mGA1.0: A Common LISP Implementation of a Messy Genetic Algorithm

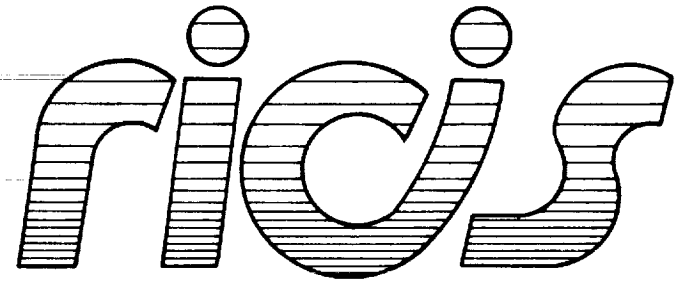
David E. Goldberg
Travis Kerzic

University of Alabama

May 1990

Cooperative Agreement NCC 9-16
Research Activity AI.12

NASA Johnson Space Center
Mission Support Directorate
Mission Planning and Analysis Division



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

mGA1.0: A Common LISP Implementation of a Messy Genetic Algorithm

Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by David E. Goldberg and Travis Kerzic, both of the Department of Engineering Mechanics at the University of Alabama. Dr. Terry Feagin, Professor of Computer Science at the University of Houston-Clear Lake, served as RICIS technical representative for this activity.

Funding has been provided by the Mission Planning and Analysis Division within the Mission Support Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Robert Savely, Head, Artificial Intelligence Section, Technology Development and Applications Branch, Mission Support Directorate, NASA/JSC.

mGA1.0: A Common LISP Implementation of a Messy Genetic Algorithm

David E. Goldberg & Travis Kerzic
Department of Engineering Mechanics
University of Alabama
Tuscaloosa, AL 35487

TCGA Report No. 90004
May 1990

The Clearinghouse for Genetic Algorithms
Department of Engineering Mechanics
University of Alabama
Tuscaloosa, AL 35487-0278

mGA1.0: A Common LISP Implementation of a Messy Genetic Algorithm

David E. Goldberg & Travis Kerzic
The University of Alabama
Tuscaloosa, AL 35487

1 Introduction

Genetic algorithms (GAs) are search and optimization procedures based on the mechanics of natural selection and natural genetics (Goldberg, 1989a; Holland, 1975). As we turn the corner on the 1990s, GAs are finding increased application in difficult search, optimization, and machine learning problems in science and engineering. As more celebrants come to the party, increasing demands are being placed on algorithm performance, and the remaining challenges of genetic algorithm theory and practice are becoming increasingly unavoidable. Perhaps the most difficult of these challenges is the so-called *linkage problem*. Simply stated, the simple genetic algorithms most prominent in current practice assume that *building blocks*—highly fit combinations of features or alleles—are physically compact to prevent their separation by the action of crossover. When this assumption is met or when the problem-coding under consideration is not *deceptive* (Goldberg, 1989b, 1989c), simple GAs work quite well; if the needed linkage is loose and the function-coding is sufficiently difficult, then simple GAs can be misled toward local optima far away from the best point. Standard suggestions to get around this difficulty, such as reordering operators, have not worked well in the past, and there are good theoretical reasons why they may never work well enough to be of timely use (Goldberg & Bridges, 1990).

It is against this backdrop that *messy genetic algorithms* (mGAs) were developed (Goldberg, Deb, & Korb, 1990; Goldberg, Korb, & Deb, 1989). Messy GAs overcome the linkage problem of simple genetic algorithms by combining variable-length strings, gene expression, messy operators, and a nonhomogeneous phasing of evolutionary processing. Results on a number of difficult deceptive test functions have been encouraging with the mGA always finding global optima in a polynomial number of function evaluations. Theoretical and empirical studies are continuing, and a first version of a messy GA is ready for testing by others. The purpose of this report is to document a Common LISP implementation called mGA1.0 and relate it to the basic principles and operators developed by Goldberg et al. (1989, 1990). Although the code has been prepared with care, it is not a general-purpose code, only a research version. This first release contains the basic mGA as well as the thresholding mechanism discussed in Goldberg et al. (1990). It does not contain the tie-breaking mechanism with null bits; this feature will be released in the first update.

In the remainder of this report, important data structures and global variables are described. Thereafter brief function descriptions are given, and sample input data are presented together with sample program output. A source listing with comments is also included.

1.1 Programming and typing conventions

A number of conventions were adopted in programming mGA1.0, and a number of typing conventions have been adopted in writing this report.

All mGA1.0 multiple-word user-defined functions and variables use an underscore between words instead of a hyphen. This makes them easier to distinguish from Common LISP standard functions. All function names are typed in all capital letters (e.g., `OBJFUNC`). All variable names are typed in all lower-case letters (e.g., `bit_specifier`).

All references to function or data names within the text are typed using the `typeface` font (e.g., ...when the `bit_specifier` variable changes value...).

2 Data Structures and Global Variables

The processing in any GA may be viewed as a battle between the members of a population of artificial individuals or organisms set in the context of some environment or objective function. Thus, at least two types of data structure are necessary for mGA processing: those related to individuals and populations and those related to the objective function. In the remainder of this section, population and objective-function data structures and their components are described. In addition, a number of other global variables are detailed.

2.1 Populations and individuals

The program mGA1.0 uses two nonoverlapping populations `newpop` and `oldpop` to coordinate genetic processing. Each generation, the new population, `newpop`, is created by selecting and perhaps recombining and mutating individuals within the old population, `oldpop`. Each population is a list containing `popsize` structures of the type `population_member`. Each individual (each an instance of the structure type `population_member`) carries the essential information about an individual required for genetic and objective-function processing. A schematic of the population and the information carried by each individual is shown in figure 1. The three components of the structure type `population_member` are as follows:

`chrom` The variable `chrom` is a list of genes (ordered pairs), where each gene specifies a gene's name (its number) and its value (a `1` or a `0`). For example, the `chrom` `((2 0) (3 0) (1 1))` specifies the three-bit string 001 in the usual fixed-position simple GA. The `chrom` lists are processed by cut, splice, and mutation operators. Gene redundancies and missing genes are handled in the decoding process. For example, the underspecified string in a three-bit problem `((2 1) (3 1))` with an all-zero competitive template would decode to the string 110, because the zero of the competitive template would fill in the missing position of the messy string. The overspecified string (in a three-bit problem) `((2 0) (3 1) (1 1) (2 1))` would decode to the string 101, because the first instance of the second gene governs decoding on a left-to-right scan. After the decoding process, the objective function calculates a function value through one or more table look-up subfunctions.

`fitness` The floating-point variable `fitness` stores the objective function value of `chrom`. mGA1.0 assumes that fitness is a function to be maximized.

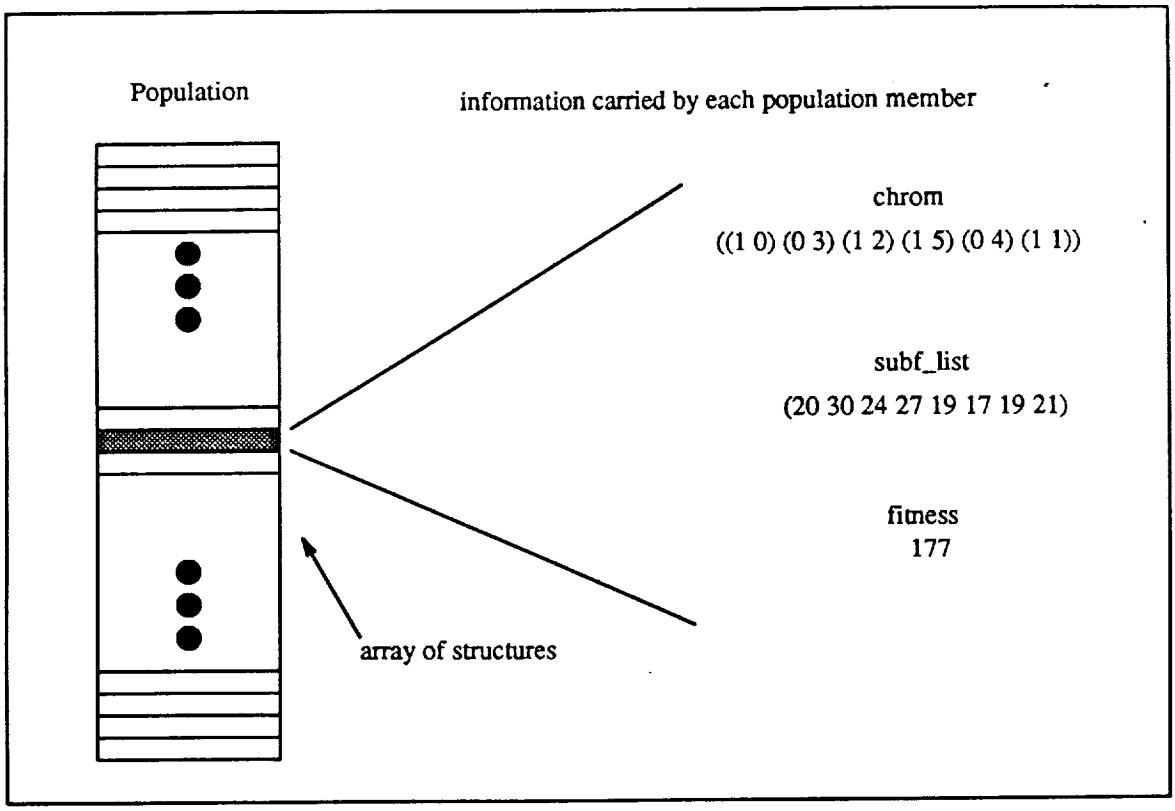


Figure 1: A schematic of the population structure shows the array of individuals together with a sample of the contents of a single individual.

`subf_list` The list of floating-point values `subf_list` stores the individual subfunction values for the objective function. For example, in a three-subfunction problem, if subfunction one has value 30, subfunction two has value 21, and subfunction three has value 22.3, the `subf_list` would have value `(30 21 22.3)`. This storage is used for statistical recordkeeping. It is not used to aid in the genetic processing of mGA1.0 (mGA1.0 is a pure blind GA).

2.2 Data structures associated with the objective function

The objective function in mGA1.0 takes the scaled sum of one or more table look-up subfunctions. This choice of objective function was dictated by the need to test the mGA on test functions that were composed of sums of deceptive functions of different size (order) and scale. In practical applications, the objective function and its data structures will require modification.

To coordinate the summation of the separate subfunctions, a list `bit_spec` of structures of type `bit_cluster` is used to store the subset of bits to be used, the scale factor, and the look-up table number in the following three components:

`bit_specifier` The list of integers `bit_specifier` names the genes (by number) required by the current subfunction. For example, the `bit_specifier (1 2 3)` indicates that genes 1, 2, and 3 are used in that order by the current subfunction.

`scale_factor` The floating-point variable `scale_factor` is a scale factor that multiplies the value of the subfunction found in the look-up table. It may be used to strengthen or weaken the relative contribution of the current subfunction to the total objective function value.

`table_specifier` The integer value `table_specifier` specifies the look-up table number to use when decoding current a subfunction.

The other structures required for objective function processing are the look-up tables themselves. The variable `lookup_table` contains a list of size `num_tables` where each element specifies a full n-bit look-up table. The storage format is as follows:

`lookup_table` The variable `lookup_table` is a list of lists, where each component list is a separate table. Each table is itself a list of lists where the separate entries contain the set of argument bits and their value. For example, the table entry for the bits 00 with value 5 would be stored as ((0 0) 5). An entire table with four entries would be written as

```
(((0 0) 5)((0 1) 7.3)((1 0) 9)((1 1) 12))
```

where the bit combinations 00 ,01 ,10, and 11 have values 5, 7.3, 9 and 12, respectively. An entire example with two tables is given below:

```
lookup_table = ((((0 0) 5)((0 1) 7.3)((1 0) 9)((1 1)12))
               (((0 0) 1)((0 1) 2)((1 0) 0)((1 1) 5))).
```

Here the first table is the one discussed above, and the second is of the same size with values 1, 2, 0, and 5. The table look up is done without regard to the ordering of entries. The look up is performed in a linear fashion, the first matching entry determining the returned value. If no match is found the value subfunction is set to 0.

