA to be used. If a level-wise mGA is desired, a `1` is entered. To run only a particular era, a `0` is entered. The problem size is entered in the third line. The `Maximum era` is set to the upper limit of the era to be used in the operation. If only a particular era of mGA is desired, that era is entered here. The probabilities of cut, splice, allelic mutation, and genic mutation are specified next. If thresholding is desired, the next entry is set to a `1`. To invoke tie-breaking, the next entry is set to a `1`. If a reduced initial population is desired, the next entry is set to a `1`. The next entry is a number between 0.0 and 1.0, defining the desired minimum pro portion of the best individual in the population for the termination of an era. If a partition analysis is desired, the next entry is set to a `1` and the partition information are specified in file `partitions`. If a plotting file with all population statistics is desired, the next entry is set to a `1`. If a record of all population members at any generation is desired, the entry on `Population record` is set to a `1` and the specific generations are listed in the input file `poprecin`. If a trace of mGA operation is desired, a `1` is entered in the next line. The next line requires an array of integers, each specifying the number of duplicates per individual in the initial population for each era. Thus, the number of entries in this line is equal to the number of era to be run.

4.2 The file `era`

This file contains the information about the parameters that are era-dependent. There are two entries for each era. A typical file for three eras is shown below:

```
Total generations (20) =
Juxtapositional popsize (100) =
Total generations (20) =
Juxtapositional popsize (100) =
Total generations (20) =
Juxtapositional popsize (100) =
```

The maximum number of generation to be used in an era is entered first. The desired population size in the juxtapositional phase is entered next. The actual population size used in the simulation may be different than the size entered, depending on the a number of factors discussed before (page 14). These two entries are repeated as many times as the total number of era to be run.

4.3 The file `subfunc`

This file contains objective function information. `Table`s and `function`s used to define subfunctions are specified next. Genes specifying a subfunction are given next. A typical file is illustrated in the following:

```
Subfunction Tables and functions
5
```

```
------------------------------
0
Table
3
0 0 0  28
0 0 1  26
0 1 0  22
0 1 1  0
1 0 0  14
1 0 1  0
1 1 0  0
1 1 1  30
------------------------------
1
Table
3
0 0 0  0
0 0 1  2
0 1 0  4
0 1 1  12
1 0 0  16
1 0 1  20
1 1 0  28
1 1 1  30
------------------------------
2
Function
7
------------------------------
3
Function
4
------------------------------
4
Function
10
------------------------------
Subfunction data
4
0 1.0 0 1 2
0 1.5 3-5
3 1.0 6-9
2 5.0 9-15
```

The first line in this file is a comment. The second line specifies the number of different `function`s or `table`s listed in this file. The next line is a comment line and used as a separator between two subfunction entries. Each entry starts with an identification number, followed by the type of the entry (a `table` or a `function`) in successive lines. This representation allows a unique identification number for `function`s or `table`s. However, it is not imperative that the `table`s and

functions are listed in any specific order of their identification number. The number of genes specifying the the table or the function is specified next. For a table, an allele combination followed by the corresponding objective function value are specified. This requires 2^k entries in a table of size k. In the above file, the allele combinations are listed in ascending order of their decoded values. But, in general, allele combinations may not be entered in any specific order. For a function, the evaluation function is specified in the file functions.c. A typical evaluation function is illustrated in the following:

```
double file1(chr, len)
int chr[];
int len;
{
       double x, val;

       x = ones(chr, len);
       return((double) x);
}
```

The variable chr is a binary string of alleles in a subfunction obtained from a messy chromosome. The variable len is the size of the subfunction. The user needs to supply a code that uses these two variables and calculates a function value. For example, the above function counts the number of 1s in a string. The routine ones() is supplied as an auxiliary function in the file utility.c. Two other utility functions, decode(chr, len) and map(min, max, x, len), are also available as utility files. The former function calculates the decoded value of the binary string chr, and the latter function maps a value x $\in [0, 2^\ell - 1]$ into a region [min, max].

The genes specifying each subfunction are entered next. The line with the entry Subfunction data is a comment line. Total number of subfunctions in a problem is specified in the next line. The following entries are information about each subfunction. The first entry is an identification number of the table or the function for a subfunction. The next entry is the scale factor to be used for the subfunction. The overall function value is calculated by summing each subfunction value multiplied by the corresponding scale factor. Finally, genes constituting the subfunction are entered in order. Successive genes in a subfunction may be entered by writing a '–' between boundary genes. The second subfunction shows an example of genes 3, 4, and 5 written as 3-5.

4.4 The file template

This file contains the template information. If a level-wise mGA is desired, this file is not required. In this case, the mGA starts with order-one strings and a random template is used. For subsequent eras, the best string found in the previous era is used as the template. But if a particular era of mGA is desired, a template specifying an allele value at every gene position is needed. For a problem of size 15, a typical template file is shown below:

```
0 0 0 1 1 1 0 0 1 1 0 1 0 1 0
```

4.5 The file partitions

This file contains information about the partitions to be investigated. A typical file is shown below:

```
Partition data
3
```

```
3 0 1 2
3 0 1 3
5 0 3 6 9 12
```

The first line is a comment. The next entry is the number of different partitions to be investigated. Each partition entry begins with the size of the partition, followed by the set of genes specifying the partition. For example, in the above file, there are three partitions to be investigated. The first partition contains three genes `0`, `1`, and `2`, in that order.

4.6 The file `poprecin`

This file contains information about the generations at which a population record is desired. A population record is printed at each specified generation for each era. A typical file is shown below:

```
Generation numbers
2
0 10
```

The first line is a comment line. The next entry is the total number of specified generations. The next line contains the generation numbers at which a population record is desired. If the entry `Population record` in the file `parameters` is set to a `1`, a population record at each specified generation is written in a file. The above file specifies that population records at the initial and the tenth generation are desired.

4.7 The file `extra`

This file contains the set of genes for additional population members to be included in the initial population. A typical file is shown below:

```
Set of genes
3 3 4 5 10
2 1 2 5
```

If the entry `Extra population members` in the file `parameters` is set to a `1`, extra population members are included in the initial population. The first line is a comment line. Thereafter, each line represents a set of genes for additional population members. The first entry is the number of genes in the set. The next entries are the gene numbers for population members. All combinations of alleles for a gene set are included in the initial population. The final entry in a line is the number of copies of each member to be included in the initial population. The second line in the above file specifies that ten copies of all eight chormosomes with genes `3`, `4`, and `5` are to be included in the initial population.

5 Installing `mGA in C`

MGA in C should run on most UNIX machines with a C compiler. This version has run successfully on an IBM RISC System/6000^{TM} machine. Minor changes may be necessary to run it in other machines.

It is suggested that a new directory is created and all files from the distribution diskette are copied into this directory. There are altogether eleven `c` files, a `Makefile`, and seven input files as discussed in the previous sections. The input files are sample files that may be modified according to the user's needs.

6 Running mGA in C

This section describes the procedure of running mGA in C. The UNIX command `make` compiles and links all `c` files. The new evaluation functions are included in file `functions.c` before executing the command `make`. The following command creates an executable file called `mga`.

```
% make
```

The input files are set according to the format described earlier. Thereafter, mGA in C can be run by executing the command:

```
% mga
```

This command produces a series of statements that awaits user's prompt. All input and output file names are entered at this point. The default file names are listed in parenthesis. The following statements are echoed when the command `mga` is executed.

```
Parameters file name (parameters) =

Era file name (era) =

Objective function file name (subfunc) =

Template file name (template) =

Partition input file name (partition) =

Pop record input file name (poprecin) =

Extra members file name (extra) =

Output file name (output) =

Pop record output file name (poprecout) =

Partition output file name (partout) =

Plotting file name (plot) =

Enter seed random number (0.0..1.0)->
```

If a different file name is desired, the full name is typed. Finally, a seed for the random number generator is entered. This initiates the execution of mGA in C.

7 Output files

The section describes the output files generated by mGA in C. The file `output` contains the global parameters and population statistics for each generation. Other files are generated according to

different options provided by the user in the input file `parameters`. Each of these files is described in the following.

7.1 The file `output`

The echo of the input parameters and the population statistics for each era are printed in this file. First, the input parameters are echoed. The population statistics—maximum, minimum, and average population objective function values, total number of function evaluations, and average string length—versus generation number are tabulated next. Thereafter, the best string found in the era is printed.

7.2 The file `plot`

This file is created, only if the entry under `Plotting file` in input file `parameters` is set to `1`. A number of population statistics at each generation is printed. This file is designed to facilitate the plotting of various population statistics. The columns in this file corresponds to the following parameters (in order):

```
Generation  Population    Average     Maximum    Minimum     Average    Function
  Number       size    string length     Objective function value     evaluations
    (1)        (2)          (3)         (4)        (5)         (6)          (7)
```

The generation number printed in this file is the cumulative number of generations since the beginning of the mGA operation.

7.3 The file `poprecout`

This file is created, only if the entry under `Population record file` in input file `parameters` is set to `1`. This produces a complete record of the population members at specified generations given in input file `poprecin`. For each population member, the objective function value, string length, and the chromosome are printed.

7.4 The file `partout`

This file is created, only if the entry under `Partition file` in input file `parameters` is set to `1`. First, the partitions are echoed. The number of strings in a partition depends on the size of the partition. If the size of a partition is p, there are 2^p strings in that partition. The proportion of each string in the population depends on the underlying objective function. The number of each string of a partition in a population is printed in an ascending order of the string's decoded parameter value.

8 An example problem

This section shows the use of mGA in C to solve an example problem. The problem used here is a 30-bit, order-three deceptive problem. Three consecutive genes are used to form a subfunction. Thus there are ten subfunctions each having the same 3-bit function. The file `subfunc` is shown in figure 4. All subfunctions are same and the function value is taken from the specified `table`. Since no new evaluation function is used, the file `functions.c` need not be changed. Other input files are shown below. Since a level-wise mGA is used, the input file `template` is not required. The following commands are used to compile and execute the code. The code prompts a number

```
Subfunction Tables and functions
1
--------------------------------
0
Table
3
0 0 0  28
0 0 1  26
0 1 0  22
0 1 1  0
1 0 0  14
1 0 1  0
1 1 0  0
1 1 1  30
--------------------------------
Subfunction data
10
0 1 0 1 2
0 1 3 4 5
0 1 6 7 8
0 1 9 10 11
0 1 12 13 14
0 1 15 16 17
0 1 18 19 20
0 1 21 22 23
0 1 24 25 26
0 1 27 28 29
```

Figure 4: The file `subfunc` for the example problem.

```
Maximization (1) = 1
Level wise mGA (1) = 1
Problem size (15) = 30
Maximum era (3) = 3
Probability of cut (0.02) = 0.016667
Probability of splice (1.0) = 1.0
Probability of allelic mutation (0.0) = 0.0
Probability of genic mutation (0.0) = 0.0
Thresholding (0) = 0
Tiebreaking (0) = 0
Reduced initial population (1) = 0
Extra population members (0) = 0
Stopping criteria factor (1.0) = 1.00
Partition file (0) = 0
Plotting file (0) = 1
Population record file (0) = 0
Trace (1) = 1
Copies (10 1 1) = 5 1 1
```

Figure 5: The file `parameters` for the example problem.

```
Total generations (20) = 20
Juxtapositional popsize (100) = 250
Total generations (20) = 20
Juxtapositional popsize (100) = 250
Total generations (20) = 20
Juxtapositional popsize (100) = 250
```

Figure 6: The file `era` for the example problem.

of statements regarding the file names of the input and output files. Since default file names are used, a <cr> is typed in each response. A random seed of 0.985 is used.

```
% make
% mga
Parameters file name (parameters) = <cr>

Era file name (era) = <cr>

Objective function file name (objfunc) = <cr>

Template file name (template) = <cr>

Partitions file name (partitions) = <cr>

Pop record input file name (poprecin) = <cr>

Extra members file name (extra) = <cr>

Output file name (output) = <cr>

Pop record output file name (poprecout) = <cr>

Partition proportion file name (partout) = <cr>

Plotting file name (plot) = <cr>

Enter seed random number (0.0..1.0)-> 0.985 <cr>
```

