PRINTED LISTING OF ALL FILES
ASSOCIATED WITH THE GALOPP SYSTEM
INCLUDING EXAMPLE APPLICATION, INPUT, AND MASTER FILES

(These files are not listed here, but are included on a disk furnished separately or on the archive server.)

ithruj2int routine (see utility.c) is used to translate each chromosome into its associated vector.

The fitness for each chromosome is simply the sum of the squares of these integers. This example application will function for any chromosome length.

Final Comments

SGA-C is intended to be a simple program for first-time GA experimentation. It is not intended to be definitive in terms of its efficiency or the grace of its implementation. The authors are interested in the comments, criticisms, and bug reports from SGA-C users, so that the code can be refined for easier use in subsequent versions.

Please email your comments to rob@galab2.mh.ua.edu, or write to TCGA:

The Clearinghouse for Genetic Algorithms
The University of Alabama
Department of Engineering Mechanics
P.O. Box 870278
Tuscaloosa, Alabama 35487

*Acknowledgments
The authors gratefully acknowledge support provided by NASA under
Grant NGT--50224 and support provided by the National Science Foundation under
Grant CTS--8451610. We also thank Hillol Kargupta for donating his tournament selection implementation.

The population size (int).
The chromosome length (int).
Print the chromosome strings each generation (y/n)?
The maximum number of generations for the run (int).
The probability of crossover (float).
The probability of mutation (float).
Application-specific input, if any.
The seed for the random number generator (float).

Chromosome Representation and Memory Utilization

SGA-C uses a machine level representation of bit strings to increase efficiency. This allows crossover and mutation to be implemented as binary masking operations (see operators.c). Every chromosome (as well as the population arrays and some auxiliary memory space) are allocated dynamically at run time. The dynamic memory allocation scheme allocates a sufficient number of unsigned integers for each population member to store bits for the user-specified chromosome length. Because of this feature, it is extremely important that {BITS_PER_BYTE be properly set (in sga.h and external.h) for your machine's hardware and C compiler.

Implementing Application-Specific Routines

To implement a specific application, you should only have to change the file app.c.

The section on app.c describes the routines in app.c in detail.
If you use additional variables for your specific problem, the easiest method of making them available to other program units is to declare them in sga.h and external.h. However, take care that you do not redeclare existing variables.

Three example applications files are included in the SGA-C distribution.

The file app.c performs the simple example problem included with the Pascal version:

> finding the maximum of x^10, where x is an integer interpretation of a chromosome.

The larger version of the same problem, as described in the Goldberg text, is provided as app1.c, which maximizes the function x^30, where x is an integer interpretation of a chromosome.

A slightly more complex application is include in app2.c. This application illustrates two features that have been added to SGA-C. The first of these is the ithruj2int function, which converts bits i through j in a chromosome to an integer. The second new feature is the utility pointer that is associated with each population member. The example application interprets each chromosome as a set of concatenated integers in binary form. The length of these integer fields is determined by the user-specified value of field_size, which is read in by the function app_data(). The field size must be less than the smallest of the chromosome length and the length of an unsigned integer. An integer array for storing the interpreted form of each chromosome is dynamically allocated and assigned to the chromosome's utility pointer in app_malloc(). The

[app.c] contains application dependent routines.

Unless you need to change the basic operation of the GA itself, you should only have to alter this file. Further instructions for altering the SGA application are included in the description of routine app.

[application()] should contain any application-specific computations needed before each GA cycle. It is called by main().

[app_data()] should ask for and read in any application-specific information. This routine is called by init_data().

[app_malloc()] should perform any application-specific calls to malloc() to dynamically allocate memory. This routine is called by initmalloc().

[app_free()] should perform any application-specific calls to free(), for release of dynamically allocated memory. This routine is called by freeall().

[app_init()] should perform any application-specific initialization needed. It is called by initialize().

[app_initreport()] should print out an application-specific initial report before the start of generation cycles. This routine is called by initialize().

[app_report()] should print out any application-specific output after each GA cycle. It is called by report().

[app_stats()] should perform any application-specific statistical calculations. It is called by statistics().

[objfunc(critter)] The objective function for the specific application. The variable critter is a pointer to an individual (a GA population member), to which this routine must assign a fitness. This routine is called by generation().

[Makefile] is a UNIX makefile for SGA-C.

New Features of SGA-C

SGA-C has several features that differ from those of the Pascal version. One is the ability to name the input and output files on the command line, i.e.,

sga my.input my.output.

If either of these files is not named on the command line, SGA-C assumes stdin and stdout, respectively.

Another new feature of SGA-C is its method of representing chromosomes in memory. SGA-C stores its chromosomes in bit strings at the machine level. Input-output and chromosome storage in SGA-C are discussed in the following sections.

Input-Output

SGA-C allows for multiple GA runs. When the program is executed, the user is first prompted for the number of GA runs to be performed. After this, the quantity of input needed depends on the selection routine chosen at compile-time, and any application-specific information required. When compiled with roulette wheel selection, the input requested from the user is as follows:

The number of GA runs to be performed (int).

subtractive method specified by Knuth:81.

[rnd(low,high)] returns an uniformly-distributed integer between low and high.

[rndreal(low,high)] returns an uniformly-distributed floating point number between low and high.

[flip(p)] flips a biased coin, returning 1 with probability p, and 0 with probability 1-p.

[advance_random()] generates a new batch of 55 random numbers.

[randomize()] asks the user for a random number seed.

[warmup_random()] primes the random number generator.

[noise(mu, sigma)] generates a normal random variable with mean mu and standard deviation sigma. This routine is not currently used in SGA-C, and is only included as a general utility.

[randomnormaldeviate()] is a utility routine used by noise. It computes a standard normal random variable.

[initrandomnormaldeviate()] initialization routine for randomnormaldeviate().

[report.c] contains routines used to print a report from each cycle ofSGA-C's operation.

[report()] controls overall reporting.

[writepop()] writes out the population at every generation.

[writechrom()] writes out the chromosome as a string of ones and zeroes. In the current implementation, the most significant bit is the rightmost bit printed.

Three selection routines are included with the SGA-C distribution:

[rselect.c] contains routines for roulette-wheel selection.

[srselect.c] contains the routines for stochastic-remainder selection (Booker:82).

[tselect.c] contains the routines for tournament selection (Brindle:81a). Tournaments of any size up to the population size can be held with this implementation. [The tournament selection routine included with the distribution was written by Hillol Kargupta, of the University of Alabama.]

For modularity, each selection method is made available as a compile time option. Edit the Makefile to choose a selection method. Each of the three selection files contains the routines select_memory and select_free (called by initmalloc and freeall, respectively), which perform any necessary auxiliary memory handling, and the routines preselect() and select(), which implement the particular selection method.

[stats.c] contains the routine statistics(), which calculates population statistics for each generation.

[utility.c] contains various utility routines.
Of particular interestis the routine ithruj2int(), which returns bits i through j of a chromosome interpreted as an int.

Files Distributed with SGA-C
(EDG NOTE: Some Information OBSOLETE)

The following is an outline of the files distributed with SGA-C, the routines contained in those files, and the structure of the SGA-C distribution.

[sga.h] contains declarations of global variables and structures for SGA-C. This file is included by main().

Both sga.h and external.h have two "defines" set at the top of the files: LINELENGTH, which determines the column width of printed output, and BITS_PER_BYTE, which specifies the number of bits per byte on the machine hardware. LINELENGTH can be set to any desired positive value, but BITS_PER_BYTE must be set to the correct value for your hardware.

[external.h] contains external declarations for inclusion in all source code files except main(). The extern declarations in external.h should match the declarations in sga.h.

[main.c] contains the main SGA program loop, main().

[generate.c] contains generation(), a routine which generates and evaluates a new GA population.

[initial.c] contains routines that are called at the beginning of a GA run.

[initialize()] is the central initialization routine called by main().
[initdata()] is a routine to prompt the user for SGA parameters.
[initpop()] is a routine that generates a random population. Currently, SGA-C includes no facility for using seeded populations.
[initreport()] is a routine that prints a report after initialization and before the first GA cycle.

[memory.c] contains routines for dynamic memory management.

[initmalloc()] is a routine that dynamically allocates space for theGA population and other necessary data structures.
[freeall()] frees all memory allocated by initmalloc().
[nomemory()] prints out a warning statement when a call to malloc() fails.

[operators.c] contains the routines for genetic operators.

[crossover()] performs single-point crossover on two mates, producing two children.
[mutation()] performs a point mutation.

[random.c] contains random number utility programs, including:

[randomperc()] returns a single, uniformly-distributed, real, pseudo-random number between 0 and 1. This routine uses the

APPENDIX SIX: EXCERPTS FROM THE SGA-C V1.1 RELEASE DOCUMENT

NOTE: This appendix describes the SGA-C release on which the GALOPP System was based. The many extensions and revisions, however, have rendered much of this information obsolete. It is included for completeness, and in order to help document the transitions from Goldberg's original Pascal SGA code (in his text-book) to the present system.

SGA-C: A C-language Implementation of a Simple Genetic Algorithm

Robert E. Smith
The University of Alabama
Department of Engineering Mechanics
Tuscaloosa, Alabama 35405

and David E. Goldberg
The University of Illinois
Department of General Engineering
Urbana, Illinois 61801

and Jeff A. Earickson
Alabama Supercomputer Network
The Boeing Company
Huntsville, Alabama 35806

SGA-C Disclaimer: SGA-C is distributed under the terms described in the file NOWARRANTY. These terms are taken from the GNU General Public License. This means that SGA-C has no warranty implied or given, and that the authors assume no liability for damage resulting from its use or misuse.

Introduction

SGA-C is a C-language translation and extension of the original Pascal SGA code presented by Goldberg, 89. It has some additional features, but its operation is essentially the same as that of the original, Pascal version.

This report is included as a concise introduction to the SGA-C distribution. It is presented with the assumptions that the reader has a general understanding of Goldberg's original Pascal SGA code, and a good working knowledge of the C programming language.

The report begins with an outline of the files included in the SGA-C distribution, and the routines they contain. The outline is followed by a discussion of significant features of SGA-C that differ from those of the Pascal version. The report concludes with a discussion of routines that must be altered to implement one's own application in SGA-C.

bestnow	Index (in [0,popsize-1]) of best individual in the current subpopulation (may not be best ever, if not using elitism).
convinterval	Number of generations between printing of convergence statistics.
crowding_factor	DeJong-type crowding factor; 0 means don't use crowding (offspring replace parents); 1 means pick a random survivor to replace with new individual (without replacement); 2 or more means replace CLOSEST (Hamming distance) individual among the 2 or more survivors tested.
incest_reduction	Flag; 0 means no mating restriction in effect; 1 means mates for parent 1 in a crossover will be picked from a pool of 3 possible parent 2's -- the one furthest away (Hamming distance)is chosen as the mate.
stochastic	Flag; 0 means NOT stochastic, so don't reevaluate unchanged individuals; 1 means IS stochastic, so evaluate every individual every generation (reduces sampling error when fitnesses vary with time or are density dependent).
elitism	Flag; 0 means NOT elitist; 1 means IS elitist: best individual is guaranteed at least one spot in next generation.
oldrand[55]	Array of random number generator, so restarts can pick up sequence where it was left off.
jrand	Index used in random number generation.
rndx2	Double used in random number generation.
rndcalcflag	Flag used in random number generation.
alpha_size	Int telling the number of alleles per locus (2 means a bit string). Legal values in each field will range from 0 through alpha_size - 1.
permproblem	Int telling whether is (==1) a permutation problem or not (==0)

ANY USER-DEFINED VARIABLES WHICH MUST BE RECORDED (except utility fields)

Any values written to the file by CallBackFun(), which are added to the end of the "standard" file. User provides the code to write these variables (if any) in a function called app_write_ckp_hdr() in file appxxxxx.c (or whatever the user's application file is called).

These files are read by function ReadCheckPointHeader, whenever a subpopulation is being restored after being checkpointed. (User supplies code to read any USER-DEFINED VARIABLES written by app_write_ckp_hdr() in its corresponding function, app_read_ckp_hdr(), and by app_write_utility() in its corresponding function, app_read_utility()..

	part of the permutation being sought, but rather encode (indirectly) values being sought for other parameters.
fieldlength	(Calculated) length of each field, for permutation-type or mixed-type problems.
pcross	Probability of crossover (per chromosome), in [0.,1.].
pmutation	Probability of mutation (per BIT, for ordinary binary representations, but per CHROMOSOME, for permutation-type or mixed-type representations).
scalemult	Multiple of average fitness to which most fit individual is to be scaled. Also used with same meaning when rank-based selection is used. Value -1 means don't do linear scaling.
conv_measure	Result of a convergence measure; see body of manual.
conv_sigma_coeff	Number of standard deviations above/below mean to be used for convergence measure above.
sigma_trunc	Number of standard deviations from mean at which raw fitnesses are to be truncated in calculating scaled fitness (0.0 means don't don't do sigma truncation).
scaling_window	Number of generations back from which LEAST fit individual will be used to determine scaling of current population (-1 means don't use; 0 means current generation only, 1 means current plus previous one, etc.; max value currently 19).
windowstart	Pointer for tracking generations for window scaling.
windowend	(same as above.)
savedmins[20]	Array of minimums used for window scaling.
sigma	Standard deviation of current raw fitnesses.
neval	Number of chromosomes evaluated in current (sub)population (to date).
lchrom	Length of the chromosome (in bits).
genspercycle	(Manypops only) Number of generations of a subpopulation calculated before it is checkpointed and replaced in memory by another subpopulation.
gen	Number of the current generation (a run initialized from a random population starts gen at 0).
maxgen	Value of gen at which run is to terminate (after writing its checkpoint files, of course). Upon restart, run begins with highest value gen attained in previous run, so maxgen must be larger than that to run at all.
run	(Onepop only) Number of the run being performed (input file can have data for multiple runs "stacked" in one file).
printstrings	Flag, 0 says don't print chromosomes; 1 says do.
nmutation	Number of mutations performed to date (in this subpopulation).
ncross	Number of crossovers performed to date (in this subpopulation).

SUBPOPULATION.

bestfit.neval — Number of evaluations IN THIS SUBPOPULATION ONLY of individuals up through first finding this most fit individual of THIS SUBPOPULATION.

bestfit.generation
Generation number (for THIS SUBPOPULATION) at which this most fit individual OF THIS SUBPOPULATION was first found.

*(bestfit.utility)
If the user defines utility fields, their contents (for the best individual IN THIS SUBPOPULATION to date) is written next, by a call to the user-defined utility writer, app_write_utility(bestfit.utility,fp). Field bestfit.utility (not written) is the pointer to these contents.

one_pop_cum_sumfitness
Sum of the raw fitnesses of all individuals evaluated in this subpopulation to date.

one_pop_sum_best_fitness
Sum of the raw fitness of the best individual of each generation of this subpopulation evaluated to date.

one_pop_best_fitness_count
Count of the number of individuals included in this subpopulation's one_pop_sum_best_fitness. Thus, offline performance for this subpopulation can always be calculated as one_pop_sum_best_fitness / one_pop_best_fitness_count.

one_pop_online_denominator
Number of individuals of this subpopulation whose raw fitnesses are included in one_pop_cum_sumfitness. Thus, online performance for this subpopulation can always be calculated as:
one_pop_cum_sumfitness / one_pop_online_denominator.

startpopnum — Number in [0, npops-1] of the FIRST subpopulation whose calculations will be done by THIS process and processor.

finishpopnum — Number in [0, npops-1] of the LAST subpopulation whose calculations will be done by THIS process and processor.

minraw — Lowest raw fitness in the current (sub)population.

avgraw — Average raw fitness in the current (sub)population.

fit_max — Highest raw fitness in the current (sub)population.

numfields — For permutation-type (or mixed-type) problems only, the total number of fields to be represented on the chromosome.

numpermfields — For permutation-type (or mixed-type) problems only, the number of fields which represent the permutation part of the solution (cities in TSP, tasks in scheduler, etc.). If NOT a mixed-type problem, numpermfields == numfields.

numextrafields — For mixed-type problems only, number of fields which are NOT

III. Files with extension .ckp or .neu

Files with extension .neu are written in file checkhdr.c, by function WriteCheckPointHeader(callname, CallBackFun), where argument fname is declared to be:
char *fname
and CallBackFun is declared as:
BOOL (*CallBackFun).