2.3 Global variables and data structures

This subsection presents an alphabetized list of mGA1.0 global variables and data structures together with brief descriptions of their purpose. A number of global variables are initialized during the setup process; detailed input data file description is presented later in this documentation.

`avg_fitness` The floating-point variable `avg_fitness` contains the average chromosome fitness taken across the entire population.

`bits_per_chrom` The integer variable `bits_per_chrom` specifies the total number of binary variables in the current objective function.

`bit_spec` This variable is a list of structures of type `bit_cluster`. Each structure specifies the realationship between sets of bits and look-up tables to permit objective function evaluation. See section 2.2 for a more detailed description.

`bldg_blk_size` The integer variable `bldg_blk_size` specifies the minimum building block size to be generated by `MAKE_NEW_POP`. It represents the variable k in the population sizing equation $2^k \binom{l}{k}$.

`cookgen` The integer variable `cookgen` specifies the number of generations in the primordial phase (the number of generations to cook the population).

`cut_prob` The floating-point variable `cut_prob` specifies the per-bit cut probability. This value is multiplied by a string's raw length to determine the chromosomal cut probability.

`cutpopgen` During the primordial phase, the population is cut in half every other generation through generation `cutpopgen`.

`data` The stream data specifies an output file for raw setup and generational data. The file has a header written at the beginning to specify the order of written information, and is intended for easy incorporation into a package with graphing abilities (e.g., MATLAB).

`first_time` The flag `first_time` is set to `t` prior to the first pass through the file setup routine. Thereafter the flag is `nil`.

`garbage_collect` The integer variable `garbage_collect` specifies the number of generation between garbage collections. This number depends on the amount of memory a system has and on the population size.

`indata` The variable `indata` is used as an input buffer for the setup file.

init_select_gen The integer variable **init_select_gen** specifies the number of generations INIT_SELECTION will be used.

instream The stream name **instream** defines the input stream and is used to read from the setup file.

lookup_table This variable is a list of look-up tables for objective-function processing. See section 2.2 for a more detailed description.

loopvar The integer variable **loopvar** is used to count the number of generations.

max_numgood The integer variable **max_numgood** is used to keep track of the maximum number of optimum building blocks in any single population member across the the current population. The optimum value is set by scanning individuals and the look-up tables in SETUP_SUBFUNC_MAX.

maxfitness The floating-point variable **maxfitness** contains the value of the maximum fitness across the current population.

maxgen The integer variable **maxgen** specifies the total number of generations that will be executed.

member_copies The integer variable **member_copies** specifies how many copies of each population member will be made in the MAKE_NEW_POP function.

minfitness The floating-point variable **minfitness** contains the value of the minimum fitness value across the current population.

mut_prob The floating-point variable **mut_prob** specifies the probability that a single bit will be mutated from a 1 to a 0 or vice versa by the MUTATION operator

newpop The variable **newpop** is a list of structures of type **population_member**. It contains the current population which is created from the old population via mGA operators every generation. See previous description in section 2.1.

subf_pos_max The floating point list **subf_pos_max** contains a list of the maximum values that each of the subfunctions can obtain. It is used by the function UPDATE_STATS to count the number of optimal building blocks.

num_subfunctions The integer variable **num_subfunctions** specifies the number of subfunctions in the current objective function.

num_tables The integer variable **num_tables** specifies the total number of look-up tables.

numgood The integer variable **numgood** counts the the total number of optimal building blocks in the population.

oldpop The list **oldpop** of structure type **population_member** contains the old population. See previous description in section 2.1.

pick The integer variable **pick** is used as a position marker and indexes into a population selection array **shuffle** used by the selection routines.

popsize The integer variable **popsize** specifies the current population size.

pspe The flag **pspe** is set to **t** to activate the *partial-string-partial-evaluation* mode within mGA1.0. When the flag is true, only fully specified subfunctions will be given a value other than zero. Otherwise, the **std_fill** array (the competitive template) is used to fill the unspecified bit positions so that underspecified chromosomes can be decoded (see UPDATE_STATS).

screen&file The stream name **screen&file** specifies statistical output that is directed to both the screen and the statistics file.

seed The floating-point variable **seed** is used to seed the random number generator when the program is first initialized.

setup_file The character string **setup_file** specifies the pathname of the setup file.

shuffle The integer array **shuffle** contains a randomly generated permutation of the numbers ranging from one to **popsize**.

shufnum The integer variable **shufnum** determines the subpopulation size searched to try to satisfy the requirement for having the threshold number of bits in common in the THRESH_SELECTION routine.

splice_prob The floating-point variable **splice_prob** specifies the probability that two chromosome pieces will be spliced together by the SPLICE operator.

stack The list **stack** is where CUT_AND_STACK places its list of chromosome pieces (see figure 2).

stat The file stream name **stat** is used to specify output to the statistics file.

std_fill The array of integers **std_fill** is used to fill in unspecified positions during chromosome decoding.

sumfitness The floating-point variable **sumfitness** holds the sum of the fitness values of all members of the current population and is used to calculate the population average fitness.

thres The flag **thres** is set to **t** to dictate the use of thresholding through the selection function THRESH_SELECT.

vertnum_spot The integer array **vertnum_spot** counts the number of optimal building blocks in each subfunction position across the population.

3 Brief Function Descriptions

This section presents an alphabetized list of brief function descriptions for primary routines called in the execution of mGA1.0. The functions are placed in seven separate files according to function: **aux.lisp**, **decode.lisp**, **mga.lisp**, **objfunc.lisp**, **ops.lisp**, **setup.lisp**,

and stats.lisp. The file aux.lisp contains utility and auxilliary code. The file decode.lisp contains code responsible for decoding a raw chromosome, thereby generating a processed chromosome. The file mga.lisp contains global variable declarations, the mGA function, and important phase coordination functions. The file objfunc.lisp contains functions related to subfunction look up, scaling, and summation. The file ops.lisp contains the genetic operators. The file setup.lisp contains initialization code, and the file stats.lisp contains statistical and reporting code. Each of the functions descriptions is followed by its corresponding file name.

(CHOOSE n1 n2) The function CHOOSE calculates the number of combinations of n1 objects taken n2 at a time. (aux.lisp)

(COMBINATION max_number num_places comb_list) The COMBINATION operator generates the next combination of max_number things taken num_places at a time in lexicographical order given the current combination in the list comb_list. (aux.lisp)

(COMPLEMENT_BIT bit) The function COMPLEMENT_BIT changes a bit from a one to a zero or vice versa. (aux.lisp)

(CROSSOVER mate1 mate2 cut_prob splice_prob) The function CROSSOVER takes two chromosomes mate1 and mate2 and specified cut and splice probabilities cut_prob and splice_probl and returns a list of between one and four offspring chromosomes through coordinated invocation of the cut and splice operators. CROSSOVER executes calls to the functions CUT_AND_STACK and SPLICE_TESTER. (ops.lisp)

(CUT chromosome) A chromosome is passed to the function CUT, whereupon a random cut point within the chromosome is selected, the chromosome is cut, and the two pieces are returned as a list. (ops.lisp)

(CUT_AND_STACK mate1 mate2 cut_prob) Two chromosomes are passed to the function CUT_AND_STACK together with their cut probability cut_prob. The pieces are CUT and the pieces are stacked in an order appropriate for subsequent splicing, and the resulting stack of between two and four pieces is returned. The operation of CUT_AND_STACK is shown in in figure 2. (ops.lisp)

(DET_SELECTION) This function performs binary tournament selection without replacement. For init_select_gen generations, the function INIT_SELECTION is called. Then, if the value of the flag thres is t, the thresholding selection function THRESH_SELECTION is called. Otherwise, NORM_SELECTION performs simple binary tournament selection. (ops.lisp)

(EXTRACT chrom) The raw chromosome chrom is passed to the function EXTRACT. The chromosome is scanned from left to right, constructing a processed string using the first gene value encountered during the scan. In a five-bit example, if the chromosome ((1 0) (2 1) (5 0)) were passed to EXTRACT, the returned result would be (0 1 nil nil 0). (decode.lisp)

(FACTORIAL n) The function (FACTORIAL) implements a factorial operator as n!. (aux.lisp)

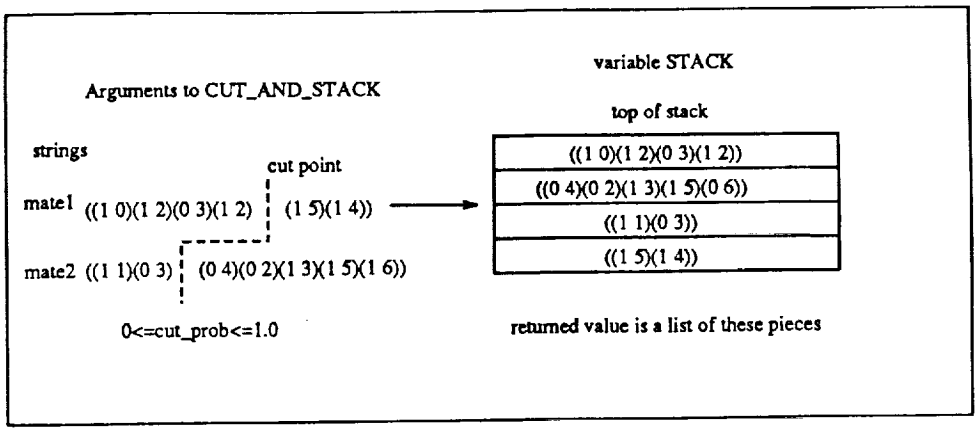


Figure 2: CUT_AND_STACK detail

(FILL_NIL_POSITIONS list1) The function FILL_NIL_POSITIONS takes a processed string list1 and fills in any unspecified positions with the corresponding values in the standard fill array std_fill. For example, the processed string (0 1 nil nil 0) passed to FILL_NIL_POSITIONS with std_fill=#(0 0 0 0 0) would return the filled-in string (0 1 0 0 0). (decode.lisp)

(FLIP probability) The function FLIP randomly returns t with the specified probability; otherwise, it returns nil. (aux.lisp)

(GET_CHROMOSOME pop loc) The index number and population are passed to GET_CHROMOSOME and the chromosome of the locth population member from the population pop is returned. (aux.lisp)

(GET_FITNESS pop loc) The index number and population are passed to GET_FITNESS and the population member's fitness value is returned. (aux.lisp)

(GET_SUBFUNCTION_LIST pop loc) The index number and population are passed to GET_SUBFUNCTION_LIST, and the member's subfunction list is returned. (aux.lisp)

(INIT_SELECTION) The function INIT_SELECTION selects and compares the fitness values of population members in their original order. This is useful in the early stages of a run to maintain tournaments among competing building blocks without additional thresholding. (ops.lisp)

(INITIAL_OUTPUT) The setup parameters read from the setup file are displayed through a call to INITIAL_OUTPUT. (stats.lisp)

(JUXTAPOSITIONAL) The processing of the juxtapositional phase is coordinated by a call to JUXTAPOSITIONAL. (mga.lisp)

(MAKE_NEW_POP num_positions max_length &optional copies) A new population consisting of all possible building blocks of length num_positions with values ranging to max_length are created by the function MAKE_NEW_POP. All

$2^k \binom{l}{k}$ building blocks are generated, running through the population so that competing building blocks are adjacent. (setup.lisp)

(mGA) The mGA function coordinates the overall processing of the messy genetic algorithm. It is responsible for controlling the primordial and juxtapositional genetic phases as well as cutting the populations size. On start up, it calls the setup file load routine and executes calls to initialize the population. (mga.lisp)

(MUTATE gene probability) The gene value gene is changed from a zero to a one or vice versa with the specified probability. (ops.lisp)

(MUTATION chrom) A raw chromosome is passed to MUTATION. Each bit is mutated or not via successive calls to MUTATE, returning the mutated chrom. (ops.lisp)