At the end of the run, two new files—`output` and `plot`—are created. The file `output` is shown in figure 7.

```
Start-up time is : Tue Sep 24 16:44:15 1991

Problem length         = 30
Number of subfunctions = 10
Subfunction            =   0   1   2   3   4   5   6   7   8   9
Table or function id   =   0   0   0   0   0   0   0   0   0   0
Subfunction size       =   3   3   3   3   3   3   3   3   3   3
Constitutive Genes:
       Subfunction   0 =   0   1   2
       Subfunction   1 =   3   4   5
       Subfunction   2 =   6   7   8
       Subfunction   3 =   9  10  11
       Subfunction   4 =  12  13  14
       Subfunction   5 =  15  16  17
       Subfunction   6 =  18  19  20
       Subfunction   7 =  21  22  23
       Subfunction   8 =  24  25  26
```

```
       Subfunction   9 =  27  28  29

Splice probability     = 1.000000
Cut probability        = 0.016667
Allelic mutation prob. = 0.000000
Genic mutation prob.   = 0.000000

Thresholding (1 if on) = 0
Tiebreaking  (1 if on) = 0

Random seed            = 0.985000
--------------------------------------------------------------------------------

Era                    = 1
Init_select_gen        = 0
Cutpop_gen             = 0
Primordial phase       = 0
Initial popsize        = 300
Popsize in juxt. phase = 300
Maximum generation     = 20

Template               = 10011011011110100000 1001100011

****************************** era = 1 ******************************
Gen  Population  Average   Maximum    Minimum    Average       Fuction
        size    str. len.  objfunc    objfunc    objfunc     evaluations

0         300       1.00     168.00     108.00     138.67     3.0100e+02
----------------------- end of primordial phase -----------------------
1         300       2.00     198.00     112.00     153.20     6.0100e+02
2         300       3.85     228.00     124.00     178.54     9.0100e+02
3         300       7.08     264.00     138.00     201.77     1.2010e+03
4         300      12.46     290.00     138.00     210.92     1.5010e+03
5         300      19.18     286.00     138.00     216.99     1.8010e+03
6         300      25.92     288.00     138.00     222.73     2.1010e+03
7         300      32.48     286.00     152.00     226.65     2.4010e+03
8         300      37.58     284.00     140.00     231.55     2.7010e+03
9         300      44.62     286.00     152.00     233.46     3.0010e+03
10        300      50.69     286.00     152.00     237.99     3.3010e+03
11        300      55.81     286.00     138.00     241.55     3.6010e+03
12        300      64.22     286.00     168.00     242.72     3.9010e+03
13        300      73.38     286.00     152.00     248.38     4.2010e+03
14        300      82.10     286.00     152.00     250.75     4.5010e+03
15        300      92.97     284.00     168.00     255.59     4.8010e+03
16        300     101.31     284.00     150.00     257.22     5.1010e+03
17        300     116.09     282.00     152.00     259.26     5.4010e+03
18        300     135.45     282.00     178.00     261.79     5.7010e+03
19        300     152.93     280.00     192.00     265.65     6.0010e+03
```

```
20        300     165.03     284.00     168.00     267.45     6.3010e+03

The best string:
00011111111111100000000000111
Fitness =    290.000
String length =    16
Chromosome: ( (5   1) (0   0) (24  0) (21  0) (16  0) (13  1) (11  1) (27  1)
              (8   1) (20  0) (27  1) (24  0) (14  1) (23  0) (14  0) (5   1) )

End of run for era 1: Tue Sep 24 16:44:38 1991

Era                    = 2
Init_select_gen        = 1
Cutpop_gen             = 6
Primordial phase       = 6
Initial popsize        = 1740
Popsize in juxt. phase = 200
Maximum generation     = 20

Template               = 00011111111111100000000000111

****************************** era = 2 ******************************
Gen  Population  Average   Maximum    Minimum    Average       Fuction
        size    str. len.  objfunc    objfunc    objfunc     evaluations

0        1740       2.00     290.00     230.00     271.66     8.0420e+03
        Termination: no individual in the initial population is better
        than the template

The best string:
00011111111111100000000000111
Fitness =    290.000
String length =     2
Chromosome: ( (0   0) (1   0) )

End of run for era 2: Tue Sep 24 16:44:40 1991

Era                    = 3
Init_select_gen        = 1
Cutpop_gen             = 11
Primordial phase       = 11
Initial popsize        = 32480
Popsize in juxt. phase = 200
Maximum generation     = 20

Template               = 00011111111111100000000000111

****************************** era = 3 ******************************
```

```
Gen  Population  Average   Maximum    Minimum    Average       Fuction
        size    str. len.  objfunc    objfunc    objfunc     evaluations

0       32480       3.00     292.00     200.00     264.88     4.0523e+04
1       32480       3.00     292.00     200.00     276.19     4.0523e+04
2       16240       3.00     292.00     224.00     283.81     4.0523e+04
3       16240       3.00     292.00     258.00     288.03     4.0523e+04
4        8120       3.00     292.00     270.00     289.58     4.0523e+04
5        8120       3.00     292.00     284.00     289.98     4.0523e+04
6        4060       3.00     292.00     288.00     290.04     4.0523e+04
7        4060       3.00     292.00     290.00     290.08     4.0523e+04
8        2030       3.00     292.00     290.00     290.15     4.0523e+04
9        1015       3.00     292.00     290.00     290.30     4.0523e+04
10        507       3.00     292.00     290.00     290.54     4.0523e+04
11        200       3.00     292.00     290.00     290.99     4.0523e+04
---------------------- end of primordial phase ----------------------
12        200       5.61     294.00     262.00     290.03     4.0723e+04
13        200      10.06     298.00     234.00     289.10     4.0923e+04
14        200      16.25     300.00     236.00     289.96     4.1123e+04
15        200      24.60     300.00     250.00     290.88     4.1323e+04
16        200      33.90     300.00     238.00     293.31     4.1523e+04
17        200      42.88     300.00     232.00     295.11     4.1723e+04
18        200      51.62     300.00     236.00     295.44     4.1923e+04
19        200      60.36     300.00     240.00     296.66     4.2123e+04
20        200      68.00     300.00     240.00     296.69     4.2323e+04

The best string:
111111111111111111111111111111
Fitness =    300.000
String length =    24
Chromosome: ( (24  1) (25  1) (26  1) (18  1) (19  1) (20  1) (24  1) (25  1)
              (26  1) (21  1) (22  1) (23  1) (15  1) (16  1) (17  1) (21  1)
              (22  1) (23  1) (24  1) (25  1) (26  1) (0   1) (1   1) (2   1) )

End of run for era 3: Tue Sep 24 16:46:21 1991
```

Figure 7: The file `output` for the example problem.

It is observed that the best solution found in the first era contains the best subfunction value for five of the ten subfunctions. In these five subfunctions, the random template is exactly one bit away from the best subfunction value. Thus, the level-one mGA has been able to solve those subfunctions correctly. Since, in this problem, a lone `0` is better than than a lone `1`, each of the other places converges to a `0`. The best solution found in this era is used as the template for the second era.

Since any two-bit change in the template does not improve the function value, the second era is terminated and the same template is used in the third era. Since, order-three strings are included in the initial population, mGA has been able to combine all best order-three combinations and find the globally optimal solution. The total function evaluations required to solve the problem to global optimality is 41,123. It is interesting to note that initial populations require $(300 + 1,740 + 32,480)$

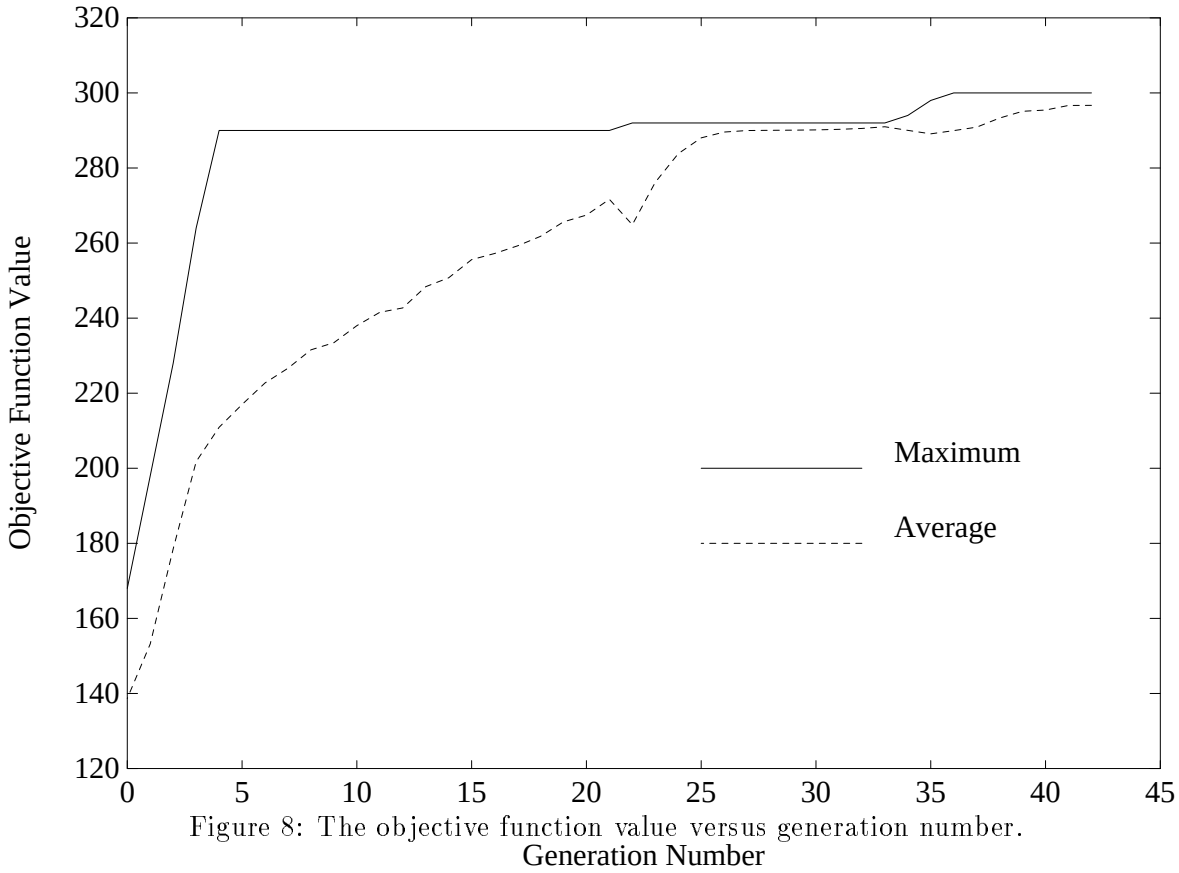

Figure 8: The objective function value versus generation number.

or 34,520 function evaluations, which is about 84% of the total function evaluations required to solve the problem. It may be possible to solve the same problem with fewer function evaluations, if the `r_initpop_flag` in the input file `parameters` is set `1`.

The population average maximum and average function values are plotted versus generation number in figure 8 using the output file `plot`. The plot shows that the solution improves at each generation and finally converges to the globally optimal point.

If it is desired to continue this experiment with different parameters, the input files can be edited. As long as a new function is not added in the `functions.c` file, recompilation is not needed. If, however, a new function is added in the `functions.c` file, the `make` command is executed before running `mga`.

9 Conclusions

This report has presented a brief introduction to messy genetic algorithms and a documentation of the data structure and functions used in mGA in C. The sample input and output files have been provided. The installing and running procedures of mGA in C have also been discussed. An example problem has been solved using mGA in C.

Because of the increased interest in the use of messy genetic algorithms, this first version of mGA in C has been released primarily for hand-on investigation of mGAs.

10 Acknowledgments

The authors acknowledge the support provided by the US Army under Contract DASG60-90-C-0153 and by National Science Foundation under Grants CTS-8451610 and ECS-9022007. The

authors appreciate any suggestions and comments regarding the implementation of the code (e-mail: deb@gal1.ge.uiuc.edu).

References

Deb, K. (1991). *Binary and floating-point function optimization using messy genetic algorithms.* PhD thesis. (IlliGAL Report No. 91004). Urbana: University of Illinois, Illinois Genetic Algorithms Laboratory. (Also available as TCGA Report No. 91004).