Files of this type contain state information (essentially, all but the actual individuals in the population) which is needed to restore the program to be able to continue running after a checkpoint dump is taken.

Argument fname is a pointer to the string which contains the name of the file which should be opened for writing the checkpoint header information. It must comply with DOS naming conventions, even on a Unix system, for compatibility reasons. Thus, only an 8-character name, a period, and a 3-character extension are allowed. For Onepops runs, this name is assembled from the user input (file or keyboard), up to 8 characters, using the field "checkptfileprefix", and appending ".neu". If the user does not specify a checkptfileprefix, then the restartfileprefix is used, unless none is specified; in that case, the default "sgackp" is used. For Manypops runs, the procedure is similar, except that the user-entered prefix is limited to 6 characters, and two-digit subpopulation numbers are appended to it, generating file names like "runone00.neu", "runone01.neu", etc. Each subpopulation in a run has its own checkpoint header file.

At the end of a complete cycle of all subpopulations in a Manypops run, all of the checkpoint header files written (with extension ".neu") are RENAMED to have extension ".ckp". This enables a restart to be performed "fairly", regardless of when a run may have been interrupted, by reading only the .ckp files, and discarding any .neu files resulting from partially completed cycles.

File contents are:

Variable Name	Explanation
popsize	number of chromosomes in population being checkpointed
indcnt	number of individuals being recorded in the accompanying .ind file (always popsize, if written by this function, but recorded a second time for future flexibility without changing file format).
chromsize	number of unsigned ints in a single chromosome
bestfit.chrom	chromsize unsigneds, containing the chromosome of the most fit individual found to date IN THIS SUBPOPULATION.
bestfit.fitness	scaled fitness (if scaling used; otherwise, raw fitness) of most fit individual to date IN THIS SUBPOPULATION.
bestfit.init_fitness	unscaled (raw) fitness (from user-defined routine objfunc()) of most fit individual to date IN THIS

At the end of a complete cycle of all subpopulations in a Manypops run, all of the checkpoint individual files written (with extension ".new") are RENAMED to have extension ".ind". This enables a restart to be performed "fairly", regardless of when a run may have been interrupted, by reading only the .ind files, and discarding any .new files resulting from partially completed cycles.

Files of this type contain the population at the time of checkpointing, together with a little additional information at the beginning of the file, as follows:

```
Var. Name         Explanation
__________  _______________________________________________________

version     version number of program writing file (not currently used)

popsize     Number of individuals in this (sub)population

individualsize
            Size (in bytes) of an individual, including the
                  chromsize unsigneds and the utility fields

lchrom      Length of chromosome in bits

bestnow     Index in oldpop of current best individual [0,popsize-1].

(Then, for each individual from 0 to popsize-1:)
{
  chrom     chromsize unsigneds containing the actual chromosome

  fitness   scaled fitness for the chromosome

  init_fitnessraw (unscaled) fitness for the chromosome

  neval     number of evaluations performed in THIS subpopulation
                  this individual was found.

  xsite[0]  site [0,lchrom-1] on parent chromosomes where crossover was done
                  (oneptx) or substring started (twoptx) when this individual
                  was created, or 0 if created some other way.

  xsite[1]  site where substring ended on parent chromosomes if this
                  individual was created by twoptx; 0 otherwise.

  parent[0] position in oldpop [1,popsize]] of parent[0] of this individual
                  (one more than subscript in array).

  parent[1] as above, second parent.

  utility fields
            if utility fields are defined (pointed to by utility on the
                  chromosome), the utility fields' contents are written here by
                  app_write_utility_fields, which the user must define if using
                  utility fields.
}
```

The .stt file contains the following information:

Variable Name	Explanation
all_pops_sumfitness	Sum of raw fitnesses of all individuals evaluated to date
all_pops_sum_best_fitness	Sum of best fitness of each subpopulation at each generation
all_pops_online_denominator	Total number of fitnesses summed into current all_pops_sumfitness
all_pops_best_fitness_count	Total number of best fitnesses summed into current all_pops_sum_best_fitness
all_pops_neval	Total number of evaluations performed to date in all subpopulations
all_pops_bestfit.chrom	Chromosome of best individual found so far in all subpopulations (chromsize unsigneds)
all_pops_bestfit.fitness	Scaled fitness of best individual found so far
all_pops_bestfit.init_fitness	Raw fitness of best individual found so far
all_pops_bestfit.neval	Evaluation number of best individual found so far
all_pops_bestfit.generation	Generation number (of its own subpopulation) in which the best individual found so far was found
all_pops_bestfit	If utility fields are defined (pointed to by utility on the chromosome), the utility fields' contents are written here by app_write_utility_fields, which the user must define if using utility fields.

II. Files with extension .ind or .new

Files with extension .ind or .new are written in file checkwt.c, by function writecheckpoint().

The file name to be written is contained in the string checkptindfilename. It must comply with DOS naming conventions, even on Unix systems, for compatibility reasons. Thus, only an 8-character name, a period, and 3-character extension are allowed. For Onepops, this name is assembled from the user input (file or keyboard), up to 8 characters, using the field "checkptfileprefix", and appending ".new". If the user does not specify a checkptfileprefix, then the restartfileprefix is used, unless none is specified; in that case, the default "sgackp" is used. For Manypops runs, the procedure is similar, except the user-entered prefix is limited to 6 characters, and two-digit subpopulation numbers are appended to it, generating such file names as "runone00.new", "runone01.new", etc. Each subpop has its own checkpoint individual file.

APPENDIX FIVE:

CONTENTS OF THE FILE TYPES WRITTEN BY GALOPPS

I. File xxxxxx99.stt:

Runs of Manypops create or read and write a file called xxxxxx99.stt, where xxxxxx is the checkptfileprefix specified by the user for this run. The file contains the overall statistics about the run, including the contributions of ALL subpopulations being run by ALL processes in ALL processors using the master file defining this Manypops run. (Onepop runs do not need or use this file at all.)

If the user is initiating a NEW run, and not restarting from any saved checkpointed populations or with "seed" individuals found in previous runs, then the run should be started with the .stt file "zeroed out." That is done merely by REMOVING (erasing, deleting) the file from the directory in which the run is being done. When the process "running" subpopulation 0 determines that there is not such a file in existence, it will create one full of zero values, which will then be used by it and all other processes involved in solving this problem. IF THE USER FORGETS to delete this file, the program will print a warning to this effect on both the program's normal output and on stderr, however it does not interfere with normal execution. The file will automatically be zeroed out.

If the user RESTARTS a run from checkpoints written previously, the .stt file should be LEFT AS IS, so that gathering of performance statistics can continue uninterrupted. Its name will agree with the checkpoints WRITTEN by the first part of the run being restarted, so it will be used when the user specifies the restartfileprefix from which the run is to be reloaded. After the first cycle of the restart, a new .stt file for THIS run will be created, matching the names of the checkpoint files being written now.

The .stt file is read at the beginning of the first cycle and the end of each cycle of each subpopulation in each process of the problem. Its values are incremented by the results of that subpopulation's cycle. Thus, it is up to date as of the last cycle completed for each subpopulation by each processor.

WARNING: Because the updating of this file is done asynchronously by (potentially) many processors, the contents become INVALID if a restart is done after an "abnormal termination." That is, if the restart results in "throwing away" of partial cycles (which occurs when some .new and .neu files are left in the directory because a cycle was not completed), then the contributions of those partial cycles will ALREADY have been added to all_pops.stt. Thus, after such a restart, the user should RECOGNIZE that the global statistics have been affected by adding in of fitnesses, evaluations, etc., even though the populations which did them have been THROWN OUT. This does NOT occur if the Manypops processes are terminated normally at the end of a cycle, or if only a single process is being used to run all subpopulations.

```
ap0to9on.dsk    (desktop file for above)
mak0to9.man     (makefile for Manypops)
ap0to0p8.in     (Manypops (mainmany.c) run)
ap0to9ma.prj    (builds Manypops, but called ap0to9ma in Borland C++)
ap0to9ma.dsk    (desktop file for above)
```

Application File **appmatch.c** -- demo application for the hybrid representation using automix.c as the setup tool:

```
makmatch.one    (makefile for Onepop)
appmatch.in     (sample input for Onepop (apmatchs) run)
apmatchs.prj    (builds Onepop, but called apmatchs in Borland C++)
apmatchs.dsk    (desktop file for above)
makmatch.man    (makefile for Manypops)
appmatc8.in     (sample input for Manypops (apmatchp) run)
apmatchp.prj    (builds Manypops, but called apmatchp in Borland C++)
apmatchp.dsk    (desktop file for above)
```

Mixed Representation Automatic Setup Utility -- **automix.c**:

It is RECOMMENDED that all users wanting to experiment with the hybrid representation use THIS program, AUTOMIX.C, to prepare the definition of their representation, which should be coded using the template appautmx.c. There are many other tools available in mixedrep.c, most of which may be used directly by the user, but they are much more difficult to learn to use, whereas automix and the appautmx.c template are very easy.

```
automix.mak     (Unix command to source to compile automix.c)
```

Mixed Representation Tutorial Assistant **mixtutor.c**:

(A tutorial program which "teaches" you about GALOPPS's hybrid (mixed reordering/value) representation. Needed only for those NOT using automix.c, which is much easier to use.)

```
mixtutor.mak    (Unix command to source to compile mixtutor)
mixtutor.prj    (Borland C++ project file to compile mixtutor)
mixtutor.dsk    (Borland C++ desktop file used with previous)
```

Mixed Representation Setup Assistant **mixsetup.c**:

(NOTE: This utility is still needed to revise parameters for the rally application, apprally.c; however, it is recommended that those coding new applications use only the automix setup program, and begin their coding from the appautmx.c template.)

```
mixsetup.mak    (Unix command to source to compile mixsetup)
mixsetup.prj    (Borland C++ project file to compile mixtutor)
mixsetup.dsk    (Borland C++ desktop file used with previous)
```

```
app1pos8.in      (Manypops multiple-subpopulation run)
8pop2nbr.mst     (sample .mst file used by several Manypops input files)
app1psnp.prj     (builds Manypops, but called app1psnp.exe, in Borland
                      C++)
app1psnp.dsk     (desktop file for above)
```

Application File **app1both.c**: (Also run program mixtutor for setup help)

```
makap1bo.one     (makefile for Onepops)
app1both.in      (sample input for Onepop (app1bots) run)
app1bots.prj     (builds Onepop, but called app1bots.exe, in Borland C++)
app1bots.dsk     (desktop file for above)
makap1bo.man     (makefile for Manypops)
app1bot8.in      (Manypops multiple-subpopulation run)
8pop2nbr.mst     (sample .mst file used by several Manypops input files)
app1botp.prj     (builds Manypops, but called app1botp.exe, in Borland
                      C++)
app1botp.dsk     (desktop file for above)
```

Application File **apprally.c**: (Also run program mixtutor for setup help)

```
makrally.one     (makefile for Onepop)
apprally.in      (sample input for Onepop (appralls) run)
appralls.prj     (builds Onepop, but called appralls.exe, in Borland C++)
appralls.dsk     (desktop file for above)
makrally.man     (makefile for Manypops)
rally4po.in      (Manypops (mainmany.c) multiple-subpopulation run)
rally4.mst       (sample .mst file used by several Manypops input files)
rally9po.in      (Manypops multiple-subpopulation run)
9popbest.mst     (sample .mst file used by several Manypops input files)
ral9st.mst       (sample .mst file used by rally app, usable by others)
rallycon.in      (Manypops multiple-subpopulation run, RESTARTING from
                      a set of subpopulations produced by rally9po.in,
                      which MUST have been run before running this.)
apprallp.prj     (builds Manypops, but called apprallp.exe, in Borland
                      C++)
apprallp.dsk     (desktop file for above)
cre8rall.mak     (compilation command for independent main program
                      cre8rall.c, which is used to make the inter-city
                      distance table needed to run apprally.c.)
cre8rall.prj     (sample project file for Borland C++ for cre8rall.c
                      program)
cre8rall.dsk     (sample desktop file for Borland C++ for cre8rall.c
                      program)
```

Application File **app0to9.c** -- Demo application with non-binary fields

----

```
mak0to9.one      (makefile for Onepop)
ap0to9on.in      (sample input for Onepop (ap0to9on) run)
ap0to9on.prj     (builds Onepop, but called ap0to9on in Borland C++)
```

```
makmansq.man     (makefile for Manypops)
man24to8.in      (sample input file for 32-population run, 10 jobs)
24into8.mst      (sample master file for 32-population run)
apmansqp.prj     (project file for Borland C++, Manypops)
apmansqp.dsk     (desktop file for Borland C++, Manypops)
10sub11p.mix     (sample subdivision file for 10-job run, 22 extra
                      fields, MUST BE MADE BY USER by running mixtutor.c
                      (see description in manual).  Input values for
                      generating THIS example are described in manual,
                      under discussion of appmansq.c)
10jobsko.par     (sample parameter file for manuf. sequencing problem,
                      written by makmanko.c or makmanuf.c, with 10 jobs.
                      THIS (or similar) FILE MUST BE MADE BY THE USER,
                      by executing makmanko or makmanuf.  Parameters to
                      be used for this sample run are described under
                      appmansq in the manual.)
makmanuf.prj     (prj file for Borland C++, builds a standalone program
                      which can write random example manufacturing
                      problems for solution by appmansq.
makmanuf.dsk     (desktop file for BorlandC++, Manypops)
makmanuf.c       (standalone program for making parameter file for jobs)
makmanuf.mak     (compiler command for compiling makmanuf.c)
makmanko.c       (standalone program for making parameter file for jobs,
                      with a known optimum solution.)
makmanko.prj     (prj file for Borland C++, builds a standalone program
                      which can write random example manufacturing
                      problems for solution by appmansq.
makmanko.dsk     (desktop file for BorlandC++, Manypops)
makmanko.mak     (compiler
```

Application File **app1perm.c**: (Also run program mixtutor for setup help)

```
----------------------------
makap1pe.one     (makefile for Onepop)
app1perm.in      (sample input for Onepop (app1prms) run)
app1prms.prj     (builds Onepop, but called app1prms.exe, in Borland C++)
app1prms.dsk     (desktop file for above)
makap1pe.man     (makefile for Manypops)
app1prm8.in      (Manypops 8-subpopulation run)
8pop2nbr.mst     (sample .mst file used by several Manypops input files)
appprm1p.prj     (builds Manypops, but called app1prmp.exe, in Borland
                      C++)
appprm1p.dsk     (desktop file for above)
```

Application File **app1posn.c**: (Also run program mixtutor for setup help)

```
----------------------------
makap1po.one     (makefile for Onepop)
app1posn.in      (sample input for Onepop (app1psns) run)
app1psns.prj     (builds Onepop, but called app1psns.exe, in Borland C++)
app1psns.dsk     (desktop file for above)
makap1po.man     (makefile for Manypops)
```

```
                    for a truly parallel run on 4 processors (or faked
                    with 4 processes on one processor,in Unix)
approyr8.in2    (see above)
approyr8.in3    (see above)
approyr8.in4    (see above)
rr24to8.in      (sample input for Manypops (approyrp) multi-subpop run)
approyrp.prj     (for Manypops ("p" is for "parallel")
approyrp.dsk      "     "
```