(N_ARY_COUNT n_ary_list maximum position) The function N_ARY_COUNT increments the position of n_ary_list by one. The position specifier starts with zero indicating the farthest right input list position and the length of the list minus one for the farthest left input list position. The function implements carry-over when a lower position is filled. In the mGA program N_ARY_COUNT is used to implement a binary counter. An example of a three bit binary counter implementation is as follows: Initialize the counting list to (0 0 0). Call the the function with the following arguments (N_ARY_COUNT '(0 0 0) 2 0). This specifies that position zero (the least significant bit) of the list will be incremented in a binary 2 mode. The result of the call would be (0 0 1). Now if the process is repeated using the result of the last call as the input list, the resulting returned sequence of lists would be: (0 1 0), (0 1 1), (1 0 0), (1 0 1),..., (1 1 1). When all the values are at their maximum the list should be reset to (0 0 0) before another call is initiated. (aux.lisp)

(NORM_SELECTION) This function compares the next two individuals indexed in the permutation shuffle, returns the index number of the better individual, and increments the pick index by two. (ops.lisp)

(OBJFUNC &optional calc_val_flag) Objective function values are assigned to the new population newpop by the function OBJFUNC through successive calls to EXTRACT, FILL_NIL_POSITIONS, SUBFUNCTION_DECODE, and SET_SUBFUNCT_VALUE. When calc_val_flag is t, table look up is performed. When calc_val_flag is nil, the evaluation is bypassed. This feature is used during the primordial phase to prevent unnecessary reevaluation of function values. (objfunc.lisp)

(PRIMORDIAL) The primordial phase of the mGA is coordinated by the function PRIMORDIAL. Only selection is performed during this phase; cut, splice, and other operators are postponed to the juxtapositional phase. With deterministic fitness functions, no reevaluation of a string's fitness is necessary during this phase. The primary purpose of this phase is to dope the population with the best building blocks, enabling their useful recombination during the juxtapositional phase. (mga.lisp)

(RESET_POP respop) This function creates a nil initialized population of size popsize in the population respop. (aux.lisp)

(RESET_STAT_INFO) The function (RESET_STAT_INFO) resets various statistical counters and variables. It is executed prior to the evaluation of objective function values across the population in OBJFUNC. (stats.lisp)

(RND lo hi) A uniformly distributed random integer between low and high limits of lo and hi is calculated by the function RND. (aux.lisp)

(SET_ASC_LIST list start) The function SET_ASC_LIST sets the first element of list to the integer start. The following list positions are set to start+1, start+2,.., start + list-length -1. (aux.lisp)

(SET_LIST list value) This function initializes the elements of list to value. (aux.lisp)

(SET_SUBFUNC_VALUE sub_info_list) This function takes the list of subfunction table numbers, scale factors, and bit lists generated by SUBFUNCTION_DECODE and returns the subfunction fitness values as a list. The fitness values are set by finding a match in the corresponding look-up table. (objfunc.lisp)

(SETUP_GA) This function is responsible for picking out the various fields in the setup file and issuing calls to the other setup functions. (setup.lisp)

(SETUP_METER) This function sets up a meter that is 20 characters wide. It is used in the objective function to show how much of the population has been processed. (aux.lisp)

(SETUP_OBJ) This function uses the setup file to set up the look-up tables by issuing calls to SETUP_TABLES. It also sets up the bit_spec subfunction specifier. (setup.lisp)

(SETUP_POP) This function accesses the setup file to set up all the population and global parameters. (setup.lisp)

(SETUP_SUBFUNC_MAX) This function sets up a list of maximum possible values for the subfunctions by accessing the lookup tables and applying the appropriate scale factor. It is used to count optimal building blocks in the function UPDATE_STATS. (stats.lisp)

(SETUP_TABLE) This function is responsible for setting up a single look-up table specified in the setup file. (setup.lisp)

(SETUP_TEMPLATE) This function uses the setup file to set the default fill-in values of the std_fill array. (setup.lisp)

(SHUFFLE_POP shuffle) This function creates a shuffled array shuffle of the population member numbers. (aux.lisp)

(SPLICE chrom1 chrom2) Two chromosomes chrom1 and chrom2 are spliced together by SPLICE, returning a single chromosome. (ops.lisp)

(SPLICE_TESTER splice_prob) The stack generated by the function CUT_AND_STACK is used by SPLICE_TESTER to coordinate splicing. Pieces are successively popped off the stack, and tests are performed for possible splicing of each piece to the next. A successful splice results in the placement of the result in a list of offspring that are subsequently placed in the new population. (ops.lisp)

(STATISTICS) This function writes various statistics values to the screen and to a file. (stats.lisp)

(SUBFUNCTION_DECODE bit_list) This function takes the processed string bit_list and generates a list of look-up table numbers, scale factors, and bit values for each subfunction. This list of lists is passed on to the function SET_SUBFUNC_VALUE, which performs the required table accesses. (objfunc.lisp)

(THRESH_SELECTION) Tournament selection with thresholding (genic selective crowding) is performed by the function THRESH_SELECTION. This function selects the first participant as the next individual mentioned in the shuffle list as indicated by the pick index. A threshold value is calculated to determine how many genes two individuals must have in common to be compared, and the next shufnum individuals in the shuffle list are checked for the requisite gene commonality. The first individual with sufficient commonality is compared to the individual chosen earlier, and the index number of the fitter is returned. If no individual with the required commonality is found, the index number of the first participant is returned. (ops.lisp)

(UPDATE_STATS pop_member) The function UPDATE_STATS updates statistics for population subfunction fitness maxima and averages. (stats.lisp)

4 Example Input and Output

This section provides an example run with setup file and output from the program. The example is included to exercise many features of mGA1.0, thereby permitting the user to easily create his own files.

4.1 The problem

The example problem is to find the optimum solution for a function composed of eight subfunctions. The subfunctions are all deceptve and are decoded from two different look-up tables. The first subfunction (subfunction 0) was designed using guidelines outlined in (Liepens & Vose, 1989). The second subfunction (subfunction 1) was designed by Goldberg (1989b). Three of the subfunctions use subfunction 0 and the other five use subfunction 1 (see figure 3).

4.2 Setup file

The setup file is used to set the population variables, look-up tables, and decoding parameters. The user is prompted for this file name when the mGA program is started. The file is composed of two primary fields, the POPULATION and OBJ fields. The setup file requires that the OBJ field

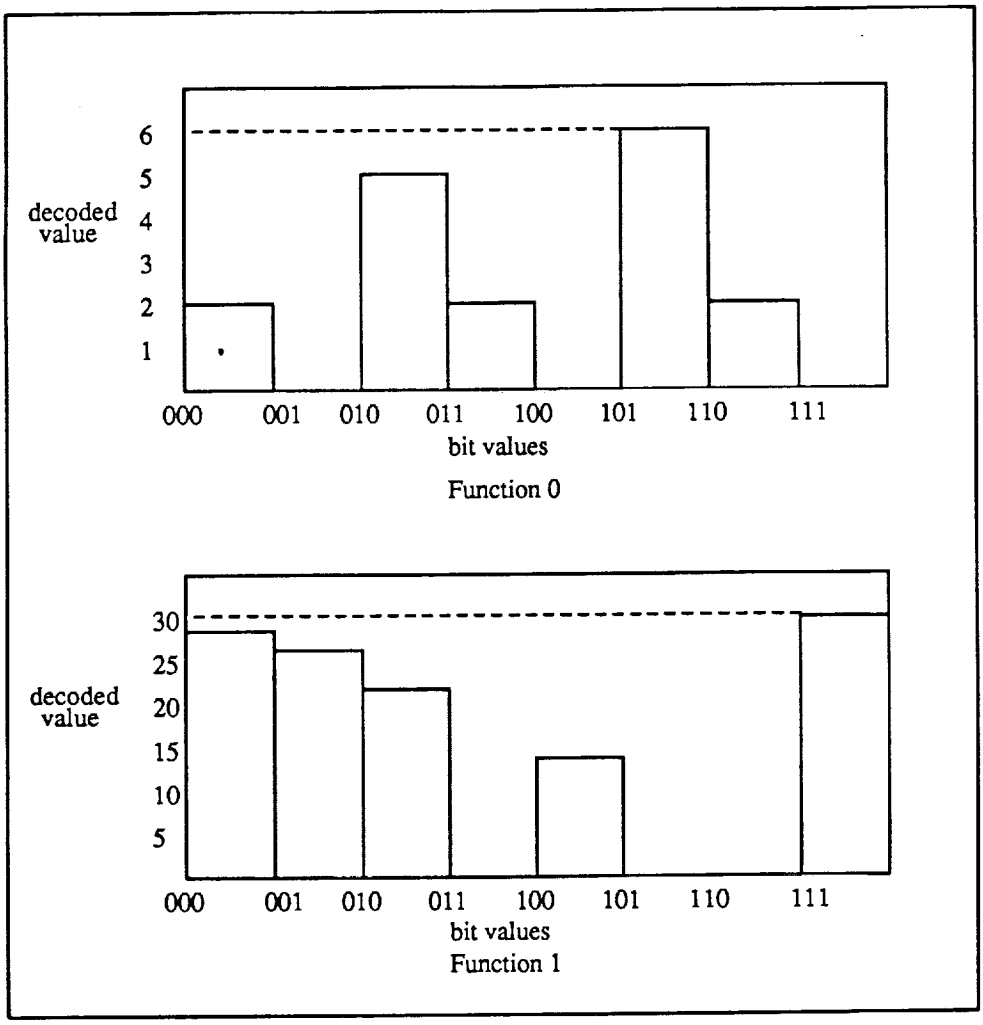


Figure 3: The two deceptive subfunctions used in the example problem.

follow the POPULATION field, because some of the subfunction variables are set in the POPULATION setup section. The setup file for the this example is as follows:

```
POPULATION;;example1 - two deceptive lookup table problem

maxgen 30
cookgen 14
cutpopgen 12
init_select_gen 3
num_subfunctions 8
chrom_length 24
bldg_blk_size 3
seed 3
cut_prob  .0208333
splice_prob 1
mut_prob 0
thres 1
shufnum 24
garbage_collect 2
***

OBJ

subfunction_bits  0  1  2  ltable 0 scale 1
subfunction_bits  3  4  5  ltable 1 scale 1
subfunction_bits  6  7  8  ltable 1 scale 1
subfunction_bits  9 10  11 ltable 0 scale 1
subfunction_bits  12 13 14 ltable 0 scale 1
subfunction_bits  15 16 17 ltable 1 scale 1
subfunction_bits  18 19 20 ltable 1 scale 1
subfunction_bits  21 22 23 ltable 1 scale 1

table 0
bvalue 0 0 0 dvalue 2
bvalue 0 0 1 dvalue 0
bvalue 0 1 0 dvalue 5
bvalue 0 1 1 dvalue 2
bvalue 1 0 0 dvalue 0
bvalue 1 0 1 dvalue 6
bvalue 1 1 0 dvalue 2
bvalue 1 1 1 dvalue 0
end_table

table 1
bvalue 0 0 0 dvalue 28
bvalue 0 0 1 dvalue 26
bvalue 0 1 0 dvalue 22
bvalue 0 1 1 dvalue 0
```

```
bvalue 1 0 0 dvalue 14
bvalue 1 0 1 dvalue 0
bvalue 1 1 0 dvalue 0
bvalue 1 1 1 dvalue 30
end_table

          0 1 2 3 4 5 6 7 8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
template 0 1 0 0 0 0 0 0 0  0  1  0  0  1  0  0  0  0  0  0  0  0  0  0
***
end
```

There are a few conventions that should be noted in this file. A comment (anything that follows a semicolon) is ignored. Anything that does not follow a valid label is ignored, so comments, such as the bit location marker placed above `template` can be used rather freely. The field name is not case sensitive, but the spelling of each field is important. The end of the file is specified by the marker `end`. The `POPULATION` field is used to set up the population and phase parameters. The parameters settings in this example are as follows:

`maxgen 30` The mGA program will execute 30 generations total (see figure 4).

`cookgen 14` The primordial phase will be executed for 14 generations, after this generation (generation 15 and on), juxtapositional processing will be used (see figure 4).

`cutpopgen 12` Starting from generation 1 and up to generation 12, the population will be cut in half every other generation (see figure 4).