De Jong, K. A. (1975). An analysis of the behavior of a class of genetic adaptive systems (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International, 36*(10), 5140B. (University Microfilms No. 76-9381)

Goldberg, D. E. (1989a). *Genetic algorithms in search, optimization, and machine learning.* Reading, MA: Addison-Wesley.

Goldberg, D. E. (1989b). Genetic algorithms and Walsh functions: Part I, a gentle introduction. *Complex Systems, 3*, 129–152.

Goldberg, D. E. (1989c). Genetic algorithms and Walsh functions: Part II, deception and its analysis. *Complex Systems, 3*, 153–171.

Goldberg, D. E., & Bridges, C. L. (1990). An analysis of a reordering operator on a GA-hard problem. *Biological Cybernetics, 62* (5), 397–405.

Goldberg, D. E., Deb, K., & Korb, B. (1990). Messy genetic algorithms revisited: Studies in mixed size and scale. *Complex Systems, 4*, 415–444.

Goldberg, D. E., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems, 3*, 493–530.

Holland, J. H. (1975). *Adaptation in natural and artificial systems.* Ann Arbor, MI: University of Michigan Press.

Knuth, D. E. (1981). *The art of computer programming* (2nd ed. vol. 2). Reading, MA: Addison-Wesley.

Liepins, G. E., & Vose, M. D. (1990). Representational issues in genetic optimization. *Journal of Experimental and Theoretical Artificial Intelligence, 2*(2), 4–30.

11 The code in C

This section contains a listing of the functions used in mGA in C. First, files with the global declarations are listed. All c-files are listed next. Finally, the `Makefile` and example input files are included.

11.1 The file `mga.def`

```
/*==========================================================
        file : mga.def

     purpose : global definitions for mGAs

   developed : 1991

      author : Kalyanmoy Deb
==========================================================*/

/* define constants */

#define NIL 0
#define UNSINTSIZE 32 /* bitsize of an integer */
#define MAX_SIZE 500 /* maximum problem_length */
#define MAX_BYTE (1 + MAX_SIZE / UNSINTSIZE) /* maximum bytes reqd. */
#define MAX_ORDER 10 /* max size of a subfunction */
#define MAX_PARTITIONS 100 /* max partitions for report */
#define MAX_PARTITIONSIZE 10 /* max partition size */
#define PAGEWIDTH 80 /* allowable page width */


#define TRACE(s) if (traceflag) {printf(s); printf("\n");}

/* reading format-1 */
#define FORMAT_1 "\
Maximization (1) = %d\n\
Level wise mGA (1) = %d\n\
Problem size (15) = %d\n\
Maximum era (3) = %d\n\
Probability of cut (0.02) = %lf\n\
Probability of splice (1.0) = %lf\n\
Probability of allelic mutation (0.0) = %lf\n\
Probability of genic mutation (0.0) = %lf\n\
Thresholding (0) = %d\n\
Tiebreaking (0) = %d\n\
Reduced initial population (1) = %d\n\
Extra population members (0) = %d\n\
```

```
Stopping criteria factor (1.0) = %f\n\
Partition file (0) = %d\n\
Plotting file (0) = %d\n\
Population record file (0) = %d\n\
Trace (1) = %d\n"

#define VARS_1 &maximizationflag, &levelmgaflag, &problem_length, &max_era,\
               &cut_prob, &splice_prob, &allelic_mut_prob,\
               &genic_mut_prob, &thresholdingflag, &tiebreakingflag,\
               &r_initpop_flag, &extrapopflag, &stopfactor, &partitionflag,\
               &plotstatflag, &popprintflag, &traceflag

/* reading format-2 */
#define FORMAT_2 "\
Total generations (20) = %d\n\
Juxtapositional popsize (100) = %d\n"

#define VARS_2 &maxgen, &juxtpopsize
```

11.2 The file mga.h

```
/*==========================================================
      file : mga.h

   purpose : global variables

 developed : 1991

    author : Kalyanmoy Deb
==========================================================*/

#include <stdio.h>
#include <math.h>
#include "mga.def"


typedef int INDV_ID, GENE_ID, ALLELES, BOOLEAN;

struct GENE { /* a gene */
        GENE_ID genenumber; /* gene number */
        ALLELES allele :  1; /* allele value */
        struct GENE *nextgene; /* pointer to the next gene */
};

struct INDIVIDUAL { /* an individual */
        struct GENE *firstgene; /* pointer to the first gene */
        struct GENE *lastgene; /* pointer to the last gene */
        unsigned *fullchrom; /* an array of filled-up chromosome */
        unsigned *fullgene; /* an array of genes */
        int chromlen; /* chromosome length */
        int genelen; /* gene length */
        double fitness; /* objective function value */
};

struct INDIVIDUAL *oldpop,*newpop,best_indv; /* oldpop, newpop, and best
                                                individual */

/* system variables */
int era; /* current era */
int max_era; /* maximum order */
int order; /* current order */
int problem_length; /* problem size */
int *copies; /* number of copies in initial pop. */
long popsize; /* population size */
int bytesize; /* number of bytes to store raw chrom */
int *bytelimit; /* bits required for each byte */
unsigned *template; /* template array */
double function_evaluations = 0.0; /* number of function evaluations */
```

```
float stopfactor; /* termination factor */
double prop_bestindv; /* proportion of best individual */
double templatefitness; /* template fitness */


/* objective function related variables */
double **chromfitness; /* fitness array */
GENE_ID **genesets; /* genes in a subfunction */
int *str_length; /* size of subfunctions */
int *table_id; /* table id of subfunctions */
float *scale; /* scale for each subfucntion */
BOOLEAN *taborfunc; /* subfunction is a table or a function */
int numsubfunc; /* number of subfunctions */
double avgfitness, maxfitness, minfitness, avgstrlen; /* statistics */

/* operator related variables */
double cut_prob; /* probability of cut */
double splice_prob; /* probability of splice */
double allelic_mut_prob; /* prob. of allelic mutation */
double genic_mut_prob; /* prob. of genic mutation */
int init_select_gen; /* initial selection duration */
int cutpop_gen; /* duration till the population is cut */
int prim_gen; /* duration of primordial phase */
int maxgen; /* maximum generation */
INDV_ID *shuffle; /* shuffle array for selection */
int gen; /* generation counter */
INDV_ID pick; /* selection counter */
int juxtpopsize; /* popsize in juxtapositional phase */
int cut_every_other_gen; /* duration until cut at every other gen */
long allelicmutation = 0; /* counter for allelic mutation */
long genicmutation = 0; /* counter for genic mutation */
int thres; /* threshold value */
int shuffle_num = 0; /* shuffle number */

/* partition variables */
int numpartition; /* number of partitions */
int *partition_len; /* size of each partition */
GENE_ID **partition_genes; /* genes in each partition */

/* variables for recording population history */
int nextpopstatgen; /* next generation to be printed */
int countpopstatgen = 0; /* counter for population */
int *sortpopstatgen; /* sorted generations */

int gencount=0; /* counter for total generations */

double *oldrand; /* array used in random no. generator */
int jrand; /* counter in random number generator */
```

```
double randomseed; /* random seed */

struct STACKTYPE { /* stack for cut and splice operation */
        struct INDIVIDUAL genefirst; /* an individual */
        struct STACKTYPE *nextmem; /* pointer to the next individual */
};
typedef struct STACKTYPE *STACKPTR; /* pointer to the stack STACKTYPE */
STACKPTR chrom_stack = NIL; /* stack after cut only */
STACKPTR newchrom_stack = NIL; /* stack after cut and splice */

FILE *fin; /* pointer to the input file, era */

/* initialize flags */
BOOLEAN cutpopflag = 0; /* population size needs to be changed */
BOOLEAN thresholdingflag = 0; /* thresholding is active */
BOOLEAN tiebreakingflag = 0; /* tie-breaking is active */
BOOLEAN traceflag = 1; /* intermediate printing is on */
BOOLEAN levelmgaflag = 0; /* level-wise mGA is on */
BOOLEAN partitionflag = 0; /* partition proportions is required */
BOOLEAN maximizationflag = 1; /* a maximization problem */
BOOLEAN popprintflag = 0; /* population printing is on */
BOOLEAN plotstatflag = 1; /* file for plotting data is required */
BOOLEAN stopmgaflag = 0; /* stopping criteria is reached */
BOOLEAN r_initpop_flag = 0; /* if reduced population is desired */
BOOLEAN extrapopflag = 0;

/* File names */
char Inputfilename[30]; /* file of input parameters */
char Erafilename[30]; /* file of individual order info */
char Objfilename[30]; /* file of objective function information */
char Templatefilename[30]; /* file of template */
char Partinfilename[30]; /* file of structures to be printed */
char Outputfilename[30]; /* file of output */
char Poprinfilename[30]; /* file of population history */
char Poproutfilename[30]; /* file of population history */
char Partoutfilename[30]; /* file of partition proportions */
char Plotfilename[30]; /* file of population statistics */
char Extrafilename[30];

/* problem specific global parameters */
```

11.3 The file mga.ext

```
/*==========================================================
        file : mga.ext

     purpose : external global variables

   developed : 1991

      author : Kalyanmoy Deb
==========================================================*/

#include <stdio.h>
#include <math.h>
#include "mga.def"

/* see mga.h for variable description */

typedef int INDV_ID, GENE_ID, ALLELES, BOOLEAN;

extern struct GENE {
        GENE_ID genenumber;
        ALLELES allele :  1;
        struct GENE *nextgene;
};

extern struct INDIVIDUAL {
        struct GENE *firstgene;
        struct GENE *lastgene;
        unsigned *fullchrom;
        unsigned *fullgene;
        int chromlen;
        int genelen;
        double fitness;
};
extern struct INDIVIDUAL *oldpop,*newpop,best_indv;

extern int era;
extern int max_era;
extern int order;
extern int problem_length;
extern int *copies;
extern long popsize;
extern int bytesize;
extern int *bytelimit;
extern unsigned *template;
extern double function_evaluations;
extern float stopfactor;
extern double prop_bestindv;
```

```
extern double templatefitness;

extern double **chromfitness;
extern GENE_ID **genesets;
extern int *str_length;
extern int *table_id;
extern BOOLEAN *taborfunc;
extern float *scale;
extern int numsubfunc;
extern double avgfitness, maxfitness, minfitness, avgstrlen;

extern double cut_prob;
extern double splice_prob;
extern double allelic_mut_prob;
extern double genic_mut_prob;
extern int init_select_gen;
extern int cutpop_gen;
extern int prim_gen;
extern int maxgen;
extern INDV_ID *shuffle;
extern int gen;
extern INDV_ID pick;
extern int juxtpopsize;
extern int cut_every_other_gen;
extern long allelicmutation;
extern long genicmutation;
extern int thres;
extern shuffle_num;

extern int numpartition;
extern int *partition_len;
extern GENE_ID **partition_genes;

extern int nextpopstatgen;
extern int countpopstatgen;
extern int *sortpopstatgen;

extern int gencount;

extern double *oldrand;
extern int jrand;
extern double randomseed;

extern struct STACKTYPE {
        struct INDIVIDUAL genefirst;
        struct STACKTYPE *nextmem;
};
typedef struct STACKTYPE *STACKPTR;
```

```
extern struct STACKTYPE *chrom_stack = NIL;
extern struct STACKTYPE *newchrom_stack = NIL;

extern FILE *fin;

extern BOOLEAN cutpopflag;
extern BOOLEAN thresholdingflag;
extern BOOLEAN tiebreakingflag;
extern BOOLEAN traceflag;
extern BOOLEAN levelmgaflag;
extern BOOLEAN partitionflag;
extern BOOLEAN maximizationflag;
extern BOOLEAN popprintflag;
extern BOOLEAN plotstatflag;
extern BOOLEAN stopmgaflag;
extern BOOLEAN r_initpop_flag;
extern BOOLEAN extrapopflag;

extern char Inputfilename[];
extern char Erafilename[];
extern char Objfilename[];
extern char Templatefilename[];
extern char Partinfilename[];
extern char Outputfilename[];
extern char Poprinfilename[];
extern char Poproutfilename[];
extern char Partoutfilename[];
extern char Plotfilename[];
extern char Extrafilename[];

extern void statistics(), reportpop(), reallocate_memory(), copy_chrom();
extern void shuffle_pop(), storepop(), freenewpop(), fill_chrom();
extern void cut_and_splice(), mutation(), push_stack();
extern void delete_chrom(), print_error(), get_string();
extern void primordial(), juxtapositional();
extern void partition_info(), poprec_info(), partitionprop();
extern void initialize_pop(), initpop(), objfunc_info(), read_subfunction();
extern void genestructure_info(), setup_popsize(), get_template();
extern void general_rep(), plot_rep(), writefullchrom(), writepop();
extern void cut(), splice(), next_allele_comb(), extra_pop();
extern int ones(), rnd();
extern BOOLEAN parity(), pop_stack(), pop_stack(), flip();
extern double decode(), objfunc(), get_func_val(), random();
extern INDV_ID select_pop();
extern ALLELES pickallele();
```