Application File **appbtsp.c**:

```
----------------------------
makbtsp.one     (makefile for Onepop)
appbtsp.in      (sample input for Onepop (appbtsp) run)
makbtsp.man     (makefile for Manypops)
appbtsp8.in     (Manypops (mainmany.c) run)
8pop2nbr.mst    (sample .mst file used by several Manypops input files)
cre8btsp.c      (program to make intercity distance input file.  To run
                    the sample input files, you should run cre8btsp (or
                    makecity, as executable is called under Unix) to
                    create files 10cities.dst (for appbtsp.in) or
                    20cities.dst (for appbtsp8.in).)
cre8btsp.mak    (sample compilation command for making the standalone
                    program which writes the table of distances
                    between n randomly selected cities, to use with
                    appbtsp.c.)
cre8btsp.prj    (sample .prj file for making cre8btsp.c, for creating
                    city distances table in Borland C++, appbtsp.c
                    only.)
cre8btsp.dsk    (sample .dsk file for making cre8btsp.c, for creating
                    city distances table in Borland C++, for appbtsp.c
                    only.)
```

Application File **appmansq.c**: (new version based on automix is forthcoming soon)

```
--------------------------------------------------------------
makmansq.one    (makefile for Onepop (mainone.c))
appmansq.in     (input file for single-population run, 5jobs)
apmansqs.prj    (prj file for Borland C++, Onepop)
apmansqs.dsk    (dsk file for Borland C++, Onepop)
5sub6prm.mix    (sample subdivision file for 5-job run, 19 extra
                    fields, MUST BE MADE BY USER by running mixtutor.c
                    (see description in manual).  Input values for
                    generating THIS example are described in manual,
                    under discussion of appmansq.c)
5jobsko.par     (sample parameter file for manuf. sequencing problem,
                    written by makmanko.c or makmanuf.c, with 5 jobs.
                    THIS (or similar) FILE MUST BE MADE BY THE USER,
                    by executing makmanko or makmanuf.  Parameters to
                    be used for this sample run are described under
                    appmansq in the manual.)
```

```
Application File app.c:  (From Goldberg's book)
-----------------------------------------------
makapp.one       (sample makefile for app.c for Onepop (i.e., mainone.c,
                      etc.on Unix, becomes Onepop; on DOS, appone.exe)
appone.in        (sample input file for app.c for Onepop (i.e.,
                      mainone.c))
appone.prj       (sample .prj file for app.c, Onepop, for Borland C++)
appone.dsk       (sample desktop file for app.c for Borland C++)
2runsapp.in      (sample input file for Onepop, 2 runs (diff. params))
makapp.man       (sample makefile for app.c, Manypops(file mainmany.c))
apppop4.in       (sample input file, app.c for Manypops (4
                      subpopulations))
4popbest.mst     (sample .mst file used by several Manypops input files)
appmany.prj      (sample .prj file for app.c for Manypops for Borland
                      C++)
appmany.prj      (sample .prj file for app.c, Manypops, for Borland C++)

Application File app1.c:  (From Goldberg's book)
------------------------------------------------
makapp1.one
app1one.in      (sample input for Onepop (app1one) run)
app1one.prj
app1one.dsk
makapp1.man
app1pop4.in     (sample input file for Manypop(app1many) multi-subpop run)
4popbest.mst    (sample .mst file used by several Manypops input files)
app1many.prj
app1many.dsk

Application File app2.c:  (From Goldberg's book)
------------------------------------------------
makapp2.one
app2one.in       (sample input for Onepop (app2one) run)
app2one.prj
app2one.dsk
makapp2.man
app2pop8.in      (sample input for Manypops (app2many) multi-subpop run)
8pop2nbr.mst     (sample .mst file used by several Manypops input files)
app2many.prj
app2many.dsk

Application File approyrd.c: (J. Holland's Royal Road Challenge Problem)
----------------------------
approyrd.in       (sample input for Onepop (approyrd) run)
approyrd.prj      (for Onepop)
approyrd.dsk      (for Onepop)
approyr8.in       (sample input for Manypops (approyrp) multi-subpop run,
                      but all running from one process)
approyr8.in1     (one of FOUR sample input files for Manypops (approyrp)
```

APPENDIX FOUR -- AUXILIARY FILES

List of the Auxiliary Files Provided with the GALOPP System, Release 2.35, and with What Problem Files They Are Associated

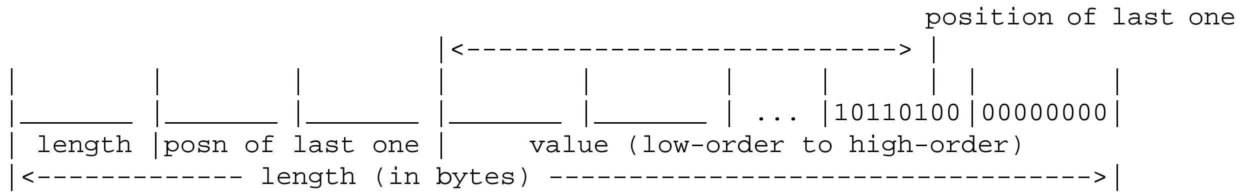
Files below are listed according to the applications with which they are used. Note that some .mst files are listed several times, as they are used with more than one application. Not included below are the .c and .h files, which are described in the section entitled "Modules To Compile" above.

NOTICE: the parameter values, number of subpopulations, length of runs, choice of neighbors and how many individuals to bring in, which selection and genetic operators to use, etc., are all chosen here ONLY FOR DEMO PURPOSES, to show the various features of the GALOPP System, **NOT as examples of good choices of values, operators, etc.** GALOPPS does NOT automatically choose good parameter settings for you by default, and it is expected that users have familiarity with genetic algorithms AT LEAST to the level of reading the first few chapters of Goldberg's book, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Examples were chosen to demonstrate all of the features of GALOPPS, not necessarily to solve particular problems with maximum efficiency. Please bear that in mind when running these examples or your own problems.

```
Information For Users:
----------------------
READMEFI.RST     (ASCII text file describing this version of the GALOPP
                       System)

nowarran         (ASCII text file disclaimer, indicating that there is NO
                       WARRANTY of any kind associated with the GALOPP
                       System -- user uses at own risk)

guide235.ps      (User's Guide for GALOPP System, Release 2.35,
                       POSTSCRIPT VERSION)

guide235.txt     (User's Guide for GALOPP System, Release 2.35, ASCII
                       text version)


Templates Defining the Structure of Input Files:
------------------------------------------------
(NOTE:  The easy way to develop an input file for YOUR application is
just to pick an example of one and then modify it until it runs.  As long
as you use the (optional) keyword on each input line, then at each stage,
the program will TELL you what it has was looking for and what it has
found, so it is very easy to develop a file with all inputs present in
the correct order.)
intempla.one     (template defining the optional input file for Onepop)
intempla.man     (template defining the optional input file for Manypops)
```

described briefly below:

A BIGNUM begins with a one-byte field providing the entire length of the BIGNUM. Next comes a two-byte field providing the position of (from the lowest-order bit) of the last ONE bit in the value field (i.e., all higher-order bits are 0). After that field comes the value field, a series of n bytes which, if concatenated, would represent the binary value of the BIGNUM. That is:



(If the user should need to represent numbers exceeding 2**4096, it is obvious how the format can be extended.)

```
   ...
   }
```

In some special cases, we can use '=' as a shortcut to do assignment quickly, but we must be very careful not to discard memory we previously malloced --for example:

```
   {
   BIGNUM tmp,bresult ;
   ...
   for   (i=0;i<1000;i++)
{
tmp = addbyI(bresult,i) ;  /* Now tmp is a temporary result.     */
     delete(bresult) ; /* bresult is going to be assigned a new value */ bresult
= tmp   ; /* Here must be sure that we don't use tmp's   */                      /
* current value any more. */
}
   ...
   }
```

(3) Every function is responsible for initializing/deleting its local BIGNUMs in its own range. In other words, every function can delete all BIGNUMs which IT initialized or was passed as a function return from a 'called function'. But BIGNUMs passed into a function as arguments should be preserved by the called function; i.e., the called function should make its own "local" copy of the BIGNUM passed to it, and then it is free to use it as it will (without altering the BIGNUM that was passed to the fu
nction. It is not permissible for the called function to alter the BIGNUM passed to it, because operations on it might require alteration of its length (and thus of the memory allocated to it), and THAT IS NOT PERMITTED. A BIGNUM MAY ONLY BE CREATED, COPIED, USED (I.E., ITS VALUE USED AS AN OPERAND), AND DELETED, BUT NEVER ALTERED. INSTEAD OF ALTERING A BIGNUM, ONE MUST CREATE A NEW ONE WITH THE DESIRED VALUE (RESULT OF AN ARITHMETIC OPERATION, FOR EXAMPLE), AND THEN, IF THE VARIABLE NAME MUST REMAIN THE SAME, ONE MUST DELETE THE OLD VALUE FOR THE VARIABLE AND CREATE A NEW ONE WITH THE SAME NAME USING THE NEW VALUE AND FUNCTION newfromB().
In some ways, this resembles using variables in the stack instead of using the original ones in ordinary C routines. In the example below, a passed parameter is copied into a new variable, bpar, which then may be used and deleted within function fun.

```
   fun(BIGNUM barg)
   {
   ...
   BIGNUM bpar   ;           /* Copy of BIGNUM argument.*/

 bpar = newfromB(barg) ; /* barg never occurs in fun() except here. */  ...
/* Now we can use bpar freely. */
   }
```

In spite of all the rules listed above, BIGNUM might be easier to use than you expect. If you like, please have a look at "bignum.c" for detailed information about BIGNUM.

To assist you in understanding these routines, the format of a BIGNUM is

```
...
delete(btmp) ;
...
}
```

The functions we can use to initialize/assign the value of BIGNUM are the following:

new,newfromB,newfromL,newfromI,newfromS,
 -or-
any function that returns a BIGNUM (addbyX, minusbyX, multbyX, dividebyX, or a user-defined function).

It is obvious that we must delete/free the local BIGNUM variable whenever we won't use it any more, because its space is obtained from malloc or realloc, and it will reside in memory forever unless the program terminates or the space is freed by a delete (which calls free). The pointer we use to hold the address of the beginning of a BIGNUM (see also file bignum.h) is a local and automatic variable, and if we return to an outer procedure(caller) without freeing it, we can no longer access it, and it is lost for ever! (... well, at least until the program terminates). If a BIGNUM variable has already been assigned a value, it is also important to delete it whenever another assignment is made, for example:

```
fun()
{
BIGNUM bthis  ;
...
bthis = newfromL(0L)  ;
...
delete(bthis) ;
bthis = newfromI(32767);
...
}
```

(2) It is usually invalid to use '=' to assign a BIGNUM to another BIGNUM, for example:

```
{
BIGNUM bthis,bthat    ;
...
bthis = bthat ; /* No good! bthis and bthat point to the same memory
 * location, and further operations on one
 * variable name may invalidate the other
 * while it is still in use.  */
...
}
```

Instead, we use newfromB to accomplish this -- for example:

```
{
BIGNUM bthis,bthat    ;
...
delete(bthis) ;  /* if bthis has already been assigned a value. */
 bthis = newfromB(bthat)       ; /* Valid! Now bthis and bthat have the
                                  * same VALUE, but are independent of
                                  * each other. */
```

APPENDIX THREE

INTRODUCTION TO THE BIGNUM LIBRARY FOR EXTENDED-RANGE POSITIVE INTEGER ARITHMETIC

(C) Copyright Michigan State University, 1994.
Designed and Implemented by Erik Goodman (MSU) and Wang Gang (Beijing University of Aeronautics and Astronautics), January, 1994.

The BIGNUM LIBRARY was designed and developed in response to a need for calculating combinations of n objects m at a time (C(n,m)) for arbitrarily large n and m, and for calculating subindices (integer subranges) of an index with range [0, C(n,m)-1] such that the product of subrange cardinalities is the cardinality of the range. The need arises from seeking to decode the positions of a class of fields among a larger set of fields, during research on genetic algorithms. Using this representation, it is possible to use the positions of "extra" fields on a chromosome to encode arbitrary parameters for search by the genetic algorithm, using reordering-type genetic operators.

BIGNUM may be visualized as an ordinary data type, similar to int or unsigned long, except that there are several additional rules governing its usage, as described below. Functions called to do arithmetic operations involving BIGNUMS are named for the operation and the data type of the other operand (i.e., addbyB adds two BIGNUMs, addbyL adds a BIGNUM and a Long int, etc.). We use the shorthand "addbyX" to refer to all of the addition functions. Most functions are defined for four data types: BIGNUM (B), long int (L), int (I), and unsigned short (S).

In case of int or long, the variable's value can be automatically stored in registers by the compiler when operations are being performed involving it, because each of them needs only two or four bytes of data to represent it. Every variable has a fixed location to store its value.

But a BIGNUM is different because we don't know the exact amount of space needed to store it until we have determined its value. In fact, we use malloc and realloc to get the space we need to store it. We must use explicit statements to allocate/initialize or delete the space each BIGNUM might need. That is why the usage of the BIGNUM type is a little more complex than usage of a variable of an ordinary data type. There are several rules we must obey in order to use BIGNUM variables properly:

(1) Usually we use newfromX and delete() to allocate storage for and initialize, and later delete, the BIGNUM we declare at the beginning of the function, for example:

```
fun()
{
BIGNUM btmp  ;
...
btmp = newfromI(256) ;
```

APPENDIX TWO -- PERMUTATION INDEXING

Permutations and Permutation Indices: Encoding and Decoding Methods

It is relatively easy to index the set of n! permutations of n objects. The encoding and decoding algorithms developed for the GALOPP System both involve an intermediate representation which we call a relative index. To decode an array containing a permutation P of {0, 1, ..., n-1}, we first calculate the relative index r(i) of each successive element P(i) of the permutation among the ordered set Intn = [0, 1, ..., n-1]. That is,

```
        initialize Iwork to Intn;
        for (i=0;i<n;i++) {
             for (j=0; j<Iwork(i); j++) {
                  r(i) = the number of elements in Iwork BEFORE P(i)
                  Iwork <- Iwork\P(i)
             }
        }
```

The removal of each value P(i) from the array Iwork is accomplished in the computer code by setting that element of Iwork to a negative number, and then NOT counting negative values when determining r(i). Thus, for each field P(i), we now have a corresponding rel_index(i). Then the index of permutation P is simply the sum of r(i) * (n-i-1)! , for i=0, 1, ..., n-2.

This process is easily reversed to calculate a permutation (element by element) from a given permutation index.

As n grows, the maximum index will quickly exceed the maximum number which can be stored in an int or long int word. While it would have been possible to treat the permutation code overflow problem in the same manner as the position codes are handled (i.e., by using extended range numbers), it is possible to subdivide the permutation index more directly, simply, and quickly:

Since n!, the number of permutations of n objects, is easily factored after its kth term, into (n!/k!) and k!, we can similarly divide the permutation index into two subfields or subindices. The first subfield indexes the possible arrangements of the first (n-k) objects, and the second, the remaining k objects. Of course, this process can be repeated, up to the limit of creating n-1 subindices. As many or as few subindices as desired can be used, so long as care is taken that none of the subindices can overflow the data type to be used. If we break n! after its factors b(j), (j=1, 2, ..., m (<n)), and consider b(0) to be n and b(m) to be 0, then subfield j contains (b(j-1))! / (b(j))! indices, and so long as each is less than or equal to the maximum value for the data type, for each j, that choice of permutation index decomposition is valid.

calculating. Let nowstart stand for the column in which we start adding M[r,p] to the result (for p from (nowstart) to (nowstart+B[r]-1)).

First initialize (result) to 0, r=0, nowstart=0 and loop r from row 0 to row 4. In each row r, for p from (nowstart) to (nowstart + B[r] - 1), add M[r,p] to result, then set nowstart to (nowstart + B[r]). Then increment r to the next row.

In this example,

```
    p=0          p=1        p=1             p=3        p=3
   B[0]=1      B[1]=0     B[2]=2         B[3]=0     B[4]=0    result += 35(row
1) + 0(row 2) + (6+3)(row 3) + 0(row 4) + 0(row 5).
```

At the end, the index of instance A is 44.