`init_select_gen 3` Starting from generation zero and up to and including generation three, the function `INIT_SELECTION` will be used for selection. (default = 1)

`num_subfunctions 8` There are eight subfunctions in the problem.

`chrom_length 24` Since there are eight subfunction each of length three, the decoded chromosome length is 24.

`bldg_blk_size 3` When the initial population is created, the function `MAKE_NEW_POP` will create building blocks of size three.

`seed 3` The random number generator is seeded with the value 3.

`cut_prob .0208333` The cut probability is set to one divided by twice the problem length. (default = 0)

`splice_prob 1` The splice probabilty is set to 1 so that all splices will be successful. (default= 1)

`mut_prob 0` Mutation is turned off in this example. (default = 0)

`thres t` Since the subfunctions lookup tables use two different scales (one is five times the other), thresholding selection is used to preserve the weaker building blocks.

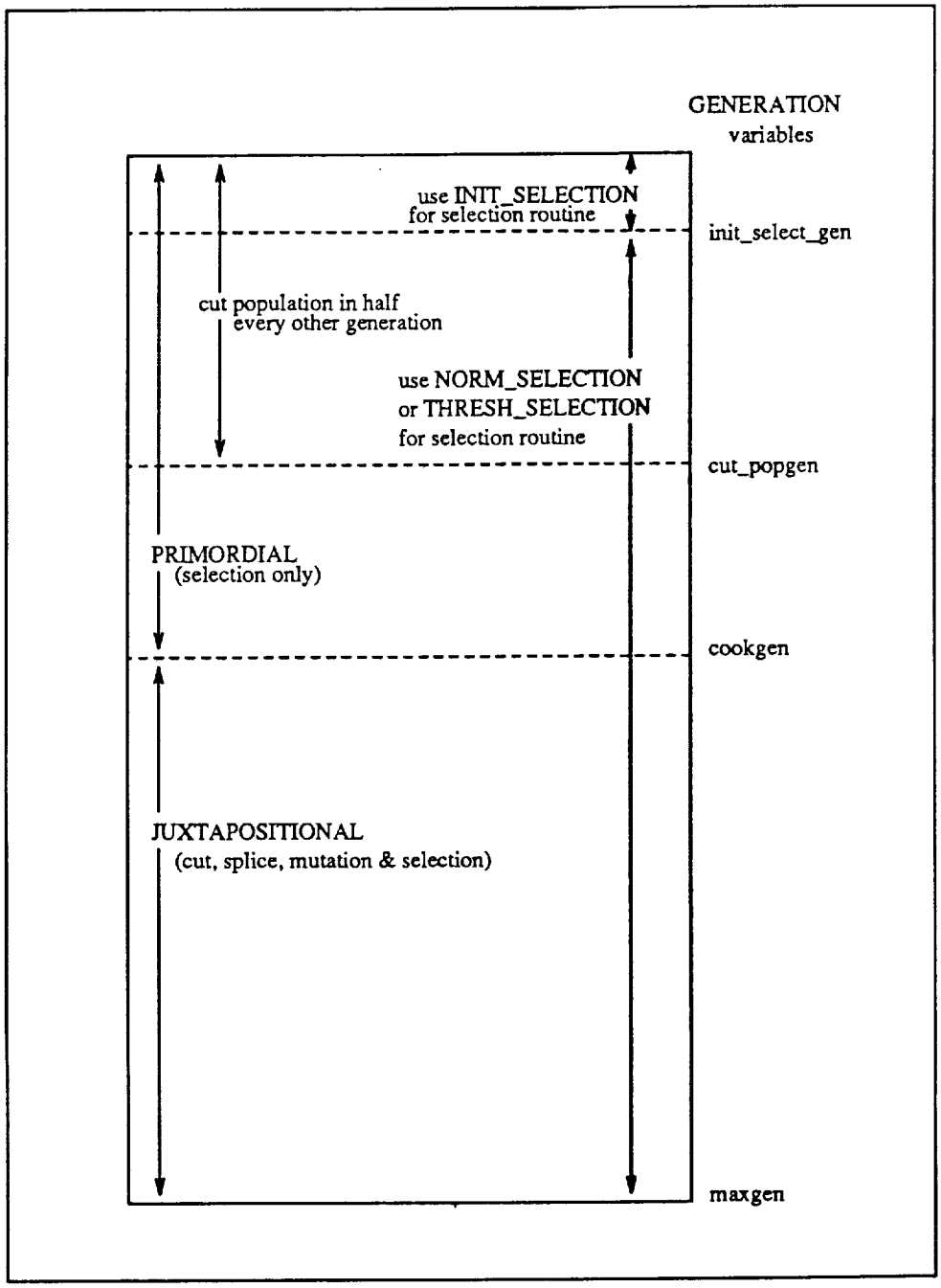


Figure 4: Coordination of the primordial and juxtapostional phases.

`shufnum 24` The shuffle number used by thresholding mechanism is set to the chromosome length (Golberg, Deb, & Korb, 1990).

`garbage_collect 2` Garbage collection will be done every other generation. (default = 1)

Some setup parameter of interest that are not used in this example are:

`pspe` If the value following this field is set to `t`, the *partial-string-partial-evaluation* mode will be activated. When in this mode, the objective function will only give values other than zero to fully specified strings. Otherwise, the `std_fill` array is used to fill the unspecified bit positions os that unsderspecified chromsomes can be decoded. (default = `nil`)

`member_copies` The number following this label will set the number of copies of each population member to be created by the function `MAKE_NEW_POP`. (default = 1)

The `OBJ` field is composed of three subfields: subfunction setup, look-up table setup, and the template field. The field is ended with the `***` pattern. The first subfield, subfunction setup, is specified by the following labels:

`subfunction_bits` This label should be followed by the position numbers that should be accessed in decoding the subfunction. For example if the bits `(3 4 5)` are specified, the subfunction decode section of the program accesses the bits in third, fourth, and fifth position to be used to find a match in the look-up table.

`ltable` This label is followed by the lookup table that should be used in decoding the subfunction.

`scale` This label is followed by a scale factor that is multiplied by the table value. This parameter can be used to either strengthen or weaken the subfunction.

The second subfield, used to setup the look-up table, is specified by the following labels:

`table` This label specifies the look-up table number that identifies the look-up table. This number can be any value, as long as it matches one specified in the subfunction definition. It is suggested that the table numbering start from zero and proceed sequentially for clarity.

`bvalue` This label specifies the bit pattern that is to be matched during table look up.

`dvalue` This label specifies the value that will be assign to the subfunction when the preceeding bit pattern is matched.

`end_table` This label is used to mark the end of a look-up table.

These tables have been set up to implement table lookup for the two subfunctions shown in figure 3.

The final subfield is the `template` field. This label is followed by the default fill-in values for unspecified bit positions. This problem intentionally sets the default fill-in value to the

complement of the optimal subfunction. Observe subfunction 0 (see figure 3) which uses bits (0 1 2) and has optimal value with bits set to (1 0 1). It has a default fill in for those positions of (0 1 0), which is the complement of the optimum. The other default fill-in postions are set in the same manner. The reason for doing this is so the mGA has to do all the work in finding the optimum values and is not given the help of having optimum values filled in by default.

4.3 Off to the program

The program is started by typing (mGA) <cr>. It will then prompt the user for the filename; this example uses example1.asc. The program will then display the setup parameters read from the file, and create the initial population. Once the initial population is created the program starts execution. The objective function will first be executed and the program will print the following:

```
OBJFUNC/STATISTICS (set fitnesses & update stats)
percent population processed
0%                  100%
**********
```

The meter shows how much of the population has been processed. The objective function is first run so that the initial population fitnesses and statistics can be calculated. After this phase the initial statistics will be output as follows:

```
Generation# 0
Popsize = 16192
Minfitness = 113
Maxfitness = 157
Average fitness = 145.93954
Maximum number of optimal subfunctions = 1
Total number of optimal subfunctions = 8
Average number of optimal subfunctions = 4.9407115e-4
Total and Average Number of optimal subfunctions in position 1 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 2 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 3 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 4 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 5 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 6 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 7 = 1 and 6.1758895e-5
Total and Average Number of optimal subfunctions in position 8 = 1 and 6.1758895e-5

Best solution so far =(0 1 0 1 1 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0)
```

The statistics are written each generation both to the screen and to a file (statoutrd.dat). Initially there is only one optimal building block in each subfunction position and ten optimal building blocks in the entire population. Once the initial statistics are printed out the program executes the primordial phase and objective function as follows:

```
PRIMORDIAL
percent population processed
```

```
0%                    100%
**********************

OBJFUNC/STATISTICS (update stats only)
percent population processed
0%                    100%
**********************

Generation# 1
Popsize = 16192
Minfitness = 127
Maxfitness = 157
Average fitness = 149.33684
Maximum number of optimal subfunctions = 1
Total number of optimal subfunctions = 16
Average number of optimal subfunctions = 9.881423e-4
Total and Average Number of optimal subfunctions in position 1 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 2 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 3 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 4 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 5 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 6 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 7 = 2 and 1.2351779e-4
Total and Average Number of optimal subfunctions in position 8 = 2 and 1.2351779e-4

Best solution so far =(0 1 0 1 1 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0)
Fitness = 157
```

The primordial phase creates copies of individuals with highest fitness. Since the structures with optimal building blocks have highest fitness they will be allocated an increasing number of copies at each generation, and hence the population will be doped with good building blocks by the juxtapositional phase . The program will continue in this manner until generation 15 (the generation after cookgen). Here the program begins the juxtapositional phase by using the cut and splice operators. The maximum number of subfunctions in a single individual immediately jumps to two and the maximum fitness increases. The number of optimal building then increases in each subsequent generation until an optimal solution is found in generation 18 (see Figure 5). From this point on the mGA creates more and more optimal solutions until at generation 30 almost every population member has optimal building blocks in all 8 positions. The final statistical output is as follows:

```
Generation# 30
Popsize = 253
Minfitness = 100
Maxfitness = 168
Average fitness = 166.09882
Maximum number of optimal subfunctions = 8
Total number of optimal subfunctions = 1979
```

```
Average number of optimal subfunctions = 7.8221345
Total and Average Number of optimal subfunctions in position 1 = 244 and 0.9644269
Total and Average Number of optimal subfunctions in position 2 = 248 and 0.9802371
Total and Average Number of optimal subfunctions in position 3 = 245 and 0.96837944
Total and Average Number of optimal subfunctions in position 4 = 249 and 0.98418975
Total and Average Number of optimal subfunctions in position 5 = 246 and 0.972332
Total and Average Number of optimal subfunctions in position 6 = 249 and 0.98418975
Total and Average Number of optimal subfunctions in position 7 = 248 and 0.9802371
Total and Average Number of optimal subfunctions in position 8 = 250 and 0.9881423

Best solution so far =(1 0 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1)
Fitness = 168
```

This case is typical for the mGA. The number of function evaluations can be calculated as follows:

$$n_0 + n_j \cdot t_j,$$

where n_0 is the initial population, n_j is the juxtapositional population size, and t_j is the number of generations in the juxtapositional phase to optimality. In the example run, the program required

16192 + 253*4 = 17,204 function evaluations.

The mGA program writes a data file (`DATA.DAT`) which contains raw numerical data useful for plotting. The plot of this example shows the convergence of the mGA in figure 5. The file has a header for data reference at the top of the file and statistical data written for each generation. As an example, the file for this problem starts as follows:

```
Generation Popsize Maximum_Fitness Average_Fitness
0 16192 157 145.93954
1 16192 157 149.33684
2 16192 157 152.36339
3 8096 157 154.07239
4 8096 157 154.8398
5 4048 157 155.01631
6 4048 157 155.05188
7 2024 157 155.10178
8 2024 157 155.20207
9 1012 157 155.37846
10 1012 157 155.72035
.
.
.
```

5 Conclusions

This report has presented brief documentation for the data structures and functions of a Common LISP implementation of a messy genetic algorithm. This implementation, mGA1.0, is available as a research code for investigation of messy genetic algorithm theory and application.