11.4 The file `initial.c`

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

     purpose : create initial population and initialize parameters

   developed : 1991

      author : Kalyanmoy Deb
==========================================================*/

#include "mga.ext"


void get_input()
/* read and set input parameters */
{
        register int i;
        FILE *fp, *finput, *fopen();
        long clock, time();
        char *ctime(), msg[PAGEWIDTH];

        gen_file("Parameters", Inputfilename, "parameters");
        gen_file("Era", Erafilename, "era");
        gen_file("Objective function", Objfilename, "subfunc");
        gen_file("Template", Templatefilename, "template");
        gen_file("Partition input", Partinfilename, "partitions");
        gen_file("Pop record input", Poprinfilename, "poprecin");
        gen_file("Extra members", Extrafilename, "extra");
        gen_file("Output", Outputfilename, "output");
        gen_file("Pop record output", Poproutfilename, "poprecout");
        gen_file("Partition output", Partoutfilename, "partout");
        gen_file("Plotting", Plotfilename, "plot");

        /* check input files */
        check_input_file(Inputfilename);

        /* read input parameters */
        finput = fopen(Inputfilename, "r");
        if (fscanf(finput, FORMAT_1, VARS_1) != 17) {
                sprintf(msg,"Error in the input file:  parameters.\n");
                print_error(msg);
        }

        check_input_file(Erafilename);
        check_input_file(Objfilename);
        if (extrapopflag)
```

```
        check_input_file(Extrafilename);

/* open output files */
scratch_file(Outputfilename);
if (plotstatflag)
        scratch_file(Plotfilename);
if (popprintflag) {
        check_input_file(Poprinfilename);
        scratch_file(Poproutfilename);
}
if (partitionflag) {
        check_input_file(Partinfilename);
        scratch_file(Partoutfilename);
}

/* set up the clock */
fp = fopen(Outputfilename, "a");
time(&clock);
fprintf(fp,"Start-up time is :  %s\n",ctime(&clock));
fclose(fp);

/* generate random numbers */
randomize();

/* get the template */
get_template();

fscanf(finput,"Copies (10 1 1) =");
if(!(copies = (int *)malloc((max_era+1) * sizeof(int))))
{
        sprintf(msg,"Insufficient memory for variable, copies.\n");
        print_error(msg);
}

for (i = era; i <= max_era; i++) {
        copies[i] = 1; /* default */
        fscanf(finput,"%d",&copies[i]);
}
fclose(finput);


/* get the objective function information */
function_evaluations = 0.0;
objfunc_info();

/* get the auxiliary information for report */
if (partitionflag)
        partition_info();
```

```
        if (popprintflag)
                poprec_info();

        /* writes the problem information */
        problem_rep();
}


void initialize()
/* initialize parameters */
{
        char msg[PAGEWIDTH];

        if (thresholdingflag)
                shuffle_num = problem_length;

        if (popprintflag)
        {
                /* set the counter to the first generation */
                nextpopstatgen = sortpopstatgen[0];
        }

        /* get other input parameters */
        if (fscanf(fin, FORMAT_2, VARS_2) != 2) {
                sprintf(msg,"Error in input file era.\n");
                print_error(msg);
        }

        /* generate initial population, get stats, and write report */
        initialize_pop();
        statistics(oldpop);
        /* set up the population sizing in primordial phase */
        setup_popsize();
        initreport();
}


void initialize_pop()
/* setup parameters for initial population */
{
        long member_id = 0;
        int temp_order, allele_combo;
        long max_size, total_size = 0, population_size[MAX_ORDER];
        char msg[PAGEWIDTH];

        gen = 0;
        stopmgaflag = 0;
        /* reset the population size array */
```

```
reset_list(population_size);

/* only order era strings are created */
order = temp_order = era;

/* but if tiebreaking is on,
   all strings of order less than era are created */
if (tiebreakingflag) {
        order = temp_order = 1;
}
while (order <= era) {
        /* if reduction in initpop is desired */
        if (r_initpop_flag)
                allele_combo = 1;
        else
                allele_combo = power(2,order);
        max_size = copies[order] * allele_combo * \
                        choose(problem_length,order);
        population_size[order] = max_size;
        total_size += max_size;
        order++;
}
/* storage allocation for */
/* population array */
if(!(newpop=(struct INDIVIDUAL *)malloc(total_size * \
                        sizeof(struct INDIVIDUAL))))
{
        sprintf(msg,"Insufficient memory for newpop.\n");
        print_error(msg);
}
if(!(oldpop=(struct INDIVIDUAL *)malloc(total_size * \
                        sizeof(struct INDIVIDUAL))))
{
        sprintf(msg,"Insufficient memory for oldpop.\n");
        print_error(msg);
}
/* storage for shuffle array */
if (!(shuffle = (int *)malloc(total_size*sizeof(int *))))
{
        sprintf(msg,"Insufficient memory for shuffle array.\n");
        print_error(msg);
}
order = temp_order;
while (order <= era) {
        initpop(&member_id, population_size[order]);
        order++;
}
if (extrapopflag)
```

```
                extra_pop(&member_id);
        order--;
        storepop(&best_indv,era);
        copy_chrom(oldpop[0],&best_indv);

        popsize = member_id;
        templatefitness = objfunc(template);
}

void extra_pop(last_member)
long *last_member;
/* creates extra population members */
{
        register int i,j,k,m;
        int flag=0, count=0, extra_size=0;
        long size, cnt;
        int patsize[MAX_PARTITIONS], extra_copies[MAX_PARTITIONS];
        GENE_ID pattern[MAX_PARTITIONS][MAX_ORDER];
        ALLELES allele_list[MAX_ORDER];
        struct GENE *tailold;
        char dummy[PAGEWIDTH];
        FILE *fp, *fopen();

        fp = fopen(Extrafilename,"r");
        get_string(fp, dummy, PAGEWIDTH);
        /* read gene information */
        while (fscanf(fp,"%d",&patsize[count]) == 1) {
                for (i=0; i<patsize[count]; i++)
                        fscanf(fp,"%d",&pattern[count][i]);
                fscanf(fp,"%d\n",&extra_copies[count]);
                extra_size += extra_copies[count] * power(2,patsize[count]);
                count++;
        }
        fclose(fp);
        size = (*last_member) + extra_size;
        /* reallocate memory size and allocate memory for extra individuals */
        reallocate_memory(size);
        cnt = (*last_member);
        for (i=0; i<count; i++) {
                for (j=0; j<patsize[i]; j++)
                        allele_list[j] = 0;
                for (j=0; j<power(2,patsize[i]); j++) {
                        if (j != 0)
                                next_allele_comb(allele_list,patsize[i]);
                        for (k=0; k<extra_copies[i]; k++) {
                                storepop(&(oldpop[cnt]),patsize[i]);
                                storepop(&(newpop[cnt]),patsize[i]);
                                tailold = oldpop[cnt].firstgene;
```

```
                                tailold→genenumber = pattern[i][0];
                                tailold→allele = allele_list[0];

                                for (m=1; m<patsize[i]; m++) {
                                        tailold = tailold→nextgene;
                                        tailold→genenumber = pattern[i][m];
                                        tailold→allele = allele_list[m];
                                }
                                fill_chrom(&(oldpop[cnt]));
                                oldpop[cnt].fitness = \
                                        objfunc(oldpop[cnt].fullchrom);
                                cnt++;
                        }
                }
        }
        *last_member = cnt;
}


void randomtemplate(temp)
unsigned *temp;
/* create a random template */
{
        register int i,j;
        unsigned mask = 1;

        for (i = 0; i < bytesize; i++)
        {
                temp[i] = 0;
                for (j = 0; j < bytelimit[i]; j++)
                {
                        if (flip(0.5))
                                temp[i] |= mask;
                        if (j < bytelimit[i]-1)
                                temp[i] <<= 1;
                }
        }
}


void next_genic_comb(list, n)
GENE_ID *list;
int n;
/* calculates the next gene combination */
{
        register int j, s, k;
        BOOLEAN exitflag = 0;
        int pos;
```

```
        pos = n-1;
        if (list[pos] < problem_length - 1)
                (list[pos])++;
        else
        {
                for (j = pos; j > 0 && !exitflag; j--)
                {
                        if (list[j-1] < problem_length - pos + j - 2)
                        {
                                (list[j-1])++;
                                for (s = j; s <= pos; s++)
                                {
                                        list[s] = list[j-1] + s - j + 1;
                                        exitflag = 1;
                                }
                        }
                }
        }
}


void next_allele_comb(list,size)
unsigned *list;
int size;
/* calculates the next bit combination */
{
        register int i;
        int total=0, pos;
        ALLELES allele_max = 1;

        pos = size-1;
        for (i = 0; i < pos+1; i++)
        {
                if (list[i]) total++;
        }
        if (list[pos] < allele_max)
                (list[pos])++;
        else if (total == pos+1)
                reset_list(list);
        else if ((list[pos-1] >= allele_max) && (list[pos] >= allele_max))
                next_allele_comb(list, pos);
        else if (list[pos] >= allele_max)
        {
                for (i = pos; i < pos+1; i++)
                        list[i] = 0;
                (list[pos-1])++;
        }
```

```
}


void storepop(pop, n)
struct INDIVIDUAL *pop;
int n;
/* allocates memory for an individual */
{
        register int j;
        struct GENE *tail;
        char msg[PAGEWIDTH];

        /* storage allocation for */
        /* genes */
        if(!((*pop).firstgene = (struct GENE *)malloc(sizeof(struct GENE))))
        {
                sprintf(msg,"Insufficient memory for firstgene.\n");
                print_error(msg);
        }

        /* initialize genes */
        tail = (*pop).firstgene;

        /* for each gene */
        for (j = 1; j < n; j++)
        {
                if (!(tail→nextgene = (struct GENE *)malloc(sizeof(struct GENE))))
                {
                        sprintf(msg,"Insufficient memory for making a gene.\n");
                        print_error(msg);
                }
                tail = tail→nextgene;
        }

        /* assign last gene pointer and chromosome length */
        (*pop).lastgene = tail;
        tail→nextgene = NIL;
        (*pop).chromlen = n;
        (*pop).genelen = n;