For the case A = [0 0 0 1 1 1 1 1], B is [3,0,0,0,0], and the index of A is ((35+15+5)+0+0+0+0) = 55.

For the case A = [1 1 1 1 1 0 0 0], B is [0,0,0,0,0], and the index of A is (0+0+0+0+0+0) = 0.

For more detail on the implementation, you will need to refer to the code.

look at the matrix M in another way: view it as a directed graph whose every vertex has a directed edge to its 'down child' and one to its 'right child'. Every entry i,j in M, for i = 0,..., m-1, is C(i+j,i). All maximal length paths from any vertex end at the terminal vertex, M[0,0].

Then each path from a starting vertex M[i,j] (= C(i+j,i)) to the lower right-hand corner represents one possible choice in C(i+j,i). That is intuitively clear because C(i+j,i) appears i rous up from the lower right-hand corner, and j columns to its left. The length of all paths to the corner from C(i+j,i) is thus a total of i+j edges, and among them, exactly i must be down edges, which is clearly the problem: (i+j edges, choose i down) or C(i+j,i) paths.

Since these paths are 1:1 onto the set of choices in C(n,m), they can be used as the basis for an index of C(n,m).

The value at each node in the digraph is the number of paths from THAT node to the terminal vertex. For each path to the terminal vertex in this graph, there is a corresponding sequence number when the paths are enumerated by a 'depth first' algorithm. We use that number as the index of that instance of C(n,m).

We could use a recursive procedure to calculate each index (or instance). But that would be very expensive. Using M(n,m) and the B array of "skip codes" developed above, we can calculate the index of that instance quickly, traversing only a single path through the graph under the guidance of the skip code.

USING THE SKIP CODES AND MATRIX M

For example, let n=8 and m=5. (In the GALOPPS system, that means we have 8 total fields, of which 5 are "extra" fields.) We must have already calculated the M up to a dimension of at least 5 rows and 4 columns, as shown, using Equation 2. Of course, we can fill in as many additional entries as we desire, because entries already computed do not change as n and m increase. However, we have written function GetMatrix so that it returns a pointer to a matrix especially set up for the particular (n,m) to be used for the problem, since it allows easier (and faster) subscript calculations.

For this example of calculating indices based on C(8,5), M includes:

```
35  15  5   1
20  10  4   1
10  6   3   1
4   3   2   1
1   1   1   1
```

Let the binary array A for which we want to find the index be:

A array = [0 1 1 0 0 1 1 1], and calculate its skip code (B array) as:
B = [1 0 2 0 0]

We will now revert to the "standard" notation for entries in matrix M, such that M[0,0] is in the upper left corner, as it simplifies this description greatly. The index will be a sum of entries selected from some columns c in each row r, according to the skip code, B[r]. Let (r) denote the row whose entries we are currently considering for possible addition to the index value ("result") we are

preceding equation indicates that C(n,m) is simply the sum of all entries in the row immediately below it which are directly below or to the right of C(n,m).)

```
  C(n,m)__________________________________________________________
   _  |     |             |                              |             |
 /|\ |  C(n-1,m-1)   C(n-2,m-1)  ...       ...    C(m-2,m-1)   C(m-1,m-1)
  |  |  C(n-2,m-2)   C(n-3,m-2)  ...       ...    C(m-3,m-2)   C(m-2,m-2)
  |  |     .             .                             .            .
(m-1)|     .             .                             .            .
  |  |     .             .                             .            .
  |  |  C(n-m+2,2)   C(n-m+1,2)  ...       ...    C(3   ,  2)  C(2   ,  2)
 \|/ |  C(n-m+1,1)   C(n-m,1)    ...       ...    C(2   ,  1)  C(1   ,  1)
   _    ----------   ----------  ---       ---    -----------  ----------
           |<---------------------- (n-m+1) ----------------------->|
```

Notice that each element C(p,q) in matrix M is the sum of (C(p-i,q-1), for i from 1 to p-q+1), just as we saw was true for the problem C(n,m). (Fortunately, there is an even easier way to compute the entries -- see below.)

We can extend the matrix and make the last row satisfy Equation 1:

```
  C(n,m) ________________________________________________________
            |             |                              |             |
/|\ |  C(n-1,m-1)   C(n-2,m-1)  ...       ...    C(m-2,m-1)   C(m-1,m-1)
 |  |  C(n-2,m-2)   C(n-3,m-2)  ...       ...    C(m-3,m-2)   C(m-2,m-2)
 |  |     .             .     .           .
 m  |     .             .       ...C(i+j,i)...        .            .       ^
 |  |     .             .                           .            .       |
 |  |  C(n-m+2,2)   C(n-m+1,2)  ...       ...    C(3,  2)     C(2,  2)    |
 |  |  C(n-m+1,1)   C(n-m,  1)  ...       ...    C(2,  1)     C(1,  1)    |
\|/ |      1             1      ...       ...        1            1     i=0
       ----------   ----------  ---       ---    -----------  ---------
          |<----------------------(n-m+1)------------------------>|
                                                         <-------- j=0
```

Now index matrix M with i and j starting from the lower right-hand corner and increasing upwards and to the left. Then element M[i,j] contains C(i+j, j) (which, of course, also equals C(i+j, i)). It is easily shown that

    M[i,j] = M[i-1,j] + M[i, j-1],   (Equation 2)

which is equivalent to the identity:

    C(p,q) = C(p-1, q) + C(p-1, q-1).

So matrix M can be filled by initializing the bottom row and rightmost column to 1, and then simply applying Equation 2 inside row and column loops. If Matrix is square with m rows and columns from 0 to m-1, then it can be used to calculate any C(n,m) for n up to and including 2m, and various economies are possible, taking advantage of the symmetry of the matrix, and of the equality of C(i+j, i) and C(i+j,j), if they do not slow down the execution of the algorithm.

Now let the dimension of M be m rows [0, m-1] and n-m+1 columns [0, n-m], and

COMBINATIONS MATRIX

In order to calculate a combination index for any given m and n, we need to have an intermediate Matrix for that tuple (n,m). We use function GetMatrix to fetch the matrix needed for this particular number of total fields and extra fields. It checks to see if the matrix for that tuple (n,m) has already been calculated (during this run of the program). If not, it creates it, and returns a pointer to it; if it already exists, it simply returns a pointer to it. The overall effect is that the matrix is calculated only once for any (n,m) in a run of the GALOPPS program -- namely, the first time the function is called (see also discussion below).

For any tuple (n,m), M(n,m) is a matrix of size [m, n-m+1]. We initialize M with the following calculation (see explanation below for the derivation of this calculation):

```
 (1) M[m-1, j] = 1    for (j=0, ..., n-m);
     M[i, n-m] = 0    for (i=0, ..., m-2)
     M[i,j] = 0       otherwise;

 (2) for(i=m-2; i>0; i--) {
     for (j=n-m-1;j>0;j--) {
       M[i, j] = M[i, j-1] + M[i-1, j]
     }
     }
```

DERIVATION OF THE COMBINATIONS MATRIX

Let C(n,m) denote the number of distinct ways of selecting m ones out of n binary values. We can develop our algorithm for calculating an index based on C(n,m) by analyzing the number of possible choices for placing each successive one FROM LEFT TO RIGHT among the n positions (i.e., by the FIRST ONE, we mean the LEFTMOST ONE):

There are (n-m+1) choices for the first 1, beginning at position 1 and ending at position n-m+1. (Note that the first 1 COULD NOT occupy position n-m+2 or higher, because if it did, there would not be room for the other m-1 ones in the remaining m-2 positions.) Now, for each position (i) which the first one could occupy, we have defined another subproblem: C(n-i,m-1), i.e., how many possible choices remain for placing the remaining ones among the remaining positions. The answer to that subproblem is, of course, the number of possible solutions for C(n,m) GIVEN the FIRST one in position (i). The subproblem can be attacked recursively in exactly the same fashion as the original problem; however, we will use another approach in order to yield faster calculations.

The count of combinations C(n,m) is then simply the sum of all of the subproblem counts for all choices of (i), i.e.:

```
  C(n,m) = sum of(C(n-i,m-1), for i from 1 to n-m+1.)  (Equation 1)
```

We can define a matrix M using the expression above, as follows: (Note that the

APPENDIX ONE -- COMBINATION INDEXING

A Method for Indexing and Calculating Indices of Combinations(n,m)

This method was developed by Erik Goodman, Michigan State University, November, 1993, and implemented by Goodman and Wang Gang, Beijing University of Aeronautics and Astronautics, December, 1993 - January, 1994.

The use of genetic algorithms to solve "mixed-type" problems, involving simultaneous solution for the optimal permutation of a set of items, and for the optimal values of a set of real or integer variables, can be done using only reordering-type genetic operators, such as uniform order-based crossover, cycle crossover, etc. It may be accomplished by adding "extra" fields to the permutation fields, and then using the subsequent ordering of the "extra" fields among themselves, and the positions of the "extra" fields on the chromosome to create two indices: the permutation index and the "position" or "combination" index. The former index describes uniquely the order in which the "extra" fields appear on the chromosome, and the latter index describes uniquely the positions on the chromosome in which "extra" fields are located. This latter problem is simply one of numbering all combinations of n (all of the fields) objects taken m (number of "extra" fields) at a time, C(n,m). While it is extremely easy to calculate C(n,m) as n!/((m!) (n-m)!), it is somewhat more difficult to construct an algorithm for sequentially numbering each of those possible choices, at least in a manner that is reasonably fast to compute.

The routines for working with position (combination) indices appear in two very similar versions: mixedrep.c contains routines for working with indices small enough to fit into a long int, while bigcomb.c is a version able to process BIGNUM-format indices of arbitrary size. Of course, it is slower than the long int version. Bigcomb.c maintains a Matrix array similar to the ordinary one, except that the elements of Matrix are BIGNUMs instead of unsigned longs. Please refer to bignum.c (APPENDIX THREE) and the code itself for detailed information on using BIGNUM.

Regardless of whether we are using BIGNUMS or long ints, we calculate a combination index on the basis of the following coding method.

SKIP CODES

Given array A, with a 1 in each element where an "extra" field appears on the chromosome, we first calculate array B, the "skip code" array, as shown in this example:

A array: (0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1 }

B array: (2,0,1,1,0,0,0 },

where B[i] stands for the number of 0's between the ith 1 and (i-1)th 1 in the A array (an implicit "0th" one is always present at the left-hand end of the array).

point/restart files (.ckp for headers and .ind for individuals). Thus, for example, the user can make several runs with Onepop on single populations, and then can use their checkpoint files to initiate a multiple subpopulation run of Manypops. The user would have to rename the .ind checkpoint files, in that case, so that they share a common prefix, are numbered 00 ... 0n, and are suffixed .ind and .ckp for individual and header files, respectively. The .stt file for global statistics will also be invalid and must be ignored. Alternatively, the user may want to "explore" the behavior of a single subpopulation created by Manypops, and may use Onepop to do that exploration. Examining the checkpoint files created by Manypops should make it very clear how to proceed.

UPDATES AND BUG REPORTING

Please report all bugs as soon as possible to:
Erik Goodman
goodman@egr.msu.edu
Phone 1-517-355-6453   Fax: 1-517-355-7516
(Or see mailing address on cover.)
PLEASE BE SURE TO INCLUDE INFORMATION ON HOW TO REACH YOU WITH QUESTIONS OR REVISED CODE!

(From Russia, you may forward material and seek information through USKOV, Vladimir L'vovich, CAD Department (RK-6), Moscow State Bauman Technological University, Building 5, 2-nd Baumanskaya Street, Moscow, 107005, Russia. email: uskov@aicad.isrir.msk.su., phone: 095/210-07-93 (home) or 095/263-65-26 (department office). Fax (personal): 095/292-65-11 (in message header, specify USKOV, V. L. - CADDPRT 011722).

(From China, you may forward material or request information through Professor Li Wei, Department of Computer Science, Beijing University of Aeronautics and Astronautics, Beijing, 100083, China (Phone: 86-1-201-7251, ext. 614 or 918, Fax: 86-1-201-5347), or try (unreliable) gabuaa@bepc2.ihep.ac.cn, Attn: Wang Gang.

If you would like to receive updates (bug fixes and/or new releases of the system) please let the author know of your interest through any of these channels.

plus a choice of EXACTLY ONE MUTATION OPERATOR:

bitmutat.c (for non-permutation problems)
scramble.c (for permutation or "mixed" problems)
swap.c (for permutation or "mixed" problems)

plus, for SINGLE POPULATION OPERATION, BOTH OF:

mainone.c
startup.c

OR for PARALLEL SUBPOPULATION OPERATION, ALL THREE OF:

mainmany.c
initsubp.c
master.c

To compile the seven **"auxiliary"** main programs provided with the GALOPP System, the following directives (or their equivalents) may be used (the files suffixed .mak may be sourced by Unix users to do this, instead):

For **automix.c** (Assists user in defining a mixed representation for use with template appautmx.c):
cc -g automix.c mixedrep.c utility.c bignum.c bigcomb.c -lm -o automix

For **mixtutor.c** (Assists user to develop appropriate representations using permutation and position indices or subindices):
cc -g mixtutor.c mixedrep.c utility.c bignum.c bigcomb.c -lm -o mixtutor

For **mixsetup.c** (Assists user to develop appropriate representations using permutation and position indices or subindices, and writing a file of the subfields and subranges):
cc -g mixsetup.c mixedrep.c utility.c bignum.c bigcomb.c -lm -o mixsetup

For **cre8btsp.c** (Creates distance table for n randomly placed cities in a square of user-specified size, for blind traveling salesman problem):
cc -g cre8btsp.c utility.c -lm -o makecity

For **cre8rall.c** (Creates distance table for n randomly placed cities in a square of user-specified size, plus one more random start/finish point (for road rally problem):
cc -g cre8rall.c utility.c -lm -o makerall

For **makmanuf.c** (Creates a file of parameters for input to the manufacturing job sequencing application, appmansq.c):
cc -g makmanuf.c utility.c -lm -o makmanuf

For **makmanko.c** (Creates a file of parameters for input to the manufacturing job sequencing application, appmansq.c, but this problem has a known optimum solution, unlike the one made by makmanuf.c):
cc -g makmanko.c utility.c -lm -o makmanko

NOTE: (At some point (probably not now), you may want to know this:)
The GALOPPS/Onepop and Manypops programs can read and write each others' check-

Modules to Compile

If an ANSI-compatible 'C' compiler is available, the code can be compiled "as is". If your compiler will not accept ANSI-style prototype declarations, then you must edit the files "external.h" and "sga.h" and comment out the lines near the top which read "#define PROTOTYPES_ACCEPTED" and "#define SECONDARY_PROTOTYPES_ACCEPTED" from both files (see additional information in the header of file "external.h").

For Unix users, example makefiles are provided, which the user may alter to suit their particular needs.

To compile and link the files for a PC, you must select the appropriate routines to compile, including selection among the choices above, using the rules below. Some Borland C++ project files are included, but they are not necessarily configured correctly for your hardware... please use the options menu to review the settings for the compiler. You will probably want to use the "large" memory model and optimization for fastest code, but are free to use other choices.

The files to compile and link for ALL types of problems are the following:

bignum.c bigcomb.c checkhdr.c checkrd.c
checkwt.c ffscanf.c generate.c memory.c
mixedrep.c random.c report.c statisti.c
utility.c

In the same directory must be the #include files:

sga.h (#included in mainone.c or mainmany.c)
external.h (#included in most other .c files)
sgafunc.h (#included in sga.h and external.h)
sgapure.h (#included in sga.h and external.h)

One must also choose EXACTLY ONE APPLICATION FILE:

app.c app1.c apphybxx.c
app2.c approyrd.c app0to9.c
appbtsp.c app1perm.c appmatch.c
app1posn.c apprally.c appautmx.c
app1both.c appmansq.c
appxxxxx.c (your problem, made by modifying the app form supplied).

plus a choice of EXACTLY ONE SELECTION METHOD FROM:

rselect.c srselect.c rnkslect.c
suselect.c tselect.c

plus a choice of EXACTLY ONE CROSSOVER OPERATOR:

oneptx.c (for non-permutation problems)
twoptx.c (for non-permutation problems)
unifx.c (for non-permutation problems)
uobx.c (for permutation or "mixed" problems)
ox.c (for permutation or "mixed" problems)
cx.c (for permutation or "mixed" problems)
pmx.c (for permutation or "mixed" problems)

future releases of GALOPPS.