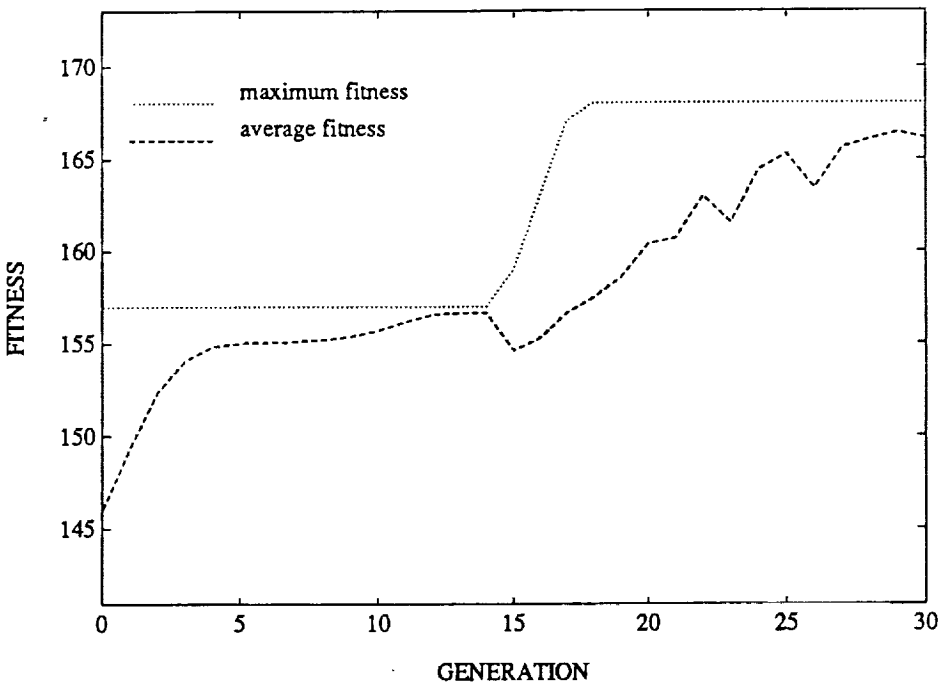


Figure 5: Optimal results are found quickly in this example.

In its present state, the code includes the phased (primordial/justapositional) processing of a messy GA, messy binary strings, messy cut and splice operators, and genic selective crowding. The code does not currently include null bits or tie breaking, although these features will be implemented soon.

Although mGA efforts are just beginning, because of their rapid convergence to global optima in difficult combinatorial problems, these techniques are ready for additional scrutiny and testing. Making this research code available is a first step in this direction.

Acknowledgments

This material is based upon work supported by NASA Cooperative Agreement NCC9-16, RICIS Research Activity No. AI.12, Subcontract No. 045 and by the National Science Foundation under Grant CTS-8451610. The authors also acknowledge equipment support provided by Texas Instruments Incorporated and the Digital Equipment Corporation.

References

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). *An investigation of messy genetic algorithms* (TCGA Report No. 90005). Tuscaloosa: University of Alabama, The Clearinghouse for Genetic Algorithms.

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

Leipens, G.E., & Vose, M.D.(1989). *Representational issues in genetic optimization.* Manuscript submitted for publication.

A Appendix

A.1 File: aux.lisp

The file aux.lisp contains utility and auxiliary code for mGA1.0.

```
;============================FILE-AUX.LISP================================
;=This file contains the auxilliary operators that accomplish some of the=
;=low level and irregular functions                                       =
;======================================================================

;=======================GET_CHROMOSOME=================================
;accesses the structure of a population member and returns the chromosome
(defun GET_CHROMOSOME (pop loc)
  (population_member-chrom (aref pop loc)))

;=====================GET_FITNESS=======================================
;accesses the structure of a population member and returns the fitness
(defun GET_FITNESS (pop loc)
  (population_member-fitness (aref pop loc)))

;==================GET_SUBFUNCTION_LIST================================
;accesses the structure of a population member and returns the subfunction
(defun GET_SUBFUNCTION_LIST (pop loc)   ; value list
  (population_member-subf_list (aref pop loc)))

;=======================RND===============================================
(defun RND (lo hi)   ;generates a random number from lo to hi
  (+ lo (random (+ (- hi lo) 1))))

;============================COMBINATION=================================
;creates a combination list of max_number items taken num_places @ a time
(defun COMBINATION (max_number num_places comb_list &aux exit)
  (setf max_number (1- max_number))   ;adjust for list index 0 to n-1
  (setf num_places (1- num_places))   ;instead of 1 to n
  (if (< (nth num_places comb_list) max_number)
  ;if low val (farthest right) is less than
  ;max_number then inrement by one
      (setf (nth num_places comb_list) (1+ (nth num_places comb_list)))
      (progn   ;otherwise
        (do ((j num_places (1- j)))   ;check the rest of the positions
     ((or (< j 1) exit))
  (if (< (nth (1- j) comb_list)
 (+ max_number (* -1 num_places) j -1))
  ;look for one that can be inremented
      (progn
        (setf (nth (1- j) comb_list) (1+ (nth (1- j) comb_list)))
  ;increment the next highest value that
```

```
  ;can be incremented
(do ((s j (1+ s)))
     ((> s num_places))
  ;and set all following vals one
  ;higher than the value to its left
  (setf (nth s comb_list)
(+ (nth (1- j) comb_list) s (- 1 j)))
  (setf exit 't)))   ;then exit
      )))))

;=======================RESET_POP=======================================
;resets all population parameters to nil by recreating the structure
(defun RESET_POP (respop)
  (do ((i 0 (1+ i)))
      ((>= i popsize))
    (setf (aref respop i) (make-population_member))))

;=======================FLIP============================================
;simulates the flip of a weighted coin
(defun FLIP (probability &aux tbase)
  (setq tbase 1000000)
  (>= (* probability tbase) (random tbase)))

;====================COMPLEMENT_BIT=====================================
(defun COMPLEMENT_BIT (bit &aux bit_ret)  ;changes a 1 to 0 or a 0 to 1
  (setq bit_ret 1)
  (cond
    ((= bit 1)
     (setq bit_ret 0)))
  bit_ret)

;=======================SHUFFLE_POP=====================================
;creates a shuffled array of size popsize
(defun SHUFFLE_POP (shuffle &aux count num other)
  (do ((i 0 (1+ i)))
      ((>= i popsize))
    (setf (aref shuffle i) i))
  (setf count (1- popsize))
  (do ((j 0 (1+ j)))
       ((>= j count))   ;swap entire population around
    (setf num (RND j count))   ;determine random swap position
    (setf other (aref shuffle num))
    (setf (aref shuffle num) (aref shuffle j))   ;swap one to the other
    (setf (aref shuffle j) other))
  ;and the other back to the original place
  shuffle)
```

```
;====================SETUP_METER=========================================
;sets up meter to show how much of the pop has been processed by OBJFUBC
(defun  SETUP_METER ()
  (format t "percent population processed~%")
  (format t "0%                  100% ~%" ))

;=========================N_ARY_COUNT=====================================
;counts a list in n-ary mode where n is maximum value of count
(defun N_ARY_COUNT (n_ary_list maximum position &aux len)
  (if (and (>= (nth (1- position) n_ary_list) maximum)
  ;if the position above is maxed out
   (>= (nth position n_ary_list) maximum))
  ;and the current positon is maxed out
      (N_ARY_COUNT n_ary_list maximum (1- position))
  ;then check next positon up with this subroutine
      (progn
(if (>= (nth position n_ary_list) maximum)
  ;otherwise if just positon is maxed out but there
  ;is room above it for incrementation
    (progn
      (setf len (list-length n_ary_list))
      (do ((i position (1+ i)))
   ((>= i len))
  ;set everything below the position to zero
        (setf (nth i n_ary_list) 0))
  ;incremenmt the next higher positon
      (setf (nth (1- position) n_ary_list)
    (1+ (nth (1- position) n_ary_list))))
    (setf (nth position n_ary_list) (1+ (nth position n_ary_list))))
  ;if neither of these conditions are met,
  ;just increment the current position
))
  n_ary_list)   ;return the incremented n_ary list

;====================================CHOOSE================================
(defun CHOOSE (n1 n2)   ;statistical operator n1 choose n2
  (/ (factorial n1) (* (factorial n2) (factorial (- n1 n2)))))

;==============================FACTORIAL=================================
(defun FACTORIAL (n &aux ret_val) ;standard stat factorial operator
  (setf ret_val 1)
  (do ((i 1 (1+ i)))
      ((> i n))
    (setf ret_val (* i ret_val)))
    ret_val)

;=======================SET_ASC_LIST=====================================
;sets up an ascending list starting at start and ending with start+length
```

```
(defun SET_ASC_LIST (list start)
  (do ((i start (1+ i)))
      ((>= i (list-length list)))
    (setf (nth i list) i)))

;==============================SET_LIST==================================
;sets all the elements of a list to a value
(defun SET_LIST (list value)
  (do ((i 0 (1+ i)))
      ((>= i (list-length list)))
    (setf (nth i list) value)))
```

A.2 File: decode.lisp

The file decode.lisp contains the code necessary to compute a processed string from a raw string.

```
;=========================FILE-DECODE.LISP================================
;=This file contains functions responsible for decoding a raw chromosome =
;=thereby generating a processed chromosome                              =
;=========================================================================

;=======================EXTRACT=========================================
;EXTRACT takes a position/value list mGA coding and puts the bits in their
;proper positions   example-- (EXTRACT ((2 0)(1 1)(3 1)....))
;                     would return     (nil 1 0 1...)
(defun EXTRACT (chrom &aux ret_list pos val pos_val no_duplicate length)
  (setf ret_list (make-list bits_per_chrom))
  ;setup list to copy bits into
  (setf no_duplicate (remove-duplicates chrom :key 'car :from-end 't))
  ;remove overspecified positions
  (setf length (list-length no_duplicate))
  ;get # of positions to be set
  (do ((i 0 (1+ i)))
      ((>= i length))
    (setf pos_val (nth i no_duplicate))
    (setf pos (nth 0 pos_val))   ;get position
    (setf val  (nth 1 pos_val))   ;get value
    (setf (nth pos ret_list) val))
  ;set ret_list pos(ition) to val(ue)
  ret_list)

;===================FILL_NIL_POSITIONS==================================
;fills unspecified positions with std_fill array vals
(defun FILL_NIL_POSITIONS (list1)
  (do ((pos 0 (1+ pos)))
      ((>= pos (list-length list1)))
    (if (equal (nth pos list1) nil)
  ;look for positions w/ nil value
        (setf (nth pos list1) (nth pos std_fill))))
  ;& fill them w/ std_fill array
  list1)
```