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

        /* allocate storage for fullgene */
```

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


void initpop(last_member, n)
long *last_member;
long n;
/* generates initial population */
{
        register int i, j, m, count;
        long current_size;
        int allele_combo;
        GENE_ID genenumber_list[MAX_ORDER];
        unsigned allele_list[MAX_ORDER];
        struct GENE *tailold;
        char msg[PAGEWIDTH];

        sprintf(msg," Initpop entered for order %d",order);
        TRACE(msg);

        /* check if reduced population is desired */
        if (r_initpop_flag)
                allele_combo = 1;
        else
                allele_combo = power(2, order);
        current_size = (*last_member) + n;
        for (i= (*last_member); i < current_size; i++) {
                /* store oldpop and newpop */
                storepop(&(oldpop[i]),order);
                storepop(&(newpop[i]),order);
        }
        /* assign values */
        for (i = 0; i < order; i++)
                genenumber_list[i] = i;

        for (count=(*last_member); count < current_size; ) {
                for (i = 0; i < order; i++)
                        allele_list[i] = 0;
                for (j = 0; j < allele_combo; j++) {
                        if (j != 0)
                                /* get the next allele combination */
                                next_allele_comb(allele_list, order);

                        for (m = 0; m < copies[order]; m++)
```

```
                        {
                                tailold = oldpop[count].firstgene;
                                tailold→genenumber = genenumber_list[0];
                                if (r_initpop_flag)
                                        tailold→allele = mut( \
                                        pickallele(genenumber_list[0],
template));
                                else
                                        tailold→allele = allele_list[0];

                                for (i = 1; tailold→nextgene != NIL; i++) {
                                        tailold = tailold→nextgene;
                                        tailold→genenumber = genenumber_list[i];
                                        if (r_initpop_flag)
                                                tailold→allele = mut( \
                                                pickallele(genenumber_list[i],\
                                                                template));
                                        else
                                                tailold→allele = allele_list[i];
                                }

                                /* fill up the under-specified genes */
                                fill_chrom(&(oldpop[count]));

                                /* evaluate objective function value */
                                oldpop[count].fitness = \
                                        objfunc(oldpop[count].fullchrom);

                                count++;
                        }
                }
                /* get the next gene combination */
                next_genic_comb(genenumber_list, order);
        }
        *last_member = current_size;

        TRACE(" Initpop exited\n");
}


int mut(bit)
int bit;
/* returns a 1 if the bit is 0 and vice versa */
{
        return((bit == 1) ?  0 :  1);
}
```

```
void objfunc_info()
/* reads objective function information */
{
        register int i, j, k;
        ALLELES allele_max = 1;
        int sum, maxallele, bit, string_length, curr_len;
        int numtable, numstrings, tablenum;
        GENE_ID dummygene[MAX_ORDER];
        double perfm;
        char str[PAGEWIDTH], dummy[PAGEWIDTH], msg[PAGEWIDTH];
        FILE *fp, *fopen();

        fp = fopen(Objfilename,"r");

        /* read subfunction table information, skip a line */
        get_string(fp,dummy,PAGEWIDTH);

        maxallele = allele_max + 1;

        fscanf(fp,"%d\n",&numtable);

        if(!(taborfunc = (BOOLEAN *)malloc(numtable * sizeof(BOOLEAN))))
        {
                sprintf(msg,"Insufficient memory for taborfunc.\n");
                print_error(msg);
        }
        if(!(str_length = (int *)malloc(numtable * sizeof(int))))
        {
                sprintf(msg,"Insufficient memory for str_length.\n");
                print_error(msg);
        }
        if(!(chromfitness = (double **)malloc(numtable * sizeof(double *))))
        {
                sprintf(msg,"Insufficient memory for chromfitness.\n");
                print_error(msg);
        }

        get_string(fp,dummy,PAGEWIDTH);
        for (k = 0; k < numtable; k++)
        {
                /* read size of the subfunction */
                fscanf(fp,"%d\n",&tablenum);
                fscanf(fp,"%s\n",&str[0]);
                fscanf(fp,"%d\n",&string_length);
                str_length[tablenum] = string_length;
                taborfunc[tablenum] = 0;

                if ((strcmp(str,"Table") == 0) || (strcmp(str,"table") == 0))
```

```
                /* read a table, if the subfunction is a table */
                {
                        taborfunc[tablenum] = 1;
                        numstrings = power(maxallele,string_length);
                        if(!(chromfitness[tablenum] = (double *) \
                                        malloc(numstrings * sizeof(double))))
                        {
                                sprintf(msg,"Insufficient memory for
chromfitness.\n");
                                print_error(msg);
                        }

                        for (i = 0; i < numstrings; i++)
                        {
                                for (j = 0,sum = 0; j < string_length; j++)
                                {
                                        /* read each bit */
                                        fscanf(fp,"%d",&bit);
                                        sum += power(maxallele,\
                                                string_length-j-1) * bit;
                                }
                                /* read the corresponding objective function value */
                                fscanf(fp,"%lf\n",&perfm);
                                chromfitness[tablenum][sum] = perfm;
                        }
                }
                get_string(fp,dummy,PAGEWIDTH);
        }
        /* read subfunction information, skip a line */
        get_string(fp,dummy,PAGEWIDTH);

        /* read number of subfunctions */
        fscanf(fp,"%d",&numsubfunc);

        if(!(table_id = (int *)malloc(numsubfunc * sizeof(int))))
        {
                sprintf(msg,"Insufficient memory for table_id.\n");
                print_error(msg);
        }
        if(!(scale = (float *)malloc(numsubfunc * sizeof(float))))
        {
                sprintf(msg,"Insufficient memory for scale.\n");
                print_error(msg);
        }

        if(!(genesets = (GENE_ID **)malloc(numsubfunc * sizeof(GENE_ID *))))
        {
                sprintf(msg,"Insufficient memory for variable, genesets.\n");
```

```
                print_error(msg);
        }

        /* for each subfunction */
        for (i = 0; i < numsubfunc; i++)
        {
                /* read the table number */
                fscanf(fp,"\n%d",&table_id[i]);
                fscanf(fp,"%f",&scale[i]);

                curr_len = str_length[table_id[i]];
                read_subfunction(fp,dummygene,curr_len);
                if(!(genesets[i] = (GENE_ID *)malloc(curr_len * sizeof(GENE_ID))))
                {
                        sprintf(msg,"Insufficient memory for genes.\n");
                        print_error(msg);
                }
                for (j = 0; j < curr_len; j++)
                        genesets[i][j] = dummygene[j];
        }
        fclose(fp);
}

void read_subfunction(fp,dummygene,nstruc)
FILE *fp;
GENE_ID dummygene[];
int nstruc;
/* reads subfunction data */
{
        register int i;
        int count = 0, sets = 0, prev = 0;

        do {
                prev = sets;
                fscanf(fp,"%d",&sets);
                if (sets >= 0) {
                        dummygene[count] = sets;
                        count++;
                }
                else {
                        sets *= -1;
                        for (i = 0; i <= sets - prev; i++) {
                                dummygene[count+i] = prev + i + 1;
                        }
                        count += sets - prev;
                }
        } while (count < nstruc);
}
```

```
void partition_info()
/* reads partition information for analysis */
{
        register int i, j;
        FILE *fp, *fopen();
        GENE_ID dummygene[MAX_PARTITIONSIZE];
        char dummy[PAGEWIDTH], msg[PAGEWIDTH];

        fp = fopen(Partinfilename,"r");

        /* read a comment */
        get_string(fp, dummy, PAGEWIDTH);

        /* read the number of partitions */
        fscanf(fp,"%d",&numpartition);
        if(!(partition_len = (int *)malloc(numpartition * sizeof(int))))
        {
                sprintf(msg,"Insufficient memory for variable partition_len.\n");
                print_error(msg);
        }
        if(!(partition_genes = (GENE_ID **)malloc(numpartition * \
                                                        sizeof(GENE_ID *))))
        {
                sprintf(msg,"Insufficient memory for partition genes.\n");
                print_error(msg);
        }

        /* for each structure */
        for (i = 0; i < numpartition; i++)
        {
                fscanf(fp,"\n%d",&partition_len[i]);

                read_subfunction(fp,dummygene,partition_len[i]);

                if(!(partition_genes[i] = (GENE_ID *)malloc(partition_len[i] * \
                                                        sizeof(GENE_ID))))
                {
                        sprintf(msg,"Insufficient memory for partition
genes.\n");
                        print_error(msg);
                }

                for (j = 0; j < partition_len[i]; j++)
                        partition_genes[i][j] = dummygene[j];
        }
        fclose(fp);
```

```
}

void poprec_info()
/* reads generations for population record */
{
        register int i;
        int numpopstatgen;
        FILE *fp, *fopen();
        char msg[PAGEWIDTH];

        fp = fopen(Poprinfilename,"r");

        get_string(fp, msg, PAGEWIDTH);

        /* read the generation numbers for population dump */
        fscanf(fp,"%d\n",&numpopstatgen);

        if (!(sortpopstatgen = (int *)malloc(numpopstatgen*sizeof(int))))
        {
                sprintf(msg,"Insufficient memory for poprecfile.\n");
                print_error(msg);
        }

        for (i = 0; i < numpopstatgen; i++)
        {
                fscanf(fp,"%d",&(sortpopstatgen[i]));
        }

        /* sort the numbers in ascending order */
        sortnum(numpopstatgen, sortpopstatgen);

        fclose(fp);
}

void setup_popsize()
/* sets up the popsize pattern in primordial phase */
{
        int cal_cut, noofcut;
        double init_prop, init_prop_1, init_prop_2;

        init_select_gen = 1;
        /* initial proportion of the best individual */
        init_prop_1 = problem_length / era;
        init_prop_1 /= (popsize / copies[era]);

        init_prop_2 = prop_bestindv;

        /* choose the greater of two */
```

```
        init_prop = (init_prop_1 > init_prop_2) ?  init_prop_1 :  init_prop_2;
        /* duration of primordial phase: (under tournament selection) is
           calculated to have the proportion of best individual equals to 0.5 */
        prim_gen = 0;
        if (init_prop < 0.5)
                prim_gen = floor(log((1.0 - init_prop) / init_prop) / log(2.0));

        cal_cut = prim_gen - (init_select_gen - 1);

        if (popsize < juxtpopsize) juxtpopsize = popsize;
        /* theoretical number of cut from juxtpopsize and initial popsize */
        noofcut = round((log(popsize) - log(juxtpopsize))/log(2.0));

        if (cal_cut <= 0) {
                /* if no cut is required */
                cutpop_gen = 0;
                juxtpopsize = popsize;
                init_select_gen = 0;
        }
        else if (noofcut > cal_cut) {
                /* too many cut is requested */
                printf("Given juxtapositional popsize is too small %d, ",\
                                                        juxtpopsize);
                juxtpopsize = popsize/power(2,cal_cut);
                printf("using %d instead.\n",juxtpopsize);
                cut_every_other_gen = 0;
                cutpop_gen = prim_gen;
        }
        else if (noofcut >= cal_cut / 2) {
                /* some cut needs to be at every generation */
                cut_every_other_gen = init_select_gen - 1 + 2 * \
                                        (cal_cut - noofcut);
                cutpop_gen = prim_gen;
        }
        else {
                /* only cut at every other generation */
                cut_every_other_gen = init_select_gen - 1 + 2 * noofcut;
                cutpop_gen = cut_every_other_gen;
        }
}


void get_template()
/* get template information and create a template */
{
        register int i, j;
        unsigned mask = 1;
        int id, sum;
```

```
ALLELES dummytemplate[MAX_SIZE];
char msg[PAGEWIDTH];
FILE *fp, *fopen();

/* set bytesize */
id = problem_length/UNSINTSIZE;
bytesize = (problem_length % UNSINTSIZE) ? id+1 :  id;

/* set up bytelimit array */
if(!(bytelimit = (int *)malloc(bytesize*sizeof(int)))) {
        sprintf(msg, "Insufficient memory for bytelimit\n");
        print_error(msg);
}
for (i = 0; i < bytesize; i++)
{
        if (i == bytesize-1)
                bytelimit[i] = problem_length - (i*UNSINTSIZE);
        else
                bytelimit[i] = UNSINTSIZE;
}

/* memory for template is allocated */
if(!(template = (unsigned *)malloc(bytesize*sizeof(unsigned)))) {
        sprintf(msg, "Insufficient memory for template\n");
        print_error(msg);
}

if (levelmgaflag)
{
        era = 1;
        /* get a random template */
        randomtemplate(template);
}
else
{
        era = max_era;

        /* read template */
        check_input_file(Templatefilename);
        fp = fopen(Templatefilename,"r");
        for (i = 0; i < problem_length; i++)
                fscanf(fp,"%d",&dummytemplate[i]);
        fclose(fp);

        /* make template */
        for (i = 0,sum = 0; i < bytesize; i++)
        {
                sum += bytelimit[i];
```

```
                        template[i] = 0;
                        for (j = 0; j < bytelimit[i]; j++)
                        {
                                if (dummytemplate[sum - 1 - j] == 1)
                                        template[i] |= mask;
                                if (j < bytelimit[i] - 1)
                                        template[i] <<= 1;
                        }
                }
        }
}
```