2) add new operators and/or selection methods without changing the structure of the other routines which call them. In that case, the author can still assist with bugs in the original system, if they occur with the original operators/ selection methods. If you add useful routines, the author would be happy to receive them for possible inclusion in future releases of the system. Upgrade to new releases of GALOPPS should still be very easy.

3) freely modify the content and structure of any of the routines in the system. This provides the greatest freedom, but the author will no longer attempt to assist in fixing any bugs discovered in such a modified system, unless you also demonstrate them in the unmodified code. However, the author will be pleased to learn of and/or receive copies of any such enhancements which are found to be useful.

Compiling/Linking the System

The systems have been run on many Unix, DOS, Windows, and Macintosh systems. A makefile is provided for each program (mak?????.one and mak?????.man for single and multiple subpopulation systems, respectively). If your system cannot use these makefiles directly, they may still be used as documentation of the modules required for compiling and linking of each system. A few additional makefile examples are included for compiling particular application problems.

For Borland C++ users on PC's, sample project files (.prj) have been included for all applications. You should first copy all files from the distribution diskette to a directory GALOPPS2.35 on your hard disk. USE CARE to be sure that the compiler options are set appropriately for your hardware configuration before you use these files to compile the system. You will want as much RAM as possible available to the GA, if you want the good performance provided by larger population sizes. You might want to set optimization for speed if your compiler allows that.

To change the problem being solved, the file appxxxxx.c or or appautmx.c or apphybxx.c (or one of the other app files) must be edited to create the new problem file. Other files need not be changed. It is suggested that the user create a copy of appxxxxx.c, called appmine.c (for example), and then edit the makefile to set the APPCODE and APPOBJ to the appropriate filenames (appmine.c and appmine.o, for example).

Just as in the original SGA-C, to change the method of selection (among roulette wheel, stochastic remainder selection, stochastic uniform sampling, tournament selection, and rank-based selection), just uncomment the appropriate set of two lines in the makefile (Unix systems) or include the appropriate filename (DOS systems). File rselect.c is roulette wheel selection; srselect.c is stochastic remainder selection, suselect.c is stochastic uniform sampling (usually recommended over the first two), tselect.c is tournament selection, and rnkslect.c is rank-based selection. Similarly, uncommenting the appropriate pair of lines chooses among the various crossover operators (oneptx.c, twoptx.c, unifx.c, uobx.c, pmx.c, cx.c, and ox.c) and the various mutation operators (bitmutat.c, swap.c, and scramble.c).

CODE DISTRIBUTION FORMAT

GALOPPS/Onepop and /Manypops share nearly all files -- the exceptions are the main programs (each has its own, labeled mainone.c and mainmany.c, respectively) and initialization routines (startup.c for Onepop and initsubp.c for Manypops), plus file master.c only for Manypops. The two programs use exactly the same application files (app.c, app1.c, app2.c, approyrd.c, appbtsp.c, apprally.c, app1perm.c, app1posn.c, app1both.c, appxxxxx.c, etc.). They use identical formats for checkpoint and restart files.

When editing any of the files distributed with the GALOPP System, your TAB character should be set equivalent to 8 SPACES, to match the setting used when the files were created. Otherwise, your listing will appear to be strangely indented.

The 'C' code runs unmodified on Unix systems (tested on Sun4, HP 9000/7XX, DECstation, and NeXT systems), on PC's (tested under MSDOS 5.0 and 6.0 and under Windows 3.1, with Borland C++, using only the 'C' features, and using Microsoft 'C'), and on Macintosh systems. A compile-time choice defining PROTOTYPES_ACCEPTED handles the compiler differences encountered so far (ANSI-type or the older K-R-type ’C’). The system was made so it can be compiled by ANSI-compliant 'C' compilers with modern prototype declarations, or without the ANSI variations. We have provided two forms of declaration for functions, in files sgafunc.h and sgapure.h. If you leave #define PROTOTYPES_ACCEPTED at the top of files sga.h and external.h, they will do full ANSI-style prototype checking. If your compiler objects to these prototypes, comment out the line #PROTOTYPES-ACCEPTED at the top of the sga.h and external.h files. In that case, use extra caution that the types of all arguments in any functions you create for calling from the appxxxxx.c routines match their declarations, for less checking is done. NOTE: On many systems, some "warning" messages will be generated during compilation, as the "skeleton" callback structure causes many variables to be declared which are not used, etc. This should NOT be a cause for alarm when GALOPPS compilations are done.

For Unix systems, makefiles are included, which can easily be modified to create whatever configuration of modules (operators, selection methods, etc.) is desired. On DOS systems, you may use the .prj files included for Borland C/C++ compilers, or may simply follow the directions given below to compile and link the necessary modules with whatever tools you are familiar.

How to Prepare the GALOPP System for Solving YOUR Problem

The author ENCOURAGES users to use this system as a basis for development of their own genetic algorithm applications and enhancements. Information about successful or unsuccessful attempts to use/modify the system would be welcomed by the author.

The system may be used in three ways:

1) write all of the code needed to run your GA application within one of the appxxxxx.c or appautmx.c or apphybxx.c files. In that case, the author will be happy to try to address bugs you may discover in the GALOPPS code, and your application should need only minor modifications to be able to be run under

```
/* from the checkpoint header file any fields added to the checkpoint   */
/* header by the user.                                                  */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


BOOL
app_malloc_utility_field(utilityfield)
/* Needed whenever user adds utility fields to the chromosome.    */
/* Otherwise, may be blank.                                       */

int    **utilityfield;

/* Application-dependent callback routine user may use for mallocing    */
/* space for the utilityfield (if used).                                */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


void
app_copy_utility(dest, source)
int    *dest;
int    *source;

/* Application-dependent callback routine user may use for copying any   */
/* utility fields used from one place to another.  Program cannot know   */
/* how you structured utility fields, so you provide the code here to    */
/* allow the program to copy them as needed.                             */
{
    /* See approyrd.c for an example. */
}


BOOL
app_read_best_utility(fp)
FILE   *fp;

/* Application-dependent callback routine user may use to read the        */
/* best utility field from the checkpoint header file.                    */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


BOOL
app_write_best_utility(fp)
FILE   *fp;

/* Application-dependent callback routine user may use to write the       */
/* best utility field to the checkpoint header file.                      */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


int
GetUtilitySize()
/* Application-dependent callback routine user must use to tell program   */
/* the size of the utility field user mallocs for each chromosome.  Used  */
/* to compute the amount of space needed per individual in the checkpoint */
/* files.                                                                 */
{
    /* See approyrd.c for an example. */
    return 0;
}
```

```
void
app_generate()
/* Complete application-specific actions NEEDED as part of making a new     */
/* generation (typically involving UTILITY fields the user has added).      */
/* Examples include any transfer of parts of structures 'individual' or     */
/* bestever which were added for this application.  This routine is a place */
/* where, for example, copy operations may be done from parents to children */
/* for ALL kids which were NOT evaluated this generation (i.e., no crossover*/
/* or mutation was done to create them, so objfunc was not called.)  So */
/* the fields usually filled in by obj_func when a NEW individual is */
/* generated can be copied from the parent here, instead.  */

{
    /* See approyrd.c for an example. */
}


void
app_stats(pop)
/* Application-dependent statistics calculations called by statistics() at  */
/* the end of each generation.                                              */
/* The call follows each generation, so can be used to update any UTILITY */
/* values for the bestever structs, by copying them from the individuals  */
/* which still contain them, for recording and reporting.                */
struct individual *pop;
{
    /* See approyrd.c for an example. */
}


BOOL
app_write_utility(UtilityBuf, fp)
int    *UtilityBuf;
FILE   *fp;

/* Application-dependent callback routine user may use for writing the  */
/* contents of the utility field (if needed) into a checkpoint file.    */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


BOOL
app_write_ckp_hdr(fp)
FILE   *fp;

/* Application-dependent callback routine user may use for writing      */
/* any needed app-specific variables into a checkpoint file.            */
{
    /* See approyrd.c for an example. */
    return TRUE;
}
BOOL
app_read_utility(UtilityBuf, fp)
int    *UtilityBuf;
FILE   *fp;

/* Application-dependent callback routine user MUST provide for reading the  */
/* contents of the utility field from a checkpoint file, if utility fields */
/* were written there by app_write_utility().                */
{
    /* See approyrd.c for an example. */
    return TRUE;
}


BOOL
app_read_ckp_hdr(fp)
FILE   *fp;
/* Application-dependent callback routine user may use for reading back */
```

```
/* desired actions in routine app_when_converged() below.  */
int * flag;
int * num_to_replace_if_converged;
{
}


void
app_when_converged()
/* This routine is called if user's routine (above) app_decide_if_converged  */
/* ever sets its flag to 2.  The intent is that user can here stop a run, or */
/* change any GA parameters as desired, for the current subpopulation, etc.  */
{
}


void
app_quiet_report()
 /* Routine to be called when other output is suppressed, to check for
  * trigger for special user-defined output, or to print desired output even
  * when running in quiet mode.  Helpful when doing many long runs.   The
  * parameters passed are to enable checking for last generation of a
  * Manypops run, for example, in case want to reset quiet to 0 for a final
  * summary printout, etc.                                                  */

{
    /* Below, supply any logic and printing to be used in quiet mode
     * operation, when most other normal output is suppressed. */

    /* for example, print warning ONE TIME if quiet flag is on, so user knows
     * why doesn't get any output.                                          */
    if(quiet < 3 && popno == 0 && cycle == 0 && gen == startgen)
        fprintf(outfp, "\n Flag quiet > 0, normal output suppressed.\n");

    /* Sample logic below starts full printing at end of next-to-last      */
     * generation of last cycle, for final summary.  (Works for GALOPPS/   */
     * Onepop (ncycles == 0) and GALOPPS/Manypops)                         */
    /* To enable this logic, remove comment delimiters on next line.       */
    /* if((cycle == ncycles - 1 || ncycles == 0) && gen == maxgen - 2)
    quiet = 0;  */
}


void
app_new_global_best_report()
/* Application-dependent report, called by globalstats() in statisti.c when */
/* GALOPPS/Manypops is run (not used by GALOPPS/Onepop runs). */
{
    /* User can print from utility fields, call output routines, etc.        */
    /* to print whatever information is wanted when a new best individual of */
    /* all populations on all processes (processors) is found.  Since the    */
    /* performance measures, etc., have NOT yet been updated in the .stt     */
    /* file, user should now output local_bestfit.xxxxx, etc., not yet */
    /* all_pops_bestfit.xxxx, which will be updated AFTER this print is done.*/
    /* Use of this callback is OPTIONAL. */

}


void
app_print_strings()
/* Application-dependent string printer, called by report() just after it has */
/* finished printing the chromosomes in "standard" binary format.             */
{
}


void
app_conv_rept()
/* Application-dependent report, called by reptconv() (in report.c) */
{
}
```

```
void
app_initreport()
/* Application-dependent initial report called once by initialize() */
{
}


void
app_malloc()
/* application dependent malloc() calls, called once by initmalloc() */
/* If use utility fields, must malloc here for all newpop, oldpop.   */
{
}


void
app_free()
/* application dependent free() calls, called by freeall().  Be sure you are */
/* really done with things before you free them.                             */
{
}


void
app_malloc_bestever_utility()
/* If you use utility fields (i.e., additional variables for each individual */
/* you must malloc them for each chromosome in app_malloc(),                 */
/* and you must ALSO malloc the utility fields for variables declared as     */
/* struct bestever, below in this routine.                                   */
/* Note:  they are never freed, so should be done only once, using           */
/* static flag "bestever_utilities_mallocd" to test.                         */
{
/*  static BOOL bestever_utilities_mallocd = FALSE;                          */
    /* Malloc below done only first time, since its memory is never freed    */
    /* except when process terminates.                                       */
/* **** DUMMY TEMPLATE GIVEN BELOW, COMMENTED OUT -- NOT USED FOR THIS APP   */
/*  if(!bestever_utilities_mallocd) {
        bestever_utilities_mallocd = TRUE;
        if ((bestfit.utility =
                (int *) malloc(sizeof(struct WHATEVER_YOU_USE))) == NULL)
    nomemory("bestfit utility");
        if ((local_bestfit.utility =
                (int *) malloc(sizeof(struct WHATEVER_YOU_USE))) == NULL)
    nomemory("local_bestfit utility");
        if ((all_pops_bestfit.utility =
                (int *) malloc(sizeof(struct WHATEVER YOU USE))) == NULL)
    nomemory("all_pops_bestfit utility");
    }
*/
    return;
}


void
app_report()
/* Application-dependent report, called each generation by report() */
{
    /* Normal app-dependent end-of-generation printing done here. */
}


void
app_decide_if_converged(flag, num_to_replace_if_converged)
/* This routine is called at the end of each generation of each subpopulation */
/* to give the user the option of checking for convergence of the */
/* subpopulation, or other condition.  When (if) that happens, if user wants  */
/* simply to reinitialize all or part of the subpopulation to random values   */
/* and keep running, user can just set flag to 1 and set */
/* num_to_replace_if_converged to the desired number to reinitialize.  If     */
/* user wants to take some other action, should set flag = 2 and code the     */
```

```
    if (firstcall) {
firstcall = 0;

    }
}


void
app_data(REVISING)
BOOL    REVISING;

/* Application dependent data input, called by init_data() */
/* Ask your input questions here, and put output in global variables */
/* NOTE:  This routine is called only once if the flag */
/* all_subpops_use_same_parameters is TRUE.  For actions that should be */
/* done for each subpopulation whether or not they are using the same  */
/* parameters, use app_init, instead. */
{

    if (REVISING)
return;

}


void
app_init(user_supplied_initialization)
/* application dependent initialization routine called by initialize() */
/* for EACH subpopulation, whether started from input or restarted from */
/* a restartfile. */
BOOL user_supplied_initialization;
{

    user_supplied_initialization = FALSE; /* Make it TRUE if you will supply */
                                          /* the code to initialize the */
                                          /* individuals in oldpop below, in */
                                          /* routine app_user_init_pop(),    */
                                          /* instead of using one of the     */
                                          /* "standard" methods supplied with*/
                                          /* GALOPPS.  */

    elitism = 1;        /* Must set to 0 or 1 to determine whether or not
                          * best indiv. is always preserved in next generation
                          * (1 = yes).  */
    stochastic = 0;     /* Must set to 0 or 1 to specify whether or not
                          * fitness function always returns same value for a
                          * given chrom.--  If always same, set stochastic = 0
                          * to avoid extra evaluations; if changes with env't
                          * or randomly, set = 1 */
}


void
app_user_init_pop(starting_guy_index)
/* If none of the standard methods supplied with GALOPPS will work for */
/* initializing the individuals in oldpop at start of run, or for creating */
/* new "random" guys when popsize is expanded, or when convergence triggers*/
/* partial re-initialization, etc., then you can create the individuals */
/* here, filling oldpop[starting_guy_index] through oldpop[popsize-1].  */
int  starting_guy_index;
{
}


void
app_after_random_init()
/* Application-dependent changes or redoing of initialization, called after */
/* the random initialization is performed and (on restart) individuals have */
/* been read in from restart file. */
{
}
```

they are called, and some of their typical uses are described below.