A.3 File: mga.lisp

The file mga.lisp contains data declarations, the main program, and major coordination functions.

```
;============================FILE-MGA.LISP================================
;=This file contains the global variable declarations, the mGA function, =
;=and important phase coordination functions                             =
;=====================================================================
(defstruct population_member chrom fitness subf_list)
  ;structure used for each member of a population

(defstruct bit_cluster table_specifier scale_factor bit_specifier)
  ;structure used for bit specifier decoder
(defvar avg_fitness)       ;statistical variable which specifies the
                           ;population average fitness
(defvar bit_spec)          ;specifies for each subfunction bits,scale
                           ;factor and lookup table
(defvar bits_per_chrom)    ;specifies the length of decoded chromosome
(defvar bldg_blk_size)    ;set in setup file is used to generate initial
                           ;population
(defvar cookgen)           ;the # of generations in the primordial phase
(defvar cut_prob)          ;prob that a chrom will be cut into two pieces
                           ;by CUT
(defvar cutpopgen)         ;the # of generations up to that the pop will
                           ;be cut in half
(defvar data)              ;stream to output data for plotting or analysis
(defvar first_time)        ;flag used by SETUP_GA to mark first time
                           ;through setup routine
(defvar garbage_collect)   ;set number of generation between garbage
                           ;collections
(defvar indata)            ;temp variable used to read from file
                           ;(see SETUP_GA)
(defvar init_select_gen)   ;# of generations to use INIT_SELECTION
(defvar instream)          ;a stream used to read from the setup file
(defvar lookup_table)      ;specifies the bit strings and their
                           ;coresponding decoded values
(defvar loopvar)           ;is the generation counter
(defvar max_numgood)       ;statistical variable which keeps track of
                           ;max # of optimum building blocks
(defvar maxfitness)        ;statistical variable which holds the maximum
                           ;decoded chrom fitness
(defvar maxgen)            ;the total # of generations that will be executed
(defvar member_copies)     ;# of building block copies to be made in initial
                           ;population
(defvar minfitness)        ;statistical variable which holds the maximum
                           ;decoded chrom fitness
(defvar mut_prob)          ;probability that s single bit will be mutated
                           ;by MUTATION
```

```
(defvar newpop)           ;array of structure population_member
(defvar num_subfunctions) ;specifies the # of subfunctions
(defvar num_tables)       ;specifies the # of lookup tables
(defvar numgood)          ;stat variable which keeps track of the total # of
                          ;(111) building blocks
(defvar oldpop)           ;array of structure population_member
(defvar pick)             ;postion marker for selection routines
                          ;THRESH_SELECT,NORM_SELECTION & DET_SELECT
(defvar popsize)          ;the size of the population
(defvar pspe)             ;flag for partial string partial evaluation
(defvar screen&file)      ;output stream for both screen and file
(defvar seed)             ;random # generator seed
(defvar setup_file)       ;name of the file that the setup parameters
                          ;are read from
(defvar shuffle)          ;shuffle array used in selection routines
(defvar shufnum)          ;max # of pop members that will be looked at in
                          ;THRESH_SELECT
(defvar splice_prob)      ;prob that two chroms will be spliced together
                          ;by SPLICE
(defvar stack)            ;global stack for subsequent calls to stack
(defvar stat)             ;output statistical file path vairable
(defvar std_fill)         ;array used for filling in unspecified positions
                          ;during chrom decode
(defvar subf_pos_max)     ;list used to count optimal building blocks
(defvar sumfitness)       ;stat variable which holds the sum of all the
                          ;fitnesses in the pop
(defvar thres)            ;is a flag when set true the program will use
                          ;THRESH_SELECT for selection
(defvar vertnum_spot)     ;stat counter that countes the # of optimal
                          ;building blocks for each subfunction position

;====================mGA================================================
;this is the main control structure of the program that is in charge
;of overall program flow
(defun mGA ( &aux max_gene)
  (format t "MGA~%")
  (setq w:more-processing-global-enable nil)
  ;keeps **MORE from waiting for key hit
  (setq stat (open "statoutrd.dat" :direction :output))
                          ;open data file for stats
  (setq screen&file (make-broadcast-stream *standard-output* stat))
                          ;sets up output screen & file streams
  (setq data (open "data.dat" :direction :output));sets up data file
  (setf loopvar 0)
  (SETUP_GA)              ;program which loads setup file
  (INITIAL_OUTPUT)        ;output initial information
  (setf subf_pos_max (SETUP_SUBFUNC_MAX)) ;setf up subfunction maxim
  ;for statistics optimium bldg block counting
```

```
  (setq *random-state* (system:random-create-array 71. 35. seed))
                          ;seed random # generator
  (setf max_gene (1- bits_per_chrom))   ;set array max number
  (MAKE_NEW_POP bldg_blk_size bits_per_chrom member_copies)
                          ;generate every useful building block
                          ;of size building_blk_size
  (OBJFUNC t)             ;calculate initial objective function vals
  (STATISTICS)            ;print out statistics
  (do ((gen_count 1 (1+ gen_count)))   ;generation loop
      ((> gen_count maxgen))
    (setf pick 0)   ;reset shuffle array pointer
    (setf loopvar gen_count)   ;sets global loopvar to local gen_count
    (if (= (rem gen_count garbage_collect) 0)
        (progn
  (format t "collecting garbage~%")   ;collect garbage
  (gc-immediately :silent t)
  (format t "finished collecting garbage~%")))
    (if (<= gen_count cookgen)
                          ;check for primordial phase or juxtapos phase
        (progn
  (setf shuffle (make-array popsize))   ;create shuffle array
  (SHUFFLE_POP shuffle)   ;initialize shuffle array
  (PRIMORDIAL)   ;execute primordial phase
  (STATISTICS)   ;print out statistics
  (if (and (<= gen_count cutpopgen) (evenp gen_count))
  ;start cutting population in half from 2nd
  ;generation  up to and including cutpopgen
      (setf popsize (round (/ (float popsize) 2)))))
                          ;cut population in half
(progn
  (SHUFFLE_POP shuffle)   ;initialize shuffle array
  (JUXTAPOSITIONAL)   ;execute juxtapositional phase
  (STATISTICS))))  ;print out statistics
  (close data)
  (close stat))   ;close the statistical and data files

;=========================PRIMORDIAL=====================================
;uses deterministic selection to reproduce a new pop (no cut or splice)
(defun PRIMORDIAL (&aux position step)
  (format t "~%~%PRIMORDIAL~%")   ;print out wht phase the program is in
  (setf oldpop (copy newpop))   ;copy over all pop info to oldpop
  (setf newpop (make-array popsize))
  (SETUP_METER)
  (setf step (round (/ (float popsize) 20)))
  ;setup disp meter stepsize-meter is 20 chars wide
  (do ((i 0 (1+ i)))
      ((>= i popsize))
    (if (= (rem i step) 0)   ;if even multiple of step, then
```

```
        (princ "*"))   ;print a mark for 1/20th of population processed
      (setq position (DET_SELECTION))   ;select individual
      (setf (aref newpop i) (aref oldpop position)))
    ;set newpop to the individual
    (OBJFUNC))   ;call short objective function

;=========================JUXTAPOSITIONAL==============================
;uses deterministic selection and cut and splice to produce new population
(defun JUXTAPOSITIONAL (&aux position1 position2 mate1 mate2 cross count
num_of_children step)
  (format t "~%~%JUXTAPOSITIONAL~%")  ;print out what phase the program is in
  (setf oldpop (copy newpop))   ;copy over all pop info to oldpop
  (SETUP_METER)
  (setf step (round (/ (float popsize) 20)))
  ;setup disp meter stepsize-meter is 20 chars wide
  (RESET_POP newpop)   ;reset shuffle array
  (setf count 0)   ;reset counter
  (do ()   ;add # of children each time through the loop
      ((>= count popsize))
    (setf position1 (DET_SELECTION))   ;select position1
    (setf position2 (DET_SELECTION))   ;select position2
    (setq mate1 (GET_CHROMOSOME oldpop position1))
  ;get chromosome of mate1 @ position1
    (setq mate2 (GET_CHROMOSOME oldpop position2))
  ;get chromosome of mate2 @ position2
    (setf cross (CROSSOVER mate1 mate2 cut_prob splice_prob))
  ;perform crossover of the two
    (setf num_of_children (list-length cross))
  ;get # of children from resulting cross
    (do ((k 0 (1+ k)))
 ((or (>= k num_of_children) (>= count popsize)))
  ;copy over children to newpop
      (setf (population_member-chrom (aref newpop count))
    (MUTATION (nth k cross)))
      (if (= (rem count step) 0)   ;if even multiple of step, then
  (princ "*"))   ;print a mark for 1/20th of population processed
      (setf count (1+ count))))
  (OBJFUNC t))   ;use long objectve function-lookup table for
  ;new chromosomes formed
```

A.4 File: objfunc.lisp

The file objfunc.lisp contains the code to perform subfunction table look up.

```
;=========================FILE-OBJFUNC.LISP=================================
;=This file contains functions related to subfunction look-up, sacaling  =
;=and summation as well as setting the overall member fitnesses          =
;===========================================================================

;==================OBJFUNC==================================================
;is responsible for accessing and decoding population members fitness via
;decode operators and set the overall population member's fitness
(defun OBJFUNC (&optional calc_val_flag &aux chromosome step)
  (RESET_STAT_INFO)
  (format t "~%~% OBJFUNC/STATISTICS ")  ;let user know whether
  (if calc_val_flag
      (format t "(set fitness & update stats)~%")
  ;long lookup table is being used or
      (format t "(update stats only)~%"))
  ;short no lookup table is being used
  (SETUP_METER)   ;sets up to display ammount of pop processed
  (setf step (round (/ (float popsize) 20)))
  ;setup disp meter stepsize-meter is 20 chars wide
  (do ((i 0 (1+ i)))
       ((>= i popsize))
    (if (= (rem i step) 0)   ;if even multiple of step, then
        (princ "*"))   ;print a mark for 1/20th of population processed
    (if calc_val_flag   ;this flag is set if decode is to be used
        (progn
  (setf chromosome (GET_CHROMOSOME newpop i))
  (setf chromosome (EXTRACT chromosome))
  (if (not pspe)  ;is pspe flag is not set then
      (setf chromosome (FILL_NIL_POSITIONS chromosome)))
  ;fill in unspecified positions w/ std_fill array
  (setf chromosome (SUBFUNCTION_DECODE chromosome))
  ;get bits & decode info for subfunctions
  (setf chromosome (SET_SUBFUNC_VALUE chromosome))
  ;generate subfunction value list
  (setf (population_member-subf_list (aref newpop i)) chromosome))
  ;save subfunction list as part of chromosome
(progn   ;if value does not need to be calculated
  (setf chromosome (GET_SUBFUNCTION_LIST newpop i))
  ;get just update the statistics
  ))
    (setf (population_member-fitness (aref newpop i))
  (eval (push '+ chromosome)))
  ;calc overall fitness (add up all subfunc vals)
    (UPDATE_STATS i))   ;& update stats with them
  (fresh-line))
```

```
;=========================SET_SUBFUNC_VALUE============================
;table lookup for list of subfunctions the list resembles the following
;((lookup_table# scale factor (subfunc bits))(lookup_table#.....()).....())
(defun SET_SUBFUNC_VALUE (sub_info_list &aux subf_string subfunct_table
  scale_factor subfunct_table# ret_val_list
  num_of_decode)
  (setf num_of_decode (list-length sub_info_list))
  ;get # of subfunctions to decode
  (setf ret_val_list (make-list num_of_decode))
  ;setup list to be returned of subfunction
  ;decoded values
  (do ((subf_num 0 (1+ subf_num)))
      ((>= subf_num num_of_decode))   ;run through all the input list
    (setf subfunct_table# (nth 0 (nth subf_num sub_info_list)))
  ;get lookup table#
    (setf scale_factor (nth 1 (nth subf_num sub_info_list)))
  ;get lookup table#
    (setf subf_string (nth 2 (nth subf_num sub_info_list)))
  ;get string to be matched
    (setf subfunct_table (nth subfunct_table# lookup_table))
  ;access the lookup table specified by subfunct_table#
    (do ((i 0 (1+ i)))
 ((>= i (list-length subfunct_table)))
  ;go through all the lookup table entries
      (if (equal (nth 0 (nth i subfunct_table)) subf_string)
  ;check for a match of decoded lists
  (setf (nth subf_num ret_val_list)
  ;if there is a match set the subfunction to
  ;the coresponding table value
(* (nth 1 (nth i subfunct_table)) scale_factor)))
      (if (equal (nth subf_num ret_val_list) nil)
  (setf (nth subf_num ret_val_list) 0))))
  ;no match in table means string not fully specified
  ret_val_list)   ;or not specified in table, so fitness is zero

;=====================SUBFUNCTION_DECODE================================
;given the bit list will decode from the lookup table the subfunction
;values final list resembles--
;((lookup_table#,scale factor,(subf_bit_list),(....),(....).....(....))
(defun SUBFUNCTION_DECODE (bit_list &aux subf_info scale_factor
   bit_specifier table_specifier temp_list
   bit_value subf_information)
  (setf subf_information '())   ;clear list
  (do ((subf_num 0 (1+ subf_num)))
  ;bit_spec specifies subfunction bits, scale factor
  ;& lookup table for each subfunction
      ((>= subf_num (list-length bit_spec)))
```

```
  (setf subf_info (nth subf_num bit_spec))
 ;get single subfunction specifier off bit_spec
  (setf scale_factor (bit_cluster-scale_factor subf_info))
 ;get scale factor
  (setf bit_specifier (bit_cluster-bit_specifier subf_info))
 ;get bit specifier (position numbers)
  (setf table_specifier (bit_cluster-table_specifier subf_info))
 ;get table specifier
  (setf temp_list '())
  (do ((i 0 (1+ i)))   ;get all specified bits & put them in a list
((>= i (list-length bit_specifier)))
    (setf bit_value (nth i bit_specifier))   ;get bit value
    (push (nth bit_value bit_list) temp_list))  ;put it on list
  (setf temp_list (list (reverse temp_list)))
  (push scale_factor temp_list)         ;put scale factor on list
  (push table_specifier temp_list)   ;and table specifier
  (push temp_list subf_information))
 ;put all this on the subfunction information list
 (reverse subf_information))
```