11.5 The file generate.c

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

   purpose : new population generation

 developed : 1991

    author : Kalyanmoy Deb
==================================================*/

#include "mga.ext"


long change_popsize()
/* find out popsize */
{
        long size;

        size = popsize;
        if ((gen >= init_select_gen) && (gen <= cutpop_gen))
        {
                if ((gen == cutpop_gen) && (size > juxtpopsize))
                        size = juxtpopsize;
                else if (gen <= cut_every_other_gen)
                /* cut at every other generation */
                {
                        if (cutpopflag == 1)
                        {
                                size = popsize / 2;
                                cutpopflag = 0;
                        }
                        else
                                cutpopflag = 1;
                }
                else /* cut at every generation */
                        size = popsize / 2;
        }
        return(size);
}


void primordial()
/* primordial phase */
{
        int pos,count;
        char msg[PAGEWIDTH];
        long newpopsize;
```

```
        TRACE(" Primordial entered");

        /* check whether population size needs to be changed */
        newpopsize = change_popsize();
        /* shuffle the population */
        pick = 0;
        shuffle_pop();
        count = 0;
        /* create a new population */
        while (count < newpopsize) {
                /* select an individual */
                pos = select_pop();

                /* copy it in the new population */
                copy_chrom(oldpop[pos],&(newpop[count]));

                count++;
        }
        popsize = newpopsize;
        reallocate_memory(popsize);

        TRACE(" Primordial exited\n");
}


void juxtapositional()
/* juxtapositional phase */
{
        int pos1, pos2, count;
        register int i;
        struct INDIVIDUAL mate1, mate2;

        TRACE(" Juxtapositional entered");

        /* shuffle the population */
        pick = 0;
        shuffle_pop();
        count = 0;
        /* free memory allocated for new population */
        freenewpop();
        /* create new population */
        do {
                /* generate mate1 and mate2 */
                storepop(&mate1,1);

                /* select two individuals for mating */
                pos1 = select_pop();
```

```
                copy_chrom(oldpop[pos1],&mate1);

                storepop(&mate2,1);
                pos2 = select_pop();
                copy_chrom(oldpop[pos2],&mate2);

                /* cut and splice two individuals */
                cut_and_splice(mate1,mate2);

                /* put children individuals in new population */
                while (count < popsize && !  isempty(newchrom_stack)) {
                        if (pop_stack(&newchrom_stack,&(newpop[count])))
                        {
                                /* perform mutation */
                                mutation(&(newpop[count]));

                                /* find out raw chrom */
                                fill_chrom(&(newpop[count]));

                                /* evaluate objective function value */
                                newpop[count].fitness = \
                                        objfunc(newpop[count].fullchrom);

                                count++;
                        }
                }
        } while (count < popsize);

        TRACE(" Juxtapositional exited\n");
}

void generate()
/* messy GA operation */
{
        struct INDIVIDUAL *tempop;

        printf("\nera = %d, generation = %d\n\n",era,gen);
        TRACE("Generate entered");

        if (gen <= prim_gen)
                primordial();
        else
                juxtapositional();

        statistics(newpop);
        reportpop(newpop);

        /* copy newpop to oldpop */
```

```
        tempop = oldpop;
        oldpop = newpop;
        newpop = tempop;

        TRACE("Generate exited");
}
```

11.6 The file operators.c

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

     purpose : messy GA operators

   developed : 1991

      author : Kalyanmoy Deb
=========================================================*/

#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 += 2;
        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;
        struct INDIVIDUAL chr3;
        struct INDIVIDUAL chr4;

        /* if first chromosome needs to be cut */
        if (flip(cut_prob * chr1.chromlen))
        {
                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))
                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)
                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);

        /* 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);
}
```

11.7 The file supports.c

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

     purpose : auxiliary files

   developed : 1991

      author : Kalyanmoy Deb
==========================================================*/

#include "mga.ext"


int length_chrom(chr)
struct GENE *chr;
/* calculates the length of a chromosome */
{
        int len = 0;
        struct GENE *temp;

        temp = chr;
        /* NIL chrom has a zero length */
        if (temp != NIL)
        {
                for (len=1; temp->nextgene != NIL; )
                {
                        len++;
                        temp = temp->nextgene;
                }
        }
        return(len);
}


void fill_chrom(chr)
struct INDIVIDUAL *chr;
/* fills up the remaining genes from the template */
{
        register int i;
        BOOLEAN gene_flag[MAX_SIZE];
        struct GENE *temp;
        int id, rem, temp_gene, count = 0;
        unsigned temp_allele, shield, tempmask_gene, tempmask = 1;
        unsigned *rawchr = (*chr).fullchrom;
        unsigned *rawgn = (*chr).fullgene;

        /* no gene has been considered yet */
```

```
        for (i = 0; i < problem_length; i++)
                gene_flag[i] = 0;

        /* assign the template alleles first */
        for (i = 0; i < bytesize; i++)
        {
                rawchr[i] = template[i];
                rawgn[i] = 0;
        }
        temp = (*chr).firstgene;

        /* for each member in the chromosome */
        while (temp != NIL) {
                temp_gene = temp→genenumber;
                temp_allele = temp→allele;

                /* if not replaced already, (left to right scan) */
                if (!  gene_flag[temp_gene])
                {
                        id = temp_gene/UNSINTSIZE;
                        rem = temp_gene % UNSINTSIZE;
                        shield = tempmask ≪ rem;

                        /* replace the allele, if different */
                        if (((rawchr[id] ≫ rem) & tempmask) != temp_allele)
                                rawchr[id] = shield ∧ rawchr[id];

                        /* make fullgene */
                        tempmask_gene = (0 ≫ rem) | 1;
                        tempmask_gene ≪= rem;
                        rawgn[id] |= tempmask_gene;
                        count++;

                        /* set the allele flag off */
                        gene_flag[temp_gene] = 1;
                }
                temp = temp→nextgene;
        }
        (*chr).genelen = count;
}


void copy_chrom(chr1,chr2)
struct INDIVIDUAL chr1,*chr2;
/* copy chr into another variable whose firstgene has been allocated */
{
        register int i,j;
        struct GENE *temp, *tchr;
```

```
        char msg[PAGEWIDTH];

        /* copy genes */
        temp = (*chr2).firstgene;
        tchr = chr1.firstgene;
        temp→genenumber = tchr→genenumber;
        temp→allele = tchr→allele;
        for (i = 1; i < chr1.chromlen; i++)
        {
                if (temp→nextgene == NIL)
                {
                        if(!(temp→nextgene = (struct GENE *) \
                                                malloc(sizeof(struct GENE))))
                        {
                                sprintf(msg,"Insufficient memory for genes\n");
                                print_error(msg);
                        }
                        temp→nextgene→nextgene = NIL;
                }
                temp = temp→nextgene;
                tchr = tchr→nextgene;
                temp→genenumber = tchr→genenumber;
                temp→allele = tchr→allele;
        }

        /* copy other members */
        (*chr2).lastgene = temp;
        temp→nextgene = NIL;
        (*chr2).chromlen = chr1.chromlen;
        for (i = 0; i < bytesize; i++)
        {
                (*chr2).fullchrom[i] = chr1.fullchrom[i];
                (*chr2).fullgene[i] = chr1.fullgene[i];
        }
        (*chr2).genelen = chr1.genelen;
        (*chr2).fitness = chr1.fitness;
}


BOOLEAN isempty(stack)
STACKPTR stack;
/* Is the stack empty? */
{
        return (stack == NIL);
}


BOOLEAN pop_stack(stack, chr)
```

```
STACKPTR *stack;
struct INDIVIDUAL *chr;
/* pops the top member in the stack; sends 1 if there is a member,
    0 otherwise */
{
        STACKPTR t = *stack;

        if (!  isempty(t))
        {
                *chr = t→genefirst;
                *stack = t→nextmem;
                free(t);
                return(1);
        }
        else
                return(0);
}


void push_stack(stack, chr)
STACKPTR *stack;
struct INDIVIDUAL chr;
/* push a member chr into the stack */
{
        STACKPTR temp;
        char msg[PAGEWIDTH];

        if(!(temp = (STACKPTR)malloc(sizeof(struct STACKTYPE))))
        {
                sprintf(msg,"Insufficient memory for stack\n");
                print_error(msg);
        }
        temp→genefirst = chr;
        temp→nextmem = *stack;
        *stack = temp;
}


ALLELES pickallele(id,rawchr)
int id;
unsigned *rawchr;
/* picks the allele value of the id-th gene from rawchr */
{
        unsigned mask = 1;
        int num, rem;

        num = id / UNSINTSIZE;
        rem = id % UNSINTSIZE;
```

```
        if (((rawchr[num] >> rem) & mask) == 1)
                return(1);
        else
                return(0);
}


void shuffle_pop()
/* shuffles the shuffle array */
{
        register int i;
        int num;
        INDV_ID temp;

        for (i = 0; i < popsize; i++)
                shuffle[i] = i;
        for (i = 0; i < popsize-1; i++)
        {
                num = rnd(i, popsize-1);
                temp = shuffle[num];
                shuffle[num] = shuffle[i];
                shuffle[i] = temp;
        }
}


void assign_beststring_to_template()
/* assigns the best string to the template */
{
        register int i;
        long clock,time();
        char *ctime();
        FILE *fp, *fopen();

        for (i = 0; i < bytesize; i++)
                template[i] = best_indv.fullchrom[i];
        templatefitness = best_indv.fitness;

        /* write the best string in output file */
        fp = fopen(Outputfilename,"a");
        fprintf(fp,"\nThe best string:\n");
        writefullchrom(fp,best_indv.fullchrom);
        fprintf(fp,"Fitness = %10.3lf\n",best_indv.fitness);
        fprintf(fp,"String length = %5d\n",best_indv.chromlen);
        fprintf(fp,"Chromosome:  ");
        fwritechrom(fp,best_indv,14);

        time(&clock);
```

```
        fprintf(fp,"\nEnd of run for era %d:  %s\n",era,ctime(&clock));
        if (era != max_era) {
                page(fp);
                fprintf(fp,"\n");
        }
        fclose(fp);
}
```

11.8 The file functions.c

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

     purpose : define objective functions

   developed : 1991

      author : Kalyanmoy Deb
==================================================*/

#include "mga.ext"


double file0(chr,len)
int chr[];
int len;
/* Liepins and Vose's fully deceptive function */
{
        double x,val;
        int d;

        d = ones(chr,len);
        if (d == 0) /* local solution: all 0's */
                x = 1.0 - 1.0/(2.0*len);
        else if (d == len) /* global solution: all 1's */
                x = 1.0;
        else /* otherwise */
                x = 1.0 - (1.0 + d)/len;
        return((double) x * len);
}



double file1(chr,len)
int chr[];
int len;
/* Ackley's one max problem */
{
        double x,val;

        x = ones(chr,len);
        return((double) 10*x);
}



double file2(chr,len)
```

```
int chr[];
int len;
/* Ackley's two max function */
{
        double x,val;
        float fabs();

        x = ones(chr,len);
        val = fabs(18.0 * x - 8.0 * len);
        return((double) val);
}



double file3(chr,len)
int chr[];
int len;
/* Ackley's trap function */
{
        double x,z,val;

        x = ones(chr,len);
        z = 0.75 * len;
        if (x <= z)
                val = 8 * len * (z-x) / z;
        else
                val = 10 * len * (x-z) / (len-z);
        return((double) val);
}

double file4(chr,len)
int chr[];
int len;
/* Ackley's Porcupine function */
{
        double x,val;

        x = ones(chr,len);
        val = 10 * x - 15 * (1 - parity(x));
        return((double) val);
}


double file5(chr,len)
int chr[];
int len;
/* Ackley's Plateaus function */
{
```

```
        double x, val, z;
        float fabs();
        int ones(), parity();

        x = ones(chr,len);
        if (x >= 1.0*len)
                val = 2.5 * len;
        else val = 0.0;
        return((double) val);
}

double file6(chr,len)
int chr[];
int len;
/* User defined function */
{
        double x,val;

        x = decode(chr,len);
        val = x;
        return((double) val);
}

double file7(chr,len)
int chr[];
int len;
/* User defined function */
{
        double x,val;

        x = decode(chr,len);
        val = x;
        return((double) val);
}

double file8(chr,len)
int chr[];
int len;
/* User defined function */
{
        double x,val;

        x = decode(chr,len);
        val = x;
        return((double) val);
}

double file9(chr,len)
```

```
int chr[];
int len;
/* User defined function */
{
        double x,val;

        x = decode(chr,len);
        val = x;
        return((double) val);
}

double get_func_val(id, chr, len)
int id;
unsigned chr[];
int len;
{
        switch (id) {
        case 0:
                return(file0(chr,len));
                break;
        case 1:
                return(file1(chr,len));
                break;
        case 2:
                return(file2(chr,len));
                break;
        case 3:
                return(file3(chr,len));
                break;
        case 4:
                return(file4(chr,len));
                break;
        case 5:
                return(file5(chr,len));
                break;
        case 6:
                return(file6(chr,len));
                break;
        case 7:
                return(file7(chr,len));
                break;
        case 8:
                return(file8(chr,len));
                break;
        case 9:
                return(file9(chr,len));
                break;
        default:
```

```
                return(0.0);
        }
}
```