The standard application template file, appxxxxx.c, is shown below, with additional explanatory comments. The user may want to compare this file with other application file examples (such as app.c, approyrd.c, etc.) to see how the various callbacks are used in particular cases.

```
/*---------------------------------------------------------------------------*/
/* appxxxxx.c - application dependent routines, "fill in the blanks" to      */
/*              define your problem for solution by GALOPPS.  Any functions  */
/*              not needed may simply be left "as is."                       */
/*---------------------------------------------------------------------------*/

#include <math.h>
#include "external.h"

static int firstcall = 1;


void
objfunc(critter)
/* Application-dependent objective function.  THIS IS THE ONLY ROUTINE THE    */
/* USER ABSOLUTELY MUST FILL IN TO DEFINE THE USER'S PROBLEM.  OTHERS ARE     */
/* OPTIONAL, DEPENDING ON THE NATURE AND COMPLEXITY OF THE PROBLEM.           */

/* This is where you code the objective function for your particular problem, */
/* getting the genotype from critter->chrom, and then placing the             */
/* raw (unscaled) fitness for critter into critter->init_fitness.             */
/* Another action which must be accomplished is to increment the current      */
/* evaluation number (neval) and record it in critter->neval.                 */

struct individual *critter;
{

    neval++;
    local_cycle_neval++;

    critter->neval = neval;

    /* The LEAST that any application can do is to replace the line below     */
    /* with a valid statement that assigns a fitness to the genotype          */
    /* (chromosome) passed in (pointed to by critter).  The constant below is */
    /* just so you can check that this "blank" template indeed compiles and   */
    /* runs with the rest of the system as you have configured it.            */

    critter->init_fitness = 10.

}


void
application()
/* This routine should contain any application-dependent computations */
/* that should be performed at the beginning of each generation of a  */
/* GA population or subpopulation. Called by main(), after the        */
/* population is initialized, within the main "generation" loop.      */
{
}


void
app_read_prob_params(REVISING)

BOOL    REVISING;

/* Application-dependent data input, called by init_data() BEFORE it reads */
/* the "standard" fields like numfields, numextrafields, lchrom, etc.  */
/* That means user can read them from a file, for example, HERE, and if they */
/* are set here, the initialization routine WON'T ask for them again.  */
{
    if(REVISING)
return;
```

Applications, using the Automix Utility to Define Representation

For hybrid or mixed-type applications (i.e., where both an optimal ordering of some fields and values for some parameters are sought), this template, appautmx.c, should be used instead, as it is already prepared to interface with the automix.c program for definition of the permutation subfields and position subranges needed. In this form, the hybrid representation is much easier to use than in the older (but still available) form below, apphybxx.c, which used the mixsetup.c utility instead of automix.c.

The alleles in the fields to be permuted are made available to the fitness function in array permarray[], and the real-valued variables are in array value[i]. The user needs only to use them to calculate a fitness, and to return that from decodemixedchrom() for assignment to the unscaled fitness variable, critter->init_fitness.

APPHYBXX.C -- A "Blank" Template for Development of New Hybrid GALOPPS Applications (Using Mixsetup.c)

For hybrid or mixed-type applications for which the automatic utility for defining the representation, automix.c, may not be sufficiently flexible, the user may instead use this template, apphybxx.c, and the companion standalone utility program,mixsetup.c, for definition of the permutation subfields and position subranges needed. In this case, the user should probably first explore the hybrid representation using another standalone program, mixtutor.c, which demonstrates representations without preparing an output file. However, the user is strongly encouraged to use automix.c, instead, unless the "guts" of the representation are of great interest.

CONTENTS OF USER'S APPLICATION-SPECIFIC FILE (APPxxxxx.C)

GALOPPS offers the user a wide range of options for definition of BOTH the problem to be solved and the GA techniques to be used to solve it. Some choice of GA techniques is made by selection of crossover operator, mutation operator, and selection method when the code is compiled (i.e., in the makefile, project file, etc.) and by setting of input parameters (type and magnitude of scaling, problem type, etc.). These are "standard" choices the user makes. The user can also "customize" the system for the particular problem, via problem-specific output, alteration of control logic, addition of genetic operators, etc. -- in MOST cases, WITHOUT having to alter the GALOPPS code at all EXCEPT within the user's application-definition file, which is usually written starting from template appxxxxx.c. This template contains two things:

1) a placeholder function, called objfunc(), in which the user MUST define the fitness function used to evaluate a chromosome, and

2) many "callback functions," which the user may COMPLETELY IGNORE if the "standard" output, program control, and representations are to be used, but which the user MAY use to perform problem-specific I/O, initialization, dynamic alteration of GA parameters, addition of "utility" fields to the chromosome, addition of state variables to be saved and restored with each subpopulation's checkpoint file, and many other actions. These functions, the places where

Fuel consumed above or below 800 liters: -10 points/liter.
Specific fuel consumption: -1 point/(liter/100 km) Driver rest period length: +1 point/hour.

Note that the problem is being solved "blind" -- that is, the program has no knowledge of the distances between cities, but instead learns the amount of fuel consumed and the time elapsed only at the completion of the tour. The variables being solved for are 1) the order of the tour (order of n cities), 2) the optimal horsepower for the vehicle, 3) the average speed to be driven during daylight, which is to be held approximately constant during the tour, 4) the length of the driver's rest period, and 5) the time (in minutes into the race) when the driver's rest period begins. A "nonsense" variable, entitled "radiovolume," is also included in the encoding/decoding scheme, simply to illustrate that the user may map unused fields to anything (or nothing), and to watch the effects of the GA operations on a field which HAS NO EFFECT ON THE FITNESS FUNCTION. NOTE: For simplicity, the objective function assumes that the elapsed time is always at least 10 hours, and there is only ONE night period, even if the race lasts beyond 27 hours.

Random placement of checkpoints for example problems can be created using using the accompanying 'C' main program, cre8rall.c. It creates the distancebetween table, including generating a random start/finish point and its distances as entries (with subscript [numcities]) in the table. This table-creating program can be compiled using the command in file
cre8rall.mak (on Unix systems), or using the .prj file in Borland C.

The values for horsepower, average speed, resttime, reststart, and radiovolume are encoded on the chromosome indirectly, via the "extra" fields, as follows:

```
permsubfield[0]:  determines dayspeed
permsubfield[1]:  determines resttime
permsubfield[2]:  is "thrown away" or ignored
posnsubrange[0]:  determines horsepower
posnsubrange[1]:  determines reststart
posnsubrange[2]:  determines radiovolume (printed, ignored)
```

APPXXXXX.C -- A "Blank" Template for Development of New "Ordinary" GALOPPS Applications

A copy of this code should be used as the template for development of a new "ordinary optimization" application, unless the user finds another example application provided is closer in form to the new problem to be solved. For hybrid or mixed-type applications (i.e., where both an optimal ordering of some fields and values for some parameters are sought), template apphybxx.c should be used instead, as it is already prepared to interface with the mixsetup.c program for definition of the permutation subfields and position subranges needed.

For many applications, the user will need only to type in a few lines in the objfunc(), and can use the remainder of the file without alteration. As applications become very complex, more of the facilities will need to be utilized.

APPAUTMX.C -- A "Blank" Template for Development of New Hybrid GALOPPS

two different possibilities for calling function combineindices, and also specifying fitness as simple sums or products of the two indices. Each has its own advantages and disadvantages, and is helpful in understanding the behavior of this representation. In order to encourage this, code fragments for all methods are included in the file, with three of the four commented out.

APPRALLY.C -- A Road Rally Optimization Problem Illustrating Solution of "Mixed" Problems Involving Simultaneously Permutation and Non-Permutation Elements, and Using Both Permutation Subindices and Position Subindices

Apprally.c demonstrates the use of mixed-type problems(i.e., those which include both fields to be permuted and ordinary numerical fields to be optimized. It uses a single, unified, permutation-type representation on the chromosome, which is then decomposed into the permutation fields and the "extra" fields, which are, in turn, "decoded" into values for integer and/or float variables. It uses a set of functions created for this purpose by Erik Goodman, for inclusion with the GALOPP System. This example solves an illustrative problem, a road rally race with rules defined as follows:

> Find the optimal path which passes all of n checkpoints ("cities") exactly once, returning to the starting point with the goals that the total elapsed time be 24 hours, the total fuel consumed be exactly 800 liters (or litres), the specific fuel consumption (sfc, in liters/100km) be minimized, and the length of the driver's rest period be maximized. Part of the optimization problem is to select the most appropriate horsepower for the vehicle, in order to achieve the required fuel consumption. Of course, that in turn depends on the route selected, the amount of rest time to be taken, etc. An additional constraint makes the problem somewhat harder to solve: When driving at night (which is 7 hours long, and begins 3 hours into the race), the car is limited to maintaining an average speed of exactly 80 kilometers/hour. The average speed to be driven (when not resting) during the daytime portion of the race is assumed to be approximately constant (and is called "dayspeed"), and affects not only the elapsed time, but also the fuel consumption. Fuel consumption is assumed to be related to speed (for each of the two speeds, day and night), depending on horsepower selected for the car, by the formula:
>
> liters/hour used = 20. * ((speed in kph)/horsepower) ^ 2 + 0.25 * (speed in kph)
> * sqrt(horsepower/100.) + 4.0
>
> Points are awarded as follows:
> Incomplete tour: 0 points total (but excluded by solution method).
>
> Complete tour: 100,000 points, minus penalty points for time difference from 24 hours, for fuel consumption difference from 2,000 liters, and for specific fuel consumption, and plus bonus points for driver rest time, as follows:
>
> Elapsed time above or below 24 hours: -10 points/minute.

40,320 possible values for the permutation index, which maps to about 15 bits of resolution for the single parameter in this problem. For a longer search problem for comparison with app1.c, you could use 12 fields, which yields about 479 million distinct codes, or about 28 bits of resolution. (App1.c is can be run using 28 bits, also).

The user will find that the performance of the GA with this representation will depend STRONGLY on which of the 4 supplied permutation-type crossover operators is used. Operators uobx.c and pmx.c definitely perform better than cx.c or ox.c for this problem.

Of course, for other problems, these permutation index values (or others) can be divided into several "subfields" representing different parameters, and each subfield may be mapped to a finite set of real numbers over some interval, if needed. Also, the permutation index and the position index can both be used, for more efficient encoding.

APP1POSN.C -- Goldberg's First Problem, But Using a Position of Extra Fields (Combinations) Representation

This is a simple illustration of using a permutation-type representation and the associated operators to solve a problem which is NOT inherently a reordering problem. The user is prompted for a number of fields to use, which determines the resolution (and size of the search space) of the problem. It uses only the positional encodings (positions of the "extra" fields on the chromosome) to represent the ordinary parameter, and uses "dummy" fields to represent the "perm" fields ("cities", etc.), since there are no true permutation fields being sought in this demonstration problem. (See section on "Mixed-Type Problems" for an explanation of these concepts). For example, selecting 16 "dummy" perm fields and 16 "extra" fields (i.e., 32 total fields) yields combinations(32, 16) or about 601 million possible values for the position index, which maps to about 30 bits of resolution for the single parameter in this problem. Thus, its behavior cam be compared to app1.c when run with a chromosome length of 30 bits.

As with app1perm.c, the performance of the GA will vary strongly, depending on which crossover and mutation operators are used. It is suggested that the user "play" with these with these two applications, and also with use of both indices simultaneously in apprally.c.

Of course, for other problems, these position index values (or others) can be divided into several "subranges" representing different parameters, and each subrange (or subindex) may be mapped to a finite set of real numbers over some interval. Also, the position index and the permutation index can both be used, for a more efficient encoding.

APP1BOTH.C -- Goldberg's First Problem, But Using Both Permutation Index and Position Index Representations

This problem is essentially a combination of app1perm.c and app1posn.c, and is particularly useful for studying the effect of various crossover operators and selection and scaling methods on the hybrid or mixed-type representation. The user is invited to modify the means of combining the two indices, exploring the

APPMATCH.C -- A Hybrid Representation Example Using Automix Setup

File appmatch.c provides a testbed for using the automix tools. It is built from appautmx.c, and defines a fitness function which is optimized when the fields to be permuted are in increasing numerical order, and the real-valued variables have values matching the corresponding permutation field's value. For example, 0,1,2,3,4,5,6 and 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 represents a perfect solution to a length 7 problem. However, this function can easily be made easier or harder to solve by changing the definition slightly. if the real-valued fields are rewarded for matching the absolute field number (i.e., 0.0 is always best for the first position), it is much easier than if the reward is for matching the corresponding numbered field (i.e., 2.0 matches a 2, wherever it is located on the chromosome, which is the default formulation.) As distributed, the function is:
antifitness = 0.10 * fabs(value[i] - permarray[i]) + abs(permarray[i] - i).
Then fitness is calculated by inverting antifitness, so long as antifitness is non-zero. Automix must be run before this problem can be run, and sample input parameters to automix are listed in comments at the top of their associated Onepop and Manypops sample input files, appmatch.in and appmanc8.in.

APPMANSQ.C -- A Manufacturing Job Sequencing Problem with Variable Speedup:

The next example is of a manufacturing sequencing problem. In this problem, the answer sought is the best ordering of manufacture of n jobs, where each job can have its manufacturing time reduced by devoting more "effort" (at a higher cost) to it. Thus, the answers desired are the best order (a permutation of the jobs) and the best speedup for each job. The input parameters defining the problem are the "normal" time for each job, its shortest time (with maximum speedup), its cost per unit time of speedup, its time due for completion, and the cost per unit time for completion after the time due. The objective function to be minimized is the sum of the speedup costs and the total tardiness penalty, which is the sum of the time each job is late times its tardiness cost per unit time. The fitness, which must be maximized, is defined as the reciprocal of the cost (i.e., 1/cost). An auxiliary program, makmanuf.c, can provide a suitable set of input parameters in a file. An alternative program, makmanko.c, provides similar inputs, but with a known optimum solution. Program automix must be run to set appropriate perm subfields and position subranges. Inputs to automix for the sample input files are shown in comments at the top of those input files, apprally.in and rally9po.in.

APP1PERM.C -- Goldberg's First Problem, But Using a Permutation Index Representation

This is a simple illustration of using a permutation-type representation and the associated operators to solve a problem which is NOT inherently a reordering problem. The user is prompted for a number of fields to use, which determines the resolution (and size of the search space) of the problem. It uses only the permutation encodings to represent the ordinary parameter, rather than using the "extra field position" encoding as well, since there are no "true" permutation fields in this problem (see section on "Mixed-Type Problems for an explanation of these concepts). For example, selecting 8 fields yields 8! or

in developing GALOPPS). For a description of these three applications, see APPENDIX SIX, documentation for the ORIGINAL SGA-C, v1.1.

However, in order to introduce the user to many of the new features of the GALOPP System, this release contains many additional example files, including appmatch.c, app0to9.c, approyrd.c, appbtsp.c, app1perm.c, app1posn.c, apprally.c, and 3 "generic" templates for new applications, appxxxxx.c, appautmx.c, and apphybxx.c. **SOME OF THESE APPLICATIONS REQUIRE THAT THE USER FIRST RUN A STANDALONE PROGRAM TO CREATE THE PARAMETER FILE DESCRIBING THE PARTICULAR PROBLEM, OR A FILE DESCRIBING THE DETAILS OF THE REPRESENTATION (FOR HYBRID PROBLEMS). IF YOU WISH TO RUN THE SAMPLE INPUT FILES AS PROVIDED, YOU SHOULD LOOK IN APPENDIX 4 FOR INFORMATION ABOUT THE INPUTS TO BE ENTERED INTO THE STANDALONE SETUP PROGRAMS IN ORDER TO PREPARE FOR THE USE OF THE SAMPLE INPUT FILE. LOOK ALSO AT THE TOP OF EACH INPUT FILE FOR INFORMATION ABOUT RUNNING AUXILIARY PROGRAMS BEFORE RUNNING THE APPLICATION.**

Each application is described briefly below:

APPROYRD.C -- The Royal Road Function

John Holland's 1993 Royal Road Function (challenge problem to attain Royal Road level 3 within 10,000 function evaluations) has been coded as a GALOPPS application example, and is very useful for learning how to select the most appropriate options and tune the parameters of a GA, especially for a binary representation problem. It is difficult to meet Holland's challenge for optimization of this function in the required number of evaluations.