A.5 File: ops.lisp

The file ops.lisp contains the various selection functions and the genetic operators.

```
;============================FILE-OPS.LISP================================
;=This file contains all the mGA genetic operators.                      =
;=============================================================================

;=====================CUT=====================================================
;takes chromosome cuts it & returns the 2 pieces
(defun CUT (chromosome &aux rd len)
  (setf len (list-length chromosome));cut position is random
  (if (> len 1)
      (setf rd (RND 1 (1- len))))
  (cond
    ((<= len 1) (list chromosome '()))
    ((list (subseq chromosome 0 rd) (subseq chromosome rd len)))))

;=============================SPLICE==========================================
;takes 2 pieces puts them together and returns the result
(defun SPLICE (chrom1 chrom2)
  (append chrom1 chrom2))
;=============================MUTATE==========================================
;changes a single bits value 1 to 0 or vice versa at a rate of probability
(defun MUTATE (gene probability)
  (cond
    ((FLIP probability) (setf gene (COMPLEMENT_BIT gene))))
  gene)

;=============================MUTATION========================================
;takes a chromosome and checks for a mutation at each bit
(defun MUTATION (chrom &aux len)
  (cond
    ((not (zerop mut_prob))
                          ;if mutation rate is not zero
     (setf len (list-length chrom))
                          ;mutation is determined by mut_prob
     (do ((i 1 (1+ i)))
         ((> i len))      ;run through entire chrom
       (setf (nth 1 (nth (1- i) chrom))
             (MUTATE (nth 1 (nth (1- i) chrom)) mut_prob)))))
  chrom)                  ;return chrom

;=========================CUT_AND_STACK=======================================
;takes 2 strings mate1 and mate2 cuts them and returns list of cut pieces
(defun CUT_AND_STACK (mate1 mate2 cut_prob &aux loc_stack)
  (cond
    ((FLIP (* cut_prob (list-length mate1)))  ;check for cut
     (setq mate1 (CUT mate1))
```

```
                          ;if so then cut mate1
     (push (cadr mate1) loc_stack)
                          ;put second piece of mate1 on stack
     (setq mate1 (pop mate1))))
                          ;set mate1 to first piece
  (cond
    ((FLIP (* cut_prob (list-length mate2)))   ;check for cut
     (progn
       (setq mate2 (CUT mate2))
                          ;if so then cut mate2
       (push (cadr mate2) loc_stack)
                          ;put second piece of mate2 on the stack
       (push (pop mate2) loc_stack)))
                          ;put first piece of mate2 on the stack
    ((push mate2 loc_stack)))
                          ;otherwise put mate2 (uncut) onto the stack
  (push mate1 loc_stack)) ;put mate1 (cut or uncut) onto the
                          ;stack & return it

;============================CROSSOVER=================================
(defun CROSSOVER (mate1 mate2 cut_prob splice_prob &aux child1 child2
                  child3 child4 cross)
  (setf stack (CUT_AND_STACK mate1 mate2 cut_prob))
                          ;generate stack of cut pieces
  (setf child1 (SPLICE_TESTER splice_prob ))
                          ;check for splice on child1
  (setf child2 (SPLICE_TESTER splice_prob ))
                          ;check for splice on child2
  (setf child3 (SPLICE_TESTER splice_prob ))
                          ;check for splice on child3
  (setf child4 (SPLICE_TESTER splice_prob ))
                          ;check for splice on child4
  (setf cross (list child1 child2 child3 child4 ))
                          ;return list of children
  (remove nil cross))


;======================SPLICE_TESTER=================================
;tests for splice between CUT_AND_STACK(ed) pieces list name -stack
;example stack (resembles) --> '((a b c) (d e f) (h i) (j k l))  execute-
;(SPLICE_TESTER 1.0) (100% prob) (a b c d e f) (top two members spliced)
;(SPLICE_TESTER 0) (0% prob)  (abc) (top member no splice)
(defun SPLICE_TESTER (splice_prob)
  (cond   ;make sure stack is not empty
    (stack (cond
      ((FLIP splice_prob)
  ;if splice then splice two pieces of stack
       (SPLICE (pop stack) (pop stack)))
```

```
      ((pop stack))))   ;otherwise return top member of stack
    ))

;============================INIT_SELECTION================================
;compares successive shuffled population members amd returns the fittest
(defun INIT_SELECTION (&aux first second fittest)
  (if (>= pick popsize)   ;if array has been gone through just reset
      (setf pick 0))      ;the pointer to zero
  (setf first pick)       ;get the first
  (setf second (1+ pick)) ;get the second
  (setf pick (+ pick 2))  ;inrement location +2
  (if (>= (GET_FITNESS oldpop first)
          (GET_FITNESS oldpop second))  ;compare fitnesses
      (setf fittest first)              ;& return fittest
      (setf fittest second))
  fittest)

;=====================DET_SELECTION=======================================
;controls type of selection routine to be used
(defun DET_SELECTION (&aux fittest)
  (if (< loopvar init_select_gen)
      (setf fittest (INIT_SELECTION))
                          ;use INIT_SELECTION for first three generations
      (progn
        (if (equal thres nil)  ;check thres flag
            (setf fittest (NORM_SELECTION shuffle))
                          ;thres flag is not set use normal selection
            (setf fittest (THRESH_SELECTION)))))
                          ;if it is set use thres selection
  fittest)

;============================NORM_SELECTION================================
(defun NORM_SELECTION (shuffle &aux first second fittest)
  (if (>= pick (1- popsize))
      (progn
        (setf pick 0)     ;when you reach the end of the array then,
        (SHUFFLE_POP shuffle)))   ;re-shuffle it
  (setf first (aref shuffle pick))   ;select two guys from it
  (setf second (aref shuffle (1+ pick)))
  (setf pick (+ pick 2))  ;increment pointer
  (if (< (GET_FITNESS oldpop first) (GET_FITNESS oldpop second))
                          ;compare the fitnesses of the two
      (setf fittest second)
      (setf fittest first))
  fittest)                ;return the fittest

;=========================THRESH_SELECTION================================
(defun THRESH_SELECTION (&aux first second chrom1 chrom2 fittest threshold
```

```
                        flag stop_position position pts_alike length1 length2)
(if (>= pick (- popsize shufnum))
                        ;if there's not enough room at the end of the pop
    (progn              ;to run through shufnum# members, then reset the
      (setf pick 0)     ;pick and the shuffle array
      (SHUFFLE_POP shuffle)))
(setf stop_position (+ pick shufnum))
(setf first (aref shuffle pick))
(setf fittest first)
(setf chrom1 (copy (GET_CHROMOSOME oldpop first)))
                        ;get chrom with removed duplicates
(setf chrom1 (remove-duplicates chrom1 :key 'car :from-end 't))
(setf length1 (list-length chrom1))
(setf position (1+ pick))
(do ((i 1 (1+ i)))((or flag (>=  position stop_position)))
  (setf position (+ i pick))
                        ;exit if thres(hold) is reached or if shufnum
                        ;members have been looked at
  (setf second (aref shuffle position))
  (setf chrom2 (copy (GET_CHROMOSOME oldpop second)))
                        ;get chrom with removed duplicates
  (setf chrom2 (remove-duplicates chrom2 :key 'car :from-end 't))
  (setf length2 (list-length chrom2))
  (setf threshold (/ (* length1 length2) shufnum))
                        ;calculate threshold value
  (setf pts_alike (list-length (intersection chrom1 chrom2 :key 'car)))
  (cond ((>= pts_alike threshold)
                        ;calculate the # of common points
         (progn         ;if # of common pts > threshold value
           (setf flag 't)
           (if (>= (GET_FITNESS oldpop first) (GET_FITNESS oldpop second))
                        ;compare fitnesses
               (setf fittest first)     ;& return most fit member
               (setf fittest second))
           (setf (aref shuffle (+ pick i)) (aref shuffle (1+ pick)))
                        ;swap the two positions
           (setf (aref shuffle (1+ pick)) second)))))
(setf pick (+ pick 2))  ;increment pointer
fittest)                ;return the fittest
```