11.9 The file objfunc.c

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

     purpose : calculate objective function value

   developed : 1991

      author : Kalyanmoy Deb
==========================================================*/

#include "mga.ext"

double objfunc(rawchr)
unsigned *rawchr;
/* evaluate the objective function value of chr */
{
        register int j, k;
        double obj;
        int powerof2, sum, len;
        ALLELES subfunc_chrom[MAX_SIZE];
        FILE *fp, *fopen();

        /* for each subfunction */
        for (k = 0,obj = 0.0; k < numsubfunc; k++)
        {
                len = str_length[table_id[k]];
                if (taborfunc[table_id[k]] == 1) {
                        /* if a table */
                        powerof2=power(2,len-1);
                        /* calculate objective function value */
                        for (j = 0, sum = 0; j < len; j++) {
                                sum += powerof2 * pickallele(genesets[k][j], \
                                                                rawchr);
                                powerof2 /= 2;
                        }
                        obj += scale[k] * chromfitness[table_id[k]][sum];
                }
                else /* if a function */
                {
                        for (j = 0; j < len; j++) /* get the binary string */
                                subfunc_chrom[j] = pickallele(genesets[k][j], \
                                                                rawchr);

                        /* use a function defined in functions.c file */
                        obj += scale[k] * get_func_val(table_id[k], \
                                                        subfunc_chrom,len);
                }
```

```
        }
        function_evaluations += 1.0;
        return(obj);
}
```

11.10 The file stats.c

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

   purpose : calculate statistics

 developed : 1991

    author : Kalyanmoy Deb
==========================================================*/

#include "mga.ext"


void statistics(pop)
struct INDIVIDUAL *pop;
/* calculates statistics of the new population */
{
        register int i,j;
        double sumfitness, fit, totlen;
        long stringlen;
        int maxid, minid, whether, ifbetter;
        int maxnum, minnum, num, best_id, abovetemplate = 0;
        char msg[PAGEWIDTH];
        FILE *fp, *fopen();

        TRACE(" Statistics entered");

        maxnum = minnum = maxid = minid = 0;
        stringlen = pop[0].chromlen;
        sumfitness = minfitness = maxfitness = pop[0].fitness;
        for (i = 1; i < popsize; i++)
        {
                fit = pop[i].fitness;
                sumfitness += fit;
                if (gen == 0)
                { /* proportion of best indv at initial pop */
                        ifbetter = (maximizationflag) ?  (fit > templatefitness)\
                                   :  (fit < templatefitness);
                        if (ifbetter)
                                abovetemplate++;
                }
                stringlen += pop[i].chromlen;
                /* calculate minimum fitness */
                if (minfitness > fit)
                {
                        minfitness = fit;
                        minid = i;
```

```
                minnum = 1;
        }
        else if (minfitness == fit)
                minnum++;
        /* calculate maximum fitness */
        if (maxfitness < fit)
        {
                maxfitness = fit;
                maxid = i;
                maxnum = 1;
        }
        else if (maxfitness == fit)
                maxnum++;
}
avgfitness = sumfitness/popsize;

/* calculate average string length */
totlen = stringlen;
avgstrlen = totlen / popsize;

/* determine the best individual and replace previous best */
best_id = (maximizationflag) ?  maxid :  minid;
whether = (maximizationflag) ?  pop[best_id].fitness > best_indv.fitness\
          :  pop[best_id].fitness < best_indv.fitness;
if (whether)
{
        delete_chrom(best_indv.firstgene);
        if(!(best_indv.firstgene = (struct GENE *) \
                                        malloc(sizeof(struct GENE))))
        {
                sprintf(msg,"Insufficient memory for best_indv genes.\n");
                print_error(msg);
        }
        best_indv.firstgene→nextgene = NIL;
        copy_chrom(pop[best_id],&best_indv);
}
/* evaluate the proportion of the individuals with fitness
   better than the template */
if (gen == 0)
{
        if (abovetemplate == 0) /* no indv. better than template */
                stopmgaflag = 2;
        prop_bestindv = abovetemplate;
        prop_bestindv /= popsize;
}

/* check for convergence */
num = (maximizationflag) ?  maxnum :  minnum;
```

```
        if ((num >= stopfactor * popsize) && (gen > prim_gen))
                stopmgaflag = 1;

        TRACE(" Statistics exited\n");
}
```

11.11 The file report.c

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

     purpose : write reports

   developed : 1991

      author : Kalyanmoy Deb
==================================================*/

#include "mga.ext"


void reportpop(pop)
struct INDIVIDUAL *pop;
/* controls the output files */
{
        register int i,j;

        TRACE(" Report entered");

        general_rep();

        if (plotstatflag)
                plot_rep();

        if (popprintflag) {
                if (gen == nextpopstatgen) {
                        writepop(pop);
                        countpopstatgen++;
                        nextpopstatgen = *(sortpopstatgen + countpopstatgen);
                }
        }

        if (partitionflag)
                partitionprop(pop);

        printf(" Best String so far:  ");
        writefullchrom(stdout,best_indv.fullchrom);
        printf(" fitness:  %lf\n",best_indv.fitness);

        TRACE(" Report exited");
}


void general_rep()
/* writes the population statistics */
```

```
{
        FILE *fp, *fopen();

        fp = fopen(Outputfilename,"a");
        fprintf(fp,"%-3d %6d %7.2lf %9.2lf %9.2lf %9.2lf %14.4e\n",\
           gen,popsize,avgstrlen,maxfitness,minfitness,avgfitness,\
           function_evaluations);
        if (stopmgaflag == 1) {
                fprintf(fp," Termination:  convergence of the population\n");
        }
        else if (stopmgaflag == 2) {
                fprintf(fp," Termination:  no individual in the initial \
                                        population is better \n");
                fprintf(fp," than the template\n");
        }
        if (gen == prim_gen) {
                repchar(fp,"-",23);
                fprintf(fp," end of primordial phase ");
                repchar(fp,"-",23);
                fprintf(fp,"\n");
        }
        fclose(fp);
}


void plot_rep()
/* writes population statistics for plotting purpose */
{
        FILE *fp, *fopen();

        fp = fopen(Plotfilename,"a");
        fprintf(fp,"%-3d %6d %7.2lf %9.2lf %9.2lf %14.4e\n",\
               gencount,popsize,avgstrlen,best_indv.fitness,avgfitness,\
               function_evaluations);
        gencount++;
        fclose(fp);
}


void fwritechrom(fp,chr,skipchar)
FILE *fp;
struct INDIVIDUAL chr;
int skipchar;
/* chr is the pointer of the firstgene of a chromosome */
{
        struct GENE *tail;
        int count = 0;
        int limit;
```

```
        limit = PAGEWIDTH - skipchar - 8;
        tail = chr.firstgene;
        fprintf(fp,"( ");
        if (tail != NIL) {
                do {
                        if (count >= limit) {
                                count = 0;
                                fprintf(fp,"\n");
                                skip_space(fp,skipchar);
                        }
                        count += 8;
                        fprintf(fp,"(%-3d %1d) ", tail→genenumber, \
                                                  tail→allele);
                } while ((tail = tail→nextgene) != NIL);
        }
        fprintf(fp,")\n");
}


void writefullchrom(fp,chr)
FILE *fp;
unsigned *chr;
/* Write a chromosome as a string of 1's (true's) and 0's (false's) */
{
        register int j,k;
        unsigned mask = 1;
        unsigned temp;

        for(k = 0; k < bytesize; k++) {
                temp = chr[k];
                for(j = 0; j < bytelimit[k]; j++) {
                        if((temp & mask) == 0)
                                fprintf(fp,"0");
                        else
                                fprintf(fp,"1");
                        temp >>= 1;
                }
                fprintf(fp," ");
        }
        fprintf(fp,"\n");
}


void problem_rep()
/* writes the problem information */
{
        register int i,j;
```

```
        FILE *fp, *fopen();

        fp = fopen(Outputfilename,"a");
        fprintf(fp,"Problem length = %d\n",problem_length);
        fprintf(fp,"Number of subfunctions = %d\n",numsubfunc);
        fprintf(fp,"Subfunction = ");
        for (i = 0; i < numsubfunc; i++)
                fprintf(fp,"%3d ",i);
        fprintf(fp,"\n");
        fprintf(fp,"Table or function id = ");
        for (i = 0; i < numsubfunc; i++)
                fprintf(fp,"%3d ",table_id[i]);
        fprintf(fp,"\n");
        fprintf(fp,"Subfunction size = ");
        for (i = 0; i < numsubfunc; i++)
                fprintf(fp,"%3d ",str_length[table_id[i]]);
        fprintf(fp,"\n");
        fprintf(fp,"Constitutive Genes:\n");
        for (i = 0; i < numsubfunc; i++) {
                fprintf(fp," Subfunction %3d = ",i);
                for (j = 0; j < str_length[table_id[i]]; j++)
                        fprintf(fp,"%3d ",genesets[i][j]);
                fprintf(fp,"\n");
        }
        fprintf(fp,"\n");
        fprintf(fp,"Splice probability = %lf\n",splice_prob);
        fprintf(fp,"Cut probability = %lf\n",cut_prob);
        fprintf(fp,"Allelic mutation prob.  = %lf\n",allelic_mut_prob);
        fprintf(fp,"Genic mutation prob.  = %lf\n\n",genic_mut_prob);
        fprintf(fp,"Thresholding (1 if on) = %d\n",thresholdingflag);
        fprintf(fp,"Tiebreaking (1 if on) = %d\n\n",tiebreakingflag);
        fprintf(fp,"Random seed = %lf\n",randomseed);
        repchar(fp,"-",PAGEWIDTH);
        fprintf(fp,"\n\n");
        fclose(fp);
}

void initreport()
/* makes an initial report */
{
        register int i,j;
        FILE *fp, *fopen();

        fp = fopen(Outputfilename,"a");
        fprintf(fp,"Era = %d\n",era);
        fprintf(fp,"Init_select_gen = %d\n",init_select_gen);
        fprintf(fp,"Cutpop_gen = %d\n",cutpop_gen);
        fprintf(fp,"Primordial phase = %d\n",prim_gen);
```

```
        fprintf(fp,"Initial popsize = %d\n",popsize);
        fprintf(fp,"Popsize in juxt.  phase = %d\n",juxtpopsize);
        fprintf(fp,"Maximum generation = %d\n\n",maxgen);
        fprintf(fp,"Template = ");
        writefullchrom(fp,template);
        fprintf(fp,"\n");

        repchar(fp,"*",31);
        fprintf(fp," era = %d ",era);
        repchar(fp,"*",31);
        fprintf(fp,"\n");

        fprintf(fp,\
        "Gen Population Average Maximum Minimum Average Fuction\n");
        fprintf(fp,\
        " size str.  len.  objfunc objfunc objfunc evaluations\n\n");
        fclose(fp);

        if (partitionflag) {
                fp = fopen(Partoutfilename,"a");
                fprintf(fp,"Number of structures = %d\n",numpartition);
                fprintf(fp,"Number Size Partitions\n");
                for (i = 0; i < numpartition; i++) {
                        fprintf(fp," %3d %3d ",i,partition_len[i]);
                        for (j = 0; j < partition_len[i]; j++)
                                fprintf(fp," %3d",partition_genes[i][j]);
                        fprintf(fp,"\n");
                }
                fclose(fp);
        }

        reportpop(oldpop);
}


void writepop(pop)
struct INDIVIDUAL *pop;
/* writes all population members */
{
        long i;
        FILE *fp, *fopen();

        fp = fopen(Poproutfilename,"a");
        fprintf(fp,"Era = %d\n",era);
        fprintf(fp,"Generation = %d\n",gen);
        fprintf(fp,"Population size = %ld\n\n",popsize);
        fprintf(fp,"Number Obj.  Function string Chromosome\n");
        fprintf(fp," value length \n");
```