APP0TO9.C -- A Non-Binary Alphabet Demonstration Problem

File **app0to9.c** illustrates the use of a non-binary alphabet.. This application searches for the single best string of length numfields, of the form:
0,1,...,m-1,0,1,...,m-1, ... 0,1,...k,
where m is the user-specified alphabet size. There is no upper limit imposed on either k or m, although it will likely not work with 2**m > MAX_UINT. The user enters the alphabet size and the number of fields, and the program initializes each field to a legal value between 0 and alpha_size-1. All crossover operators (oneptx, twoptx, and unifx) perform crosses ONLY at the boundaries between fields, and mutation changes one field to a different legal value.

APPBTSP.C -- A Blind Traveling Salesman Problem

The tools for using the permutation-type operators in this release are illustrated in appbtsp.c. The user may generate a set of randomly placed cities using a freestanding program called cre8btsp.c (the command for compiling it is in file cre8btsp.mak). The program writes a table which contains the distances between all n (user-input) cities. When running the appbtsp.c application, the user is asked for the name of this distance table. The GALOPPS program then searches for the shortest path among all of these cities (no mandatory starting point is given, so there are n equivalent paths, one for each starting point). As with any reordering-type problem, the user may select a favorite crossover operator and permutation operator at compile time, in the makefile (Unix systems) or project file (PC systems).

GENERAL FORMAT FOR A MASTER FILE

FULLY GENERAL VERSION (ARBITRARY PATTERN)

```
SAMPLE MASTER FILE:     |        EXPLANATION:
subcnt = 4              |    There are 4 subpopulations
subpop = 0 2            |    Subpop 0 has 2 neighbors
  neighbor = 1  2       |    gets, from subpop 1, 2 randomly chosen chrom's
  neighbor = 3  2       |    gets, from subpop 3, 2 randomly chosen chrom's
subpop = 1 1            |    Subpop 1 has 1 neighbor
  neighbor = 0 -1       |    He gets, from subpop 0, its one best chrom
subpop = 2 2            |    Subpop 2 has 2 neighbors
  neighbor = 3  2       |    gets, from subpop 3, 2 randomly chosen chrom's
neighbor = 1 -2         |    gets, from subpop 1, best + 1 randomly chosen chrom
subpop = 3 1            |    Subpop 3 has 1 neighbor
  neighbor = 0  2       |    gets, from subpop 0, 2 randomly chosen chrom's
```

SHORT VERSION (SYMMETRICAL PATTERN)

```
SAMPLE MASTER FILE:     |        EXPLANATION:
subcnt = -4             |    There are 4 subpops; "-" means all use same pattern
subpop = 0 2            |    Subpop 0 has 2 neighbors
  neighbor = 1  2       |    gets, from subpop 1, 2 randomly chosen chrom's
  neighbor = 3  2       |    gets, from subpop 3, 2 randomly chosen chrom's
```

General Format for Master File:

```
subcnt = <number of subpops> or -<number of subpops>
subpop = <subpopindex> <numberofneighbors>
  neighbor = <subpopindex> <FLAG>
  etc.
```

where subcnt, if negative, means that the neighbors will be specified for only ONE subpopulation, and all others will use CORRESPONDING patterns (modulo npops, and neighbor patterns "wrap around" so that subpopulation "npops" is really subpopulation "0", etc.

and FLAG means:
   if positive, the number of randomly chosen individuals to read from the specified neighbor's file;
   if zero, no chromosomes are to be read from that neighbor;
   if negative, the BEST chromosome is to be read, plus |FLAG|-1 additional randomly chosen individuals (so -1 means best only, -2 means best and one other, etc.).

For simplicity, the master file is stored in memory as a fixed-size table, holding a maximum of 50 subpopulations. This is readily alterable by the user, if needed.

NEW EXAMPLE PROBLEM FILES

The SGA-C v1.1 release from which this software was developed contained three example files: app.c, app1.c, and app2.c. These are also included in this release (in a modified form, of course, for compatibility with the changes made

On MULTIPLE processors, or using multiple copies of Manypops on a single processor (in a Unix environment, for example), the subpopulations are run with exactly the same sort of intercommunication and execution, but if multiple processors are used, more than one subpopulation may be in execution AT THE SAME TIME (true parallelism). This new capability of Release 2.20 and beyond is described more fully below.

Regardless of the mode of execution (one process or many processes or many processors), each subpopulation has individual control over all of its own parameters, including the number of generations to run in each cycle, population size, etc. However, some properties (like chromosome length, etc.) are shared among all subpopulations, since individuals must be able to pass from one subpopulation to another. The exchange of individuals between subpopulations is specified by a new form of text file, called a "master" file, *.mst. The master file is a table which contains the number of subpopulations, and for each, its number of neighbors, followed by the number of each neighbor, and how many individuals (and how chosen) are to be read from that neighbor. A SHORT version is also available for situations when the pattern of migration is to be identical for ALL subpopulations (for example, each subpopulation ***n*** reads the best individual from the subpopulations numbered ***n-1*** and ***n+1*** (modulo npops, the number of subpopulations). Details follow the example:

evaluations than higher-numbered subpopulations). To avoid this, chromosomes are always read from .ind files, and the .new files do not replace the .ind files until the END OF EACH CYCLE. This allows all neighbors to be treated "equally;" it also means that, should a run be INTERRUPTED DURING A CYCLE, the user may do a simple restart by throwing away all .new files and restarting at a cycle boundary from the .ind files.

A similar procedure, in which the state of the program is saved in .neu files, then renamed to .ckp files at the end of the cycle, is done to assure that restarts may always be done from the beginning of a cycle, using only the .ind and .ckp files, and all .neu and .new files can be discarded or ignored.

During the running of GALOPPS/Manypops, at the end of any subpopulation's cycle in which it achieved a new global best (unscaled) fitness, the program prints a special output message describing the individual found. The user may find it useful to scan output files for these special lines, which are the only ones containing the word "Achieved", to track the progress of the entire set of subpopulations.

ISLAND PARALLELISM -- GALOPPS/MANYPOPS FOR SIMULATION OF MULTIPLE SUBPOPULATIONS

A major capability of the GALOPPS system is the capability to simulate a number of "island" subpopulations running in parallel. The capability is provided using a second "main" program, Manypops, and different initialization routines. All other code, and user files, are common or compatible between the single population and parallel subpopulations systems. The single-population version, called GALOPPS/Onepop, has a checkpoint/restart capability, which serves as the basis for the new version, GALOPPS/Manypops (an island-parallel genetic algorithm), which allows the user to simulate parallel operation of multiple subpopulations, with periodic interchange of individuals among the various subpopulations. This coarse-grain parallelism is intended to assist the user in avoiding premature convergence on difficult multimodal problems. Files (populations) created via the checkpoint facility of GALOPPS/Onepop may be used to initialize runs of Manypops, and vice-versa (given the proper choices of file names).

Manypops can SIMULATE parallel subpopulations, or, in Release 2.20 and beyond, can use multiple processors to run more than one subpopulation simultaneously, as described below.

PRINCIPLES OF GALOPPS/Manypops (Release 2.20 and Beyond)

GALOPPS/Manypops is built upon the checkpoint/restart capability of GALOPPS/ Onepop. When run on a SINGLE processor from a SINGLE process, it operates by running in sequence a set of subpopulations (labeled 00, 01, 02,...), first reading a checkpoint file for one of the subpopulations, then reading individuals from "neighboring" subpopulations as specified by the user, running for a specified number of generations, then writing a new checkpoint file. Then it proceeds to the next subpopulation. It operates in "cycles," in which each subpopulation receives one turn.

the file prefix for the new checkpoint files to be written during this run (if none given, the prefix specified below for restart files (if any) is used; otherwise, a default "sgackp" is used).

the file prefix for the restart files to initialize each subpopulation (if response is "xyz", then files xyz00.ind, xyz00.ckp, xyz01.ind, xyz01.ckp, ..., must exist in the current directory). If response is a "return" (i.e., no file prefix is given), then the user is prompted for all of the parameters for the run from the keyboard, or input parameters come from the file specified in the command line, in the format described above under "New Format for Input Files (or as "naked" responses identical to keyboard input).

the file prefix for the master file (if none is specified, default file prefix "master" is used. If file cannot be opened or read, then populations are assumed to be independent (no migration).)

IF a new run (not a restart)
    the problem type (permutation or not)

    the size of each field (alpha_size, >=2)

    the number of fields on the chromosome (numfields)

    IF a permutation problem (permproblem == 1)
        the number of extra fields (>= 0, >0 only for "hybrid" reps)

    IF not a permutation problem
        whether to use superuniform initialization or not

    whether or not all subpopulations to be run by this process use the same parameter values (so they are specified only once below).

whether to complete the run without offering the user any further opportunity to modify the parameters of the subpopulations (this question is asked at the beginning of each cycle, until the user responds 'n' for "no"). Once this is selected, the program runs to completion with no further user interaction. It may not be selected in the first cycle if the populations are not being initialized from restart files.

Then, if a restart file prefix was given, the run begins; if not, of if the user wanted to make changes, then the user enters the GA parameters for each subpopulation in turn (from genspercycle through randomseed (first subpop) or convinterval (subsequent subpops), and it executes its first cycle. Note that if restarting from stored files, exchange of chromosomes with neighbors begins with the first cycle, whereas, if the subpopulations are just being initialized, exchanges will commence only at the beginning of the second cycle). When the "individual" files are written by GALOPPS/Manypops, they are initially labeled with the suffix ".new" to indicate that they contain the latest calculations. However, these ".new" files are not the ones read by the neighbors, since that would typically create an assymetry between the neighbors "ahead" of the subpopulation and those "behind" it (typically giving an unfair edge to subpopulations with lower index numbers, because they would have completed more

```
      permproblem = (y|n)
      IF (NOT defined in a file read here by user's app_read_problem_params() )
          alpha_size = nn
          numfields = nn
          IF (permproblem)
              numextrafields = nn
      IF (not permproblem && alpha_size == 2)
          superuniform = (y|n)
      all_subpops_use_same_parameters = (y|n)
no_more_changes = (y|n)
WHILE (no_more_changes == no || restartfileprefix IS BLANK)
    changes_to_this_pop = (y|n)  /* No, unless answered yes */
    IF (changes_to_this_pop == y || (restartfileprefix IS BLANK
              && (popno == 0 || NOT all_subpops_use_same_parameters))
        /* Can specify or change any of the values below. */
        genspercycle = nn
        popsize = nnn
        printstrings = (y|n)
        pcross = [0., 1.0]
        pmutation =  [0., 1.0] /* (NOTE: is probability/chromosome for
                               permproblem == y, and probability/field IF not
                                a permproblem) */
        scaling_window = [-1 - 20] /* (NOTE: must be -1 IF using rank-based
                                       selection) */
        IF (scaling_window < 0)
            sigma_trunc = [0. - ~5.0]
            scalemult = 0.0 or [1.0n - ~2.n] /* (NOTE: must be >1.0 IF
                                                using rank-based selection. */
        crowding_factor = [0 - ~5]   /* (NOTE:  0 means crowding off) */
        IF (crowding_factor > 0)
            incest_reduction = (y|n)
        conv_sigma_coeff = [0. - ~5.0]
        convinterval = [0 - ~100]
        IF (tournament selection is compiled in)
            tourneysize = n
        IF (restartfileprefix IS BLANK && popno == 0)
           USER-DEFINED INPUTS, IF ANY, (in app_data, app_input, etc.) GO HERE
            randomseed = .123
```

EXPLANATION OF INPUT FOR A MANYPOPS RUN

Upon starting the GALOPPS/Manypops, the user is asked:

the number of subpopulations,

the number of the first subpopulation THIS process should calculate

the number of the last subpopulation THIS process should calculate

the number of cycles (restarts of each subpopulation) to run,

what quantity/frequency of output is desired (the "quiet" flag, with settings from 0 (show all output) to 3 (show NO output).

```
numfields = nn
IF (permproblem)
    numextrafields = nn
    IF (not permproblem && alpha_size == 2)
superuniform = (y|n)
maxgen = nnn
IF (strlen(restartfileprefix))
    other_changes = (y/n)
IF (!strlen(restartfileprefix) || other_changes)
/* Can specify or change any of the values below. */
popsize = nnn
printstrings = (y|n)
pcross = [0., 1.0]
pmutation =  [0., 1.0] (NOTE: is probability/chromosome for
    permproblem == y, and probability/field IF not
    a permproblem)
scaling_window = [-1 - 20] (NOTE: must be -1 IF using rank-based
selection)
        IF (scaling_window < 0)
    sigma_trunc = [0. - ~5.0]
    scalemult = 0.0 or [1.0n - ~2.n] (NOTE: must be >1.0 IF
                                      using rank-based selection.
        crowding_factor = [0 - ~5]   (NOTE:  0 means crowding off)
        IF (crowding_factor > 0)
    incest_reduction = (y|n)
        conv_sigma_coeff = [0. - ~5.0]
        convinterval = [0 - ~100]
        IF (tournament selection is compiled in)
    tourneysize = n
        IF (restartfileprefix IS BLANK && popno == 0)
    USER-DEFINED INPUTS, IF ANY, from app_data, app_input, etc., GO HERE
    randomseed = .123
IF (numberofruns > 1)
    (The same fields, beginning after numberofruns line, go here for each
     subsequent run.)
```

MANYPOP INPUT SPECIFICATION

```
/*-----------------------------------------------------------------------------*/
/* appinputtemplate -- specs for sample input file  for  multi-population run*/
/* (Manypops), which actually doesn't run ANY problem.  The IF lines DO NOT  */
/* APPEAR IN AN ACTUAL INPUT FILE.  In all cases in which a default value is */
/* shown at run time, a blank line or blank in the value field can be used to*/
/* indicate acceptance of the default shown.                                 */
/*-----------------------------------------------------------------------------*/
npops = nn
startpopnum = nn  [0, npops-1]
finishpopnum = nn [startpopnum, npops-1]
ncycles = nnn
quiet = [0-3]
checkptfileprefix =   (blank or up to 6 characters, NOT starting with 'z')
restartfileprefix =   (blank or up to 6 characters, NOT starting with 'z')
masterfileprefix =    (blank or up to 8 characters, NOT starting with 'z')
IF (restartfileprefix IS BLANK) /* i.e., start of new run */
```

```
restartfileprefix =
checkptfileprefix = appckp
permproblem = n
alpha_size = 2
numfields = 10
superuniform = n
maxgen = 5
/* Parameters for first run follow */
popsize = 20
printstrings = n
pcross = .5
pmutation = 0.01
scaling_window = -1
/* Note that if scaling_window is not -1, sigma_trunc and scalemult */
/* MUST be removed or commented out, as the program will NOT try to */
/* read them. */
sigma_trunc = 0
scalemult = 1.40
crowding_factor = 0
conv_sigma_coeff = 7
convinterval = 0
randomseed = 0.345
```

Additional fields are required for GALOPPS/Manypops and for solving permutation-type problems. If you use tournament selection (tselect.c), you will also need to enter a field for the tournament size for each population. If you elect crowding (>0), you may also elect incest_reduction. In each case, if you omit a parameter, when you run the GA, it will tell you what it was looking for, and what it found instead, so it is easy to correct.

New in Release 2.20 and beyond, there is a SHORT form for Manypops runs, useful if ALL of the subpopulations (to be run by the process being started) are supposed to use the SAME parameters (e.g., population size, crossover rate, scaling method, etc.) In that case, the user should set the parameter "all_subpops_use_same_parameters" = y (for yes). In that case, the program will prompt for (or read from file) the input parameters for only ONE subpopulation, and will use the same values for all others. This includes the random number seed... in this case, the random number generator just continues operating without reseeding when a new subpopulation is loaded in.