A.6 File: setup.lisp

The file setup.lisp contains the initialization code for mGA1.0.

```
;============================FILE-SETUP.LISP=============================
;=This file contains the clode for setting up all the parameters od the  =
;=mGA via a user specified setup file. The routines are primarily        =
;=involved with accesssing the file, pattern matching and the setting    =
;=the specified parameter. The exeption is MAKE_NEW_POP (see below)      =
;========================================================================
;=========================MAKE_NEW_POP===================================
(defun MAKE_NEW_POP(num_positions max_length &optional copies &aux count
    bit_positions position_list bit_list bits max_size)
  (format t "~%Making Population of building block size ~A" num_positions)
  (format t "~%and chromosome length ~A" max_length)
  (cond ((not copies)
 (setf copies 1)))
  (setf bit_positions (expt 2 num_positions))
  ;calc bit position counter
  (setf max_size
(* copies (* bit_positions (CHOOSE max_length num_positions))))
  ;calc pop size for blocks
  (setf newpop (make-array max_size))
  ;create newpop array
  (setf position_list (make-list num_positions))
  ;create position list
  (SET_ASC_LIST position_list 0)
  ;initialize to (0 1 2 ...num_positions-1)
  (setf bit_list (make-list num_positions))   ;create bit_list
  (setf count 0)
  (do ()
      ((>= count (1- max_size)))  ;generate every block
    (SET_LIST bit_list 0) ;reset bit_list to all zeros
    (do ((j 0 (1+ j)))   ;run through all bit combinations
 ((>= j bit_positions))
      (setf bits nil)   ;reset temporary bit/value holder
      (if (/= j 0)   ;on second and subseq passes increment bit_list
  (N_ARY_COUNT bit_list 1 (1- num_positions)))
      (do ((k 0 (1+ k)))
   ((>= k num_positions)) ;create pos/value list
        (push (list (nth k position_list) (nth k bit_list)) bits))
      (do ((m 0 (1+ m)))
  ((>= m copies)) ;make copies if specified
(setf (aref newpop count) (make-population_member))
  ;create structure population_memeber
(setf (population_member-chrom
(aref newpop count)) (reverse bits))
  ;set member to the bit/val list
(setf count (1+ count)))
```

```
      )
    (COMBINATION max_length num_positions  position_list ))
  ;increment to next position
  (format t "~%Population Size is now ~A~%" count)
  (setf popsize count))   ;return population size created

;---------->>>>>>>>>LOAD PROGRAM STARTS HERE<<<<<<<<<<<<<----------
;the purpose is to setup all the global variables given a setup file

;================================SETUP_GA=================================
(defun SETUP_GA ()
  (setf first_time 1)
  (fresh-line)
  (format t "enter name of setup file--") ;
  (setf setup_file (read))   ;get name of setup file from user
  (fresh-line)
  (setf instream (open setup_file :direction :input))
  ;open setup file for reading
  (setf indata (read instream))   ;read from setup file
  (do () ((equal indata 'end))   ;read until end marker is reached
    (if (equal indata 'OBJ)   ;if OBJ marker is reached go to SETUP_OBJ
        (SETUP_OBJ))
    (if (equal indata 'POPULATION)   ;population setup marker
        (SETUP_POP))
    (setf indata (read instream))
    )
  (close instream)   ;close setup file
  (format t "setup file ~A is loaded~%" setup_file))
  ;inform that setup file has been loaded

;============================SETUP_OBJ=====================================
;sets up objective function parameters
(defun SETUP_OBJ (&aux inlist table# scale subfunction)
  (setf subfunction 0)
  (setf num_tables 0)
  (setf bit_spec (make-list num_subfunctions))
  ;create bit specifier array
  (format t "setting up objective function~%")
  ;tell user what's being set up
  (setf indata (read instream))   ;read from file
  (do () ((equal indata '***))   ;read until *** patterm is found
    (if (equal indata 'subfunction_bits)  ;if setup for subfunction bits
        (progn
  (setf inlist nil)   ;reset all parameters
  (setf table# nil)
  (setf scale nil)
  (setf indata (read instream))   ;read next data item from file
  (do () ((not (numberp indata))) ;read until a # is not encountered
```

```
    (push indata inlist)
  ;make it into a list of numbers
  ;(the subfunction bits)
    (setf indata (read instream)))   ;read next
  (if (equal indata 'ltable)   ;if lookup table label then,
      (progn
        (setf indata (read instream))   ;read next value
(setf table# indata)   ;get lookup table number
(if (> table# num_tables) ;keep track of # of lookup tables
    (setf num_tables table#))
(setf indata (read instream))   ;read next file data
))
  (if (equal indata 'scale)   ;if scale factor
      (progn
        (setf indata (read instream))   ;read in data
(setf scale indata)))   ;assign it to scale
  ;now that all data has been gotten for 1
  ;subfunction setup bit_spec structure for it
  (setf (nth subfunction bit_spec)
(make-bit_cluster :bit_specifier (reverse inlist)
  :table_specifier table#
  :scale_factor scale))
  (fresh-line)
  (setf subfunction (1+ subfunction))))
  ;increment subfunction counter

    (if (equal indata 'table)
  ;if lookup table then go to lookup table setup routine
        (SETUP_TABLE))
    (if (equal indata 'template)
  ;if template then go to template setup section
        (SETUP_TEMPLATE))
    (setf indata (read instream)) ;read next data
    ))

;====================SETUP_TEMPLATE=======================================
;sets up competitive template
(defun SETUP_TEMPLATE ()  ;
  (setf std_fill (make-list bits_per_chrom))
  ;setup size of standard fill array
  (do ((i 0 (1+ i)))
      ((>= i bits_per_chrom))   ;get all the bits
    (setf (nth i std_fill) (read instream))))   ;set the bits

;====================SETUP_TABLE=========================================
;sets up lookup table made of two parts-bits and coresponding value
(defun SETUP_TABLE (&aux table_number inlist data_value)
  (format t "setting up lookup table~%")
```

```
  (cond
    (first_time   ;if the first time through then create
  ;the lookup table
     (setf lookup_table (make-list (1+ num_tables)))
     (setf first_time nil)))
  (setf table_number (read instream))   ;get table number
  (setf indata (read instream))           ;read next file data
  (do () ((equal indata 'end_table))   ;read until the end of that table
    (setf inlist nil)
    (setf data_value nil)          ;get data value from file
    (if (equal indata 'bvalue)   ;if bit value
        (progn
  (setf indata (read instream))
  (do () ((not (numberp indata)))
    (push indata inlist)
    (setf indata (read instream)))   ;assign to bit list
  (if (equal indata 'dvalue)           ;if data value
      (progn
        (setf indata (read instream))
(setf data_value indata)  ;assign data value
(setf indata (read instream))))
  (push  (list (reverse inlist)  data_value)
 (nth table_number lookup_table))
  ;put bits & coresponding value on lookup table
  ))))

;============================SETUP_POP=====================================
;this routine sets up population parameters
(defun SETUP_POP ()
  (format t "setting up population parameters~%")
   (setf bits_per_chrom nil)
  (setf maxgen nil)
  (setf num_subfunctions nil)
  (setf cookgen nil)
  (setf shufnum nil)
  (setf seed nil)
  (setf cutpopgen nil)
  (setf cut_prob 0)
  (setf splice_prob 0)
  (setf mut_prob nil)
  (setf pspe nil)
  (setf thres nil)
  (setf garbage_collect 1)
  (setf bldg_blk_size nil)
  (setf member_copies 1)
  (setf init_select_gen 1)
  (setf indata (read instream))
  (do () ((equal indata '***))   ;read until star pattern is encountered
```

```
 (cond ((equal indata 'chrom_length)
 (setf indata (read instream))
 (setf bits_per_chrom indata))
((equal indata 'maxgen)
 (setf indata (read instream))
 (setf maxgen indata))
((equal indata 'num_subfunctions)
 (setf indata (read instream))
 (setf num_subfunctions indata))
((equal indata 'cookgen)
 (setf indata (read instream))
 (setf cookgen indata))
((equal indata 'shufnum)
 (setf indata (read instream))
 (setf shufnum indata))
((equal indata 'seed)
 (setf indata (read instream))
 (setf seed indata))
((equal indata 'cutpopgen)
 (setf indata (read instream))
 (setf cutpopgen indata))
((equal indata 'cut_prob)
(setf indata (read instream))
(setf cut_prob indata))
((equal indata 'splice_prob)
 (setf indata (read instream))
 (setf splice_prob indata))
((equal indata 'mut_prob)
 (setf indata (read instream))
 (setf mut_prob indata))
((equal indata 'pspe)
 (setf indata (read instream))
 (setf pspe indata))
((equal indata 'thres)
 (setf indata (read instream))
 (setf thres indata))
((equal indata 'bldg_blk_size)
 (setf indata (read instream))
 (setf bldg_blk_size indata))
((equal indata 'init_select_gen)
 (setf indata (read instream))
 (setf init_select_gen indata))
((equal indata 'member_copies)
 (setf indata (read instream))
 (setf member_copies indata))
((equal indata 'garbage_collect)
 (setf indata (read instream))
 (setf garbage_collect indata)))
```

```
(setf indata (read instream))))       ;get next data
```

A.7 File: stats.lisp

The file stats.lisp contains statistical and reporting functions for mGA1.0.

```
;============================FILE-STATS.LISP=================================
;=This file contains the statistical and reporting related code           =
;============================================================================

;=================INITIAL_OUTPUT=============================================
;writes out initial data to data file, statistics file and screen
(defun INITIAL_OUTPUT ()
  (format screen&file "~3%RUN PARAMETERS~%")
  (format screen&file "Maxgen = ~A~%" maxgen)
  (format screen&file "Cookgen = ~A~%"cookgen)
  (format screen&file "Cutpopgen = ~A~%" cutpopgen)
  (format screen&file "Init_select_gen = ~A~%" init_select_gen)
  (format screen&file "Threshold = ~A~%" thres)
  (format screen&file "Shuffle-Down Number = ~A~%" shufnum)
  (format screen&file "Random Seed = ~A~%"  seed)
  (format screen&file "pspe = ~A~%" pspe)
  (format screen&file "Competitive Template- ~A~3%" std_fill)
  (format data "Generation Popsize Maximum_Fitness Average_Fitness~%"))

;====================UPDATE_STATS============================================
(defun UPDATE_STATS (pop_member &aux good_count subf_list total_fitness)
  (setf total_fitness 0)  ;reset counter variables
  (setf good_count 0)
  (setf subf_list (GET_SUBFUNCTION_LIST newpop pop_member))
  (setf total_fitness (GET_FITNESS newpop pop_member))
  (do ((j 0 (1+ j)))
      ((>= j (list-length subf_list)))
  ;run through all the subfunctions for the subf_list
    (if (>= (nth j subf_list) (- (nth j subf_pos_max) .01))
  ;check for max subfunctions
        (progn
  (setf good_count (1+ good_count))
  ;local counter to keep track of max # of optimal
  ;subfunctions in single chrom
  (setf (aref vertnum_spot j) (1+ (aref vertnum_spot j)))
  ;keeps track of max# of optimal subfunctions on
  ;a per subfunction basis
  (setf numgood (1+ numgood)))))
  ;count total number of optimal subfunctions
  (setf sumfitness (+ sumfitness total_fitness))
  ;keep track of the total sum fitness
  (if (> total_fitness (nth 0 maxfitness))
  ;keep track of max fitness in population
      (progn
(setf (nth 0 maxfitness) total_fitness)
```

```
(setf (nth 1 maxfitness) pop_member)
))
  (if (< total_fitness minfitness)
      (setf minfitness total_fitness))
  ;keep track of min fitness in population
  (if (> good_count max_numgood)
  ;keep track of max# of optimal subfunctions
  ;in single population member
      (setf max_numgood good_count)))

;==================RESET_STAT_INFO=======================================
(defun RESET_STAT_INFO () ;resets all statistical counters
  (setf vertnum_spot (make-array num_subfunctions))
  (setf maxfitness (make-list 2))
  (setf (nth 0 maxfitness) 0)   ;maximum fitness in the population
  (setf (nth 1 maxfitness) 1)
  (setf minfitness subf_pos_max)
  (setf minfitness (eval (push '+ minfitness)))
  ;minimum fitness in the population
  (setf sumfitness 0)   ;total fitness of population
  (setf numgood 0)   ;total number of optimal optimal subfunctions
  ;in population
  (setf max_numgood 0)   ;max optimal subfunctions in a single chromosome
  (array-initialize vertnum_spot 0))
  ;number of optimal subfunctions in each
  ;subfunction position

;============================STATISTICS=================================
;prints out population statistics  to file and screen each generation
(defun STATISTICS ()
  (format t "~%~%running STATISTICS~%")
  (setf avg_fitness (float (/ sumfitness popsize)))
  ;calc average fitness
  (format screen&file "Generation# ~A~%" loopvar)
  ;print out various statistics
  (format screen&file "Popsize = ~A~%" popsize)
  (format screen&file "Minfitness = ~A~%" minfitness)
  (format screen&file "Maxfitness = ~A~%" (nth 0 maxfitness))
  (format screen&file "Average fitness = ~A~%" avg_fitness)
  (format screen&file "Maximum number of optimal subfunctions = ~A~%"
  max_numgood)
  (format screen&file "Total number of optimal subfunctions = ~A~%"
  numgood)
  (format screen&file "Average number of optimal subfunctions = ~A~%"
  (float (/ numgood popsize)))
  (format data "~A" loopvar)
  (format data " ~A ~A ~A ~%" popsize (nth 0 maxfitness) avg_fitness)
  (do ((i 0 (1+ i))) ;prints out information per subfunctional
```

```
        ((>= i num_subfunctions))
(format screen&file
"Total and Average Number of optimal subfunctions in position ~A = ~A and ~A~%"
(1+ i) (aref vertnum_spot i)
(/ (float (aref vertnum_spot i)) popsize)))
  (format screen&file "~%Best solution so far =~A~%"
  (FILL_NIL_POSITIONS
    (EXTRACT (GET_CHROMOSOME newpop (nth 1 maxfitness)))))
  (format screen&file "Fitness = ~A~%" (nth 0 maxfitness)))

;=========================SETUP_SUBFUNC_MAX==============================
;this function sets up a list that has the maximum value each subfunction
;in that position can obtain (scale factor applied)
(defun SETUP_SUBFUNC_MAX (&aux ret_list subf_max subf_info
  scale_factor table_specifier subfunct_table)
  (setf ret_list '())   ;initialize return list
   (do ((i 0 (1+ i)))   ;run through all the subfunctions
      ((>= i (list-length bit_spec)))
    (setf subf_max 0)   ;set max to lowest possible val
    (setf subf_info (nth i bit_spec))
  ;get subfunction specifier
    (setf scale_factor (bit_cluster-scale_factor subf_info))
  ;get scale factor from list
    (setf table_specifier (bit_cluster-table_specifier subf_info))
  ;get lookup table specifier off list
    (setf subfunct_table (nth table_specifier lookup_table))
  ;access the single lookup table
    (do ((j 0 (1+ j)))   ;go through entire lookup table
((>= j (list-length subfunct_table)))
  ;and find the highest value in the table
      (if (> (nth 1 (nth j subfunct_table)) subf_max)
  (setf subf_max (nth 1 (nth j subfunct_table)))))
    (push (* scale_factor subf_max) ret_list))
  ;put it on the list
   (reverse ret_list))   ;set returned lists to proper order
```