```
        for (i = 0; i < popsize; i++) {
                fprintf(fp," %-5ld ",i);
                fprintf(fp,"%10.3lf ",pop[i].fitness);
                fprintf(fp," %5d ",pop[i].chromlen);
                fwritechrom(fp,pop[i],32);
        }
        fprintf(fp,"\n\n");
        fclose(fp);
}


void partitionprop(pop)
struct INDIVIDUAL *pop;
/* calculates proportion of all strings */
{
        register int i,j,k;
        int powerof2,sum[MAX_PARTITIONS],numgenetable[MAX_PARTITIONS];
        int len, count, limit;
        BOOLEAN gene_flag[MAX_SIZE];
        ALLELES temp_allele, allele_max = 1;
        GENE_ID sub_list[MAX_PARTITIONS][MAX_PARTITIONSIZE], temp_gene;
        struct GENE *temp;
        FILE *fp, *fopen();

        /* initialize proportion array */
        for (i = 0; i < numpartition; i++) {
                for (j = 0; j < power(2,partition_len[i]); j++)
                        sub_list[i][j] = 0;
        }

        /* for each population member */
        for (i = 0; i < popsize; i++) {
                /* for each partition */
                for (j = 0; j < numpartition; j++) {
                        /* first initialize string counter and number of gene entry */
                        sum[j] = 0;
                        numgenetable[j] = 0;
                }
                /* then, for each gene along the problem length */
                for (j = 0; j < problem_length; j++) {
                        /* set no gene has been considered yet */
                        gene_flag[j] = 0;
                }
                /* Starting from the first gene */
                temp = pop[i].firstgene;
                while (temp != NIL) { /* till the last gene */
                        temp_gene = temp→genenumber;
                        temp_allele = temp→allele;
```

```
                /* check whether the gene has been considered already */
                if (!gene_flag[temp_gene]) {
                        gene_flag[temp_gene] = 1;

                        /* for each partition */
                        for (j = 0; j < numpartition; j++) {
                                len = partition_len[j];
                                powerof2 = power(2,len - 1);
                                for (k = 0; k < len; k++) {
                                /* if gene matches with a gene in the partition */

                                        if (temp_gene == \
                                                partition_genes[j][k]) {
                                        /* if yes, increment matched gene \
                                        counter for the partition and \
                                        set up the array */
                                                numgenetable[j]++;
                                                sum[j] += powerof2 * \
                                                        temp_allele;
                                        }
                                        powerof2 /= 2;
                                }
                        }
                }
                /* consider the next gene */
                temp = temp→nextgene;
        }

        /* for each partition */
        for (j = 0; j < numpartition; j++) {
                /* if all members of the partition has been found */
                if (numgenetable[j] == partition_len[j]) {
                        /* increment the right array */
                        sub_list[j][sum[j]]++;
                }
        }
}

/* write proportion of members of each structure */

fp = fopen(Partoutfilename,"a");
fprintf(fp,"\nEra %d, Generation %d, Population size %ld\n",\
              era, gen, popsize);
fprintf(fp,"Partition Number of strings in the population\n");

limit = PAGEWIDTH - 24;
/* for each partition */
```

```
        for (i = 0; i < numpartition; i++) {
                fprintf(fp," %3d ",i);
                /* print proportion of all members in the partition */
                for (j = 0,count = 0; j < power(allele_max+1, partition_len[i]);\
                        j++)
                {
                        if (count > limit) {
                                count = 7;
                                fprintf(fp,"\n");
                                skip_space(fp, 19);
                        }
                        count += 7;
                        fprintf(fp,"%-6d ",sub_list[i][j]);
                }
                fprintf(fp,"\n");
        }
        fclose(fp);
}
```

11.12 The file main

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

     purpose : main file

   developed : 1991

      author : Kalyanmoy Deb
==================================================*/

#include "mga.h"


main()
{
        FILE *fopen();

        /* get input parameters and create template */
        get_input();

        /* open input file for era dependent information */
        fin = fopen(Erafilename,"r");

        /* for each era, till the maximum era */
        while (era <= max_era)
        {
                printf("BEGIN ERA %d .  .  .\n",era);
                /* initialize variables and create initial population */
                initialize();

                /* GA loop */
                while (++gen <= maxgen && !  stopmgaflag) {
                        generate();
                }

                /* The best string so far is assigned to the template
                   for next era */
                assign_beststring_to_template();

                /* free all memory stored for variables */
                freeallmemory();

                printf("END ERA %d .  .  .\n\n",era);

                /* increment era */
                era++;
        }
```

```
        fclose(fin);
}
```

11.13 The file utility

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

    purpose : utility funations used in other files

  developed : 1991

     author : Kalyanmoy Deb
=====================================================*/

#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 starting at pointer temp */
{
        if (chrom != NIL)
        {
                delete_chrom(chrom→nextgene);
                free(chrom);
        }
}


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;
                }
                free(newpop[i].fullchrom);
                free(newpop[i].fullgene);
        }
}

void reallocate_memory(size)
/* reallocates storage for arrays, used only in primordial phase */
{
        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)
int *list;
/* resets the elements of list to zero */
{
        register int i;

        for (i = 0; i < strlen(list); 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);
}
```

11.14 The file random.c

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

     purpose : random number generator

  developed : 1991

      author : Kalyanmoy Deb

   (Translated in C from David Goldberg's SGA code in Pascal)
==========================================================*/

#include "mga.ext"


void advance_random()
/* Create next batch of 55 random numbers */
{
        register int j;
        double new_random;

        for(j = 0; j < 24; j++) {
                new_random = oldrand[j] - oldrand[j+31];
                if(new_random < 0.0)
                        new_random = new_random + 1.0;
                oldrand[j] = new_random;
        }
        for(j = 24; j < 55; j++) {
                new_random = oldrand [j] - oldrand [j-24];
                if(new_random < 0.0)
                        new_random = new_random + 1.0;
                oldrand[j] = new_random;
        }
}

void warmup_random(random_seed)
double random_seed;
/* Get random off and runnin */
{
        register int i, j;
        double new_random, prev_random;
        void advance_random();

        oldrand[54] = random_seed;
        new_random = 0.000000001;
        prev_random = random_seed;
```

```
        for(j = 1 ; j <= 54; j++) {
                i = (21 * j) % 54;
                oldrand[i] = new_random;
                new_random = prev_random-new_random;
                if(new_random < 0.0)
                        new_random = new_random + 1.0;
                prev_random = oldrand[i];
        }
        advance_random();
        advance_random();
        advance_random();
        jrand = 0;
}

double random()
/* Fetch a single random number between 0.0 and 1.0 - Subtractive Method */
/* See Knuth, D. (1969), v. 2 for details */
{
        void advance_random();

        jrand++;
        if(jrand >= 55) {
                jrand = 1;
                advance_random();
        }
        return(oldrand[jrand]);
}


BOOLEAN flip(prob)
double prob;
/* Flip a biased coin - true if heads */
{
        return (random() <= prob ?  1 :  0);
}

int rnd(low, high)
int low, high;
/* Pick a random integer between low and high */
{
        int i;
        double random();

        if (low >= high)
                i = low;
        else {
                i = (random() * (high - low + 1)) + low;
                if(i > high)
```

```
                        i = high;
        }
        return(i);
}

void randomize()
/* Get seed number for random and start it up */
{
        register int j;

        oldrand = (double *)malloc(55 * sizeof(double));
        do {
                for (j = 0; j <= 54; j++)
                        oldrand[j] = 0.0;
                jrand=0;
                printf("\nEnter seed random number (0.0..1.0)-> ");
                scanf("%lf", &randomseed);
        } while ((randomseed < 0.0) || (randomseed > 1.0));
        warmup_random(randomseed);
}
```

11.15 The file Makefile

```
#
# Makefile for MGA-C
#
CC=cc
CFLAGS= -O -w -p -lm -g

mga:  main.o generate.o initial.o report.o operators.o objfunc.o functions.o \
        stats.o supports.o utility.o random.o
        $(CC) $(CFLAGS) -o mga main.o generate.o initial.o report.o \
                        operators.o objfunc.o functions.o stats.o \
                        supports.o utility.o random.o

main.o:  main.c mga.h mga.ext mga.def
        $(CC) $(CFLAGS) -c main.c

generate.o:  generate.c mga.ext mga.def
        $(CC) $(CFLAGS) -c generate.c

initial.o:  initial.c mga.ext mga.def
        $(CC) $(CFLAGS) -c initial.c

report.o:  report.c mga.ext mga.def
        $(CC) $(CFLAGS) -c report.c

operators.o:  operators.c mga.ext mga.def
        $(CC) $(CFLAGS) -c operators.c

objfunc.o:  objfunc.c mga.ext mga.def
        $(CC) $(CFLAGS) -c objfunc.c

functions.o:  functions.c mga.ext mga.def
        $(CC) $(CFLAGS) -c functions.c

stats.o:  stats.c mga.ext mga.def
        $(CC) $(CFLAGS) -c stats.c

supports.o:  supports.c mga.ext mga.def
        $(CC) $(CFLAGS) -c supports.c

utility.o:  utility.c
        $(CC) $(CFLAGS) -c utility.c

random.o:  random.c mga.ext
        $(CC) $(CFLAGS) -c random.c

mga.h:  mga.def
```

```
mga.ext:  mga.def mga.h


clean:
        rm -f *.o mga
```

11.16 The file parameters

```
Maximization (1) = 1
Level wise mGA (1) = 1
Problem size (15) = 30
Maximum era (3) = 3
Probability of cut (0.02) = 0.016667
Probability of splice (1.0) = 1.0
Probability of allelic mutation (0.0) = 0.0
Probability of genic mutation (0.0) = 0.0
Thresholding (0) = 0
Tiebreaking (0) = 0
Reduced initial population (1) = 0
Extra population members (0) = 0
Stopping criteria factor (1.0) = 1.00
Partition file (0) = 0
Plotting file (0) = 1
Population record file (0) = 0
Trace (1) = 1
Copies (10 1 1) = 5 1 1
```

11.17 The file era

```
Total generations (20) = 10
Juxtapositional popsize (100) = 200
Total generations (20) = 10
Juxtapositional popsize (100) = 200
Total generations (20) = 20
Juxtapositional popsize (100) = 250
```

11.18 The file subfunc

```
Subfunction Tables and functions (defined in objfunc.c)
8
------------------------------
0
Table
3
0 0 0 28
0 0 1 26
0 1 0 22
0 1 1 0
1 0 0 14
1 0 1 0
1 1 0 0
1 1 1 30
------------------------------
1
Table
3
0 0 0 0
0 0 1 2
0 1 0 4
0 1 1 12
1 0 0 16
1 0 1 20
1 1 0 28
1 1 1 30
------------------------------
2
Function
3
------------------------------
3
Function
4
------------------------------
4
Function
5
------------------------------
5
Function
6
------------------------------
6
Function
8
------------------------------
```

```
7
Function
10
-------------------------------
Subfunction data
10
2 1 0-2
2 1 3-5
2 1 6-8
2 1 9-11
2 1 12-14
0 1 15-17
0 1 18-20
0 1 21-23
0 1 24-26
0 1 27-29
```

11.19 The file `template`

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

11.20 The file `partitions`

```
partition data
3
3 0 1 2
3 0 1 3
5 0 3 6 9 12
```

11.21 The file `poprecin`

```
Population Statistics
2
0 9
```

11.22 The file `extra`

```
Set of genes
3 3 4 5 10
2 1 2 5
```