SPECIFICATIONS FOR INPUT FILES -- FOR ONEPOP AND MANYPOPS

ONEPOP INPUT SPECIFICATION

```
/*------------------------------------------------------------------------*/
/* appinputtemplate --  sample input file for single-population run (Onepop),*/
/* which actually doesn't run ANY problem.   The IF lines DO NOT APPEAR IN AN*/
/* ACTUAL INPUT FILE.  In all cases in which a default value is shown at run */
/* time, a blank line or blank in the value field can be used to indicate    */
/* acceptance of the default shown.                                          */
/*------------------------------------------------------------------------*/
numberofruns = 1
quiet = [0-3]
restartfileprefix = (blank or up to 8 characters, NOT starting with 'z')
checkptfileprefix = (blank or up to 8 characters, NOT starting with 'z')
IF (restartfileprefix IS BLANK)   /* i.e., start of run, not restart. */
    permproblem = (y|n)
    IF (NOT defined in a file read by user's app_read_problem_params() )
alpha_size = nn
```

```
            horsepower = indextofloat(permsubvalue[0],
            permsubfieldmax(0,permsubdivpts)
                                 , 20., 400.);

            resttime = indextofloat(permsubvalue[1], permsubfieldmax(1,
            permsubdivpts), 0., 12.);

            dayspeed = indextofloat((long)posnsubindexvalue[0],
            (long)(posnsubrange[0] - 1)
                             , 20., 200.);

            reststart = indextofloat((long)posnsubindexvalue[1],
            (long)(posnsubrange[1] - 1), 0., 20.);

            radiovolume = indextofloat((long)posnsubindexvalue[2] ,
            (long)(posnsubrange[2] - 1)
                                                     , 0., 100.);
```

The net effect of these calls is that dayspeed is now a float variable with posnsubrange[0] possible values from 20. to 200., etc.

File appmansq.c is another example of an application making use of the hybrid representation, without using automix.c and appautmx.c, to solve a manufacturing sequencing with speedup problem for the least cost of total speedup and total tardiness.

NEW FORMAT FOR INPUT FILES

GALOPPS Release 2.05 and beyond includes an optional format for input files to GALOPPS. The user may optionally include on each input line (from keyboard or file) the name of the parameter being entered and an "=" sign before the value. This is not useful from the keyboard, which already prompts for input, but may be valuable when composing disk files of input parameters for SGA. It is suggested that the user prepare a template with all of the parameter names in their correct order, and then simply fill in the values. A sample input file is shown next in this README document. Parameters in the file must still appear in the same order as if the keywords are not specified, but the user has available an aid in composing the file, and the input is checked to be certain that each field name in the file matches the field being read. When the program detects a field DIFFERENT from what it expected, it informs the user of what it found and what it was looking for, then exits. This makes it easy to correct the input file. For your convenience in documenting what a file is set up to do, or for "commenting out" unneeded fields, C-style comment lines:
/* Comment text */
are allowed in the file, but ONLY as standalone lines, not as "trailing" comments.

SAMPLE OF OPTIONAL INPUT FILE FORMATS

(Single Population Version, for solving problem app.c, for example, not a permutation-type problem):

```
/* This is a sample input file for Onepop */
numberofruns= 1
quiet = 0
/* blank restartfileprefix means will take input from this file, not */
/* from a restart file */
```

```
    chromtointarray(tmp, critter->chrom);

    permno = whichpermut(tmp,numfields);

    critter->init_fitness =
        (double) indextofloat(permno, (smallfactorial(numfields) - 1)
,0.,1.0);

    critter->init_fitness = pow(critter->init_fitness, (double) n);
```

The first call transfers numfields fields from the chromosome to an array of ints, called tmp. The second uses tmp to calculate a (long) int, the permutation number which represents the arrangement of the fields in tmp. The third converts that long int index into a (double) value in the range [0., 1.0], and makes that the fitness of this chromosome. The fourth statement raises that fitness to the power n (30 makes it like app1.c). This is all that the user needs to add to the template appxxxxx.c in order to solve this problem using permutation indices. It could all be done with one or two statements, but that would be harder to follow.

2) File apprally.c solves a truly "mixed"-type problem: namely, it solves for an optimal order of visiting a number of checkpoints (cities) while simultaneously optimizing the values of four other parameters which are tightly related to the order in which the checkpoints are visited. It is NOT searching for the shortest tour, but for one which, together with the other variables, yields the MOST POINTS. The problem is fully described at the top of file apprally.c, and also under "New Example Problem Files" below, but briefly, the variables are used as follows:

```
      permsubfield[0]:  determines dayspeed (speed driven during daytime)
      permsubfield[1]:  determines resttime (duration)
      permsubfield[2]:  is defined, but "thrown away" or ignored
      posnsubrange[0]:  determines horsepower (max. HP rating of engine)
      posnsubrange[1]:  determines reststart (starting time of rest)
      posnsubrange[2]:  determines radiovolume (printed, ignored,demo)
```

All that is shown here is the mechanics of getting from the chromosome to a set of values for the four parameters to be optimized with the tour.

```
            /* first, get the city (checkpoint) order                   */
            getpermfields(numcities, cities, critter->chrom);

            /* next, get the permsubvalues                              */
            getextrafields(numcities, numfields, extraarray, critter->chrom);
            permuttosubindices(extraarray, numextrafields, permsubvalue, 3,
            permsubdivpts);

            /* next, get the posnsubindexvalues                         */
            chromtointarray(array, critter->chrom);
            getpositionsubindices(numextrafields, posnsubrange,
            numposnsubranges, array
                       , posnsubindexvalue);

            /* Now convert them to float forms for use in calculating fitness */
```

"perm" and "extra" values into the array (use the functions putpermfields and putextrafields to do that).

fctrnbym: prints a set of factors (NOT necessarily prime) of the number combinations(n,m). Needed in "mixtutor" program, to assist user in picking appropriate subranges for his position subindices, if they are to be used.

countextravalues: returns long ints, depending on flag, which tell the number of values in fields which can be encoded in "mixed-type" chromosomes:

flag == 1: number of permutations of the "extra" fields, which is numextrafields!. Tells number of values available for use by longtofloat() or longtointfields() or for dividing into subfields by permuttosubindices() when only permutation numbering is used (maximum index is one less).

flag == 2: number of possible arrangements of "extra" fields on a "mixed-type" chromosome (IGNORING THEIR VALUES), which is C(numfields, numextrafields), the number of combinations of numfields objects taken numextrafields at a time. Tells number of values for use by longtofloat() or longtointfields() (for example) when only position numbering is used (max index is one less, of coures).

flag == 3: the product of the two values above, which is the number of values available for use by longtointfields when BOTH (unsubfielded) permutation numbering and position numbering are used to encode the long int value.

combineindices: simple offet combination of two index fields, such that resultindex = majorindex * cardinality(minorindex) + minorindex.

The functions listed above are found in one of two files: utility.c or mixedrep.c. Examples of their use are found in the app(xxxxxx).c files and in the program mixtutor, which you should run if you will be solving a mixed-type problem without using appautmx.c and automix.c.

Sample Code to Use a Mixed-Type Encoding -- FOR NON-AUTOMIX APPLICATIONS, ONLY

The principal need for these tools in the objective function is usually to provide a means to supply the order-based fields of the chromosome for use in the fitness function, and to translate the "extra" fields into the ordinary parameters which are then evaluated together with the order-based fields in the fitness function. Two cases are illustrated below:

1) An ultra-simple recoding of Goldberg's problem one (as solved for x^30 in file app1.c), taken from file app1perm.c.

2) A more complex example (apprally.c), illustrating use of permutation subfields together with position subindices to encode several parameters.

1) The "core" of app1perm.c's objective function is only 4 statements:

getpositionsofextras: puts 1's in an array in the positions occupied by the "extra" fields in a mixed-type problem.

longtointranges: is used to "decode" a long int (e.g., a single permutation index or a single position index) into a set of smaller-range long ints, using specified range limits for each subfield. NOTE: all ranges (i.e., upper limits) MUST be divisors of maxvalue, the maximum value of the long field from which they are to be decoded, and their product must be maxvalue.

intrangestolong: is used to assemble a long (for use as a permutation index or position index) from a given set of smaller long int fields. Similar condition on product of subfield ranges: product must be maxvalue.

arraytopositionindex: uses array of permuted values to generate an index which is characteristic of the positions of the "extra" fields among the total set of fields in the array. This is actually a scheme for encoding combinations of numfields objects taken numextrafields at a time. Returns a long int; if this would overflow, you may instead use getpositionsubindices to get a SET of indices which can encode arbitrarily many combinations of fields.

positionindextoarray: PARTIAL-DOMAIN inverse to arraytopositionindex. Takes a (long) position index as a specification of where to place 0's and 1's in an array passed to it. Returns a 1 if successful, 0 otherwise. CAUTION: may only be used with position codes less than about 65000, because larger codes may overflow intermediate calculations. THIS SHOULD NOT LIMIT YOUR CAPABILITIES IN ANY WAY, because this is only an "inverse" routine, used for encoding chromosomes, not decoding them, and if you NEED MORE RANGE, you can get it by using position SUBRANGES and calling function posnsubindicestoarray.

longtofloat: given a long argument, and its range, transforms it linearly into a float in the range of given arguments [lower, upper]. Useful for decoding "multiparameter, mapped, fixed-point"-coded chromosomes (Goldberg).

floattolong: given a float argument, and its range [lower, upper], transforms it linearly into a long int in range [0, maxvalue], where maxvalue is a given long argument. Useful for generating "multiparameter, mapped, fixed-point" codings for chromosomes (Goldberg).getpositionsubindices: given an array of field values ("perm" and "extra" values), and information about the subrange size for each subindex, calculates the position subindices. Uses the BIGNUM representation internally, so can handle arbitrary number of fields without overflow problems. Subindices returned in an array are SHORT ints. If you want bigger numbers, use combineindices to combine them. For help with laying out "legal" subranges, run the program "mixtutor" which is supplied with this package.

posnsubindicestoarray: ALMOST inverse function of getpositionsubindices. Takes a set of position subindices and the definition of their ranges and fills an array with 0's and 1's in the positions which corresond to "perm" fields and "extra" fields, respectively. Does NOT load the actual permuted

inverseindexamong: inverse of function described above; used to go from a permutation index back to the permuted array it represents.

safefactorial: factorial of numbers up to 12 (max before long int overflows); checks arguments, calls smallfactorial. Returns code 1 if successful, 0 otherwise. Factorial value returned in second parameter. Prints a warning message, also, if arguments out of range.

smallfactorial: table lookup; returns (long) j!, 0 <= j <= 12.

combinations: returns (long) combinations (n,m); i.e., number of combinations of n things taken m at a time. Returns 0 if answer would overflow a long int. Uses prime factorization of denominator and division of the factors of the numerator to guarantee that no overflow occurs unless answer is just too big.

primefactorsofnfactorial: places the prime factors of (argument n)! into a data structure "prime". As supplied, n must be <= 100, but adding more primes to structure remedies that if needed.

printprimefactorsofnbym: prints the prime factors of the number combinations(n,m).

returnprimefactorsofnbym: returns the prime factors of the number combinations(n,m) in a data structure.

permsubfieldmax: returns (long)maximum value which can be stored in a permutation subfield (needed for scaling of floats, etc.).

putfield: puts a single int field into a reordering-type chromosome.

getfield: fetches a single int field from a reordering-type chromosome.

chromtointarray: fetches all fields of a reordering-type chromosome into a given array.

intarraytochrom: puts the set of fields (numfields elements ranging from [0, numfields - 1], in any order) from a given array into the chromosome (reordering-type representation only).

getpermfields: fetches all of the fields actually representing the permutation part of a mixed problem.

putpermfields: loads the values in array of "permutation" fields (cities in the traveling salesman problem, for example) onto a chromosome, in the positions marked by 0's in the position array also passed to it.

getextrafields: fetches all of the "extra" or non-permutation fields in a mixed problem (for eventual decoding into the "permutation index" part of the ordinary parameters).

putextrafields: takes array of "extra" fields and array with 1's in positions where they should appear, and puts them on a chromosome.

Because this mixed-type capability is new and conceptually unfamiliar, it is introduced in some detail below.

For problems prepared using appautmx.c and automix.c, NONE of the routines listed below need be used by the user directly. Only a handful of the following routines is typically required for solution of a mixed-type problem using the older, apphybxx.c, template. However, different encodings and therefore different routines may be appropriate for different problems. Many routines are included which go "backwards" from the usual procedure of taking a chromosome and decoding the parameter values which it represents for use by the fitness function and output printing. However, they are included to help the users who need to go both ways (e.g., to initialize chromosomes in particular ways to specified index values, etc.) and to help users to test their encodings in case problems are encountered. Each routine has at the top a description of what it does and what its calling parameters are.

Briefly, the tools do the following (BUT SEE THE EXAMPLES BELOW FOR SAMPLES OF THE SORT OF SIMPLE CALLS WHICH ARE USUALLY ADEQUATE FOR USE OF THIS CAPABILITY).

decodemixedchrom: EXCEPTION: located in the appautmx.c template file. Given a chromosome, decodes the chromosome into two arrays, "permarray[]" and "value[]", which are the chromosome's ordering of the perm fields and values for the value fields, for the user to use to calculate fitness. For this to work, user must use the utility automix.c to define an input file read by the appautmx.c application template.

whichpermut: returns the index of the permutation of the "extra" fields as they appear in the array passed in. This function may be used ONLY if the number of subfields is 12 or fewer, because it uses only a single long int to return the index, and 13 or more fields causes overflow.

permutencode: returns the array passed in, but permuted into the order specified by the long int index also passed in. Therefore, useful only for problems where the number of "extra" fields is 12 or fewer.

permuttosubindices: returns the index of the permutation of the "extra" fields subfields (or whole field, if numsubfields = 1) as they appear in the array passed in. This index is one of the numerical values used to make int or float fields or subfields for representing "ordinary" parameters.

nfactovermfact: returns long int value of n!/m!, or 0 and a printed warning if that would overflow a long int (i.e., >31 bits).

indexamong: when called sequentially with an array which starts with the sorted ints [0,1,...,n], and with a single element of the array, b, returns b's position in the array EXCLUDING those positions already returned. Destroys the array. (Used for calculating permutation indices.)

subindicestopermut: takes an array of permutation subindices (or a single index, if numsubfields = 1) which characterize a permutation (array), and generates the (permuted) set of "extra" fields it represents, in an int array.

30th power. It shows that, for the permutation indices, uniform order-based crossover and pmx are vastly superior to cycle crossover and order crossover for solving this "counting ones" problem. Note, however, that the "ones" being counted are no longer on the chromosome, but in the "intermediate" representation: the permutation index.

App1posn.c uses only the position index, and thus can be used similarly to explore the effects of the various operators on variables driven by the position index.

App1both.c uses both permutation and position indices, and is very instructive if used to explore the effects of various ways of combining the two indices to derive a fitness (addition, multiplication, or either of two orders of calling function combineindices are suggested for user exploration). The method used interacts strongly with the genetic operators used and also with the form of scaling or selection operators used.

The limitations of this approach, and the best ways to apply it to a given problem, are among the subjects of current research of the author.

INTRODUCTION TO THE USE OF THE MIXED-TYPE TOOLS:

AUTOMIX.C for Preparing Hybrid Representations with Template Appautmx.c

A new utility program, **automix.c**, allows the user to prepare a hybrid representation with a minimum of effort. The user need only specify how many variables are to be permuted, how many real-valued variables are needed, what resolution (i.e., how many discrete values) are needed for each real-valued variable, and a fraction which determines a weighting of the permutaions index versus the position index in determining the values of the real variables. This last value, a fraction between 0.5 and 1.0, if set to a low value, tends to slow the search, but make the chromosome longer; a higher value tends to make the search proceed somewhat faster, but at the expense of a somewhat longer chromosome. Applications set up using automix.c are prepared using the appautmx.c template as a starting point. It reads the file written by automix.c, and also queries the user regarding the desired ranges for the real-valued variables. (Of course, if desired, the user may set those ranges by reading from a problem-specific data file, etc.)

When the capabilities of automix.c are not sufficient, a standalone routine, mixtutor.c, is provided to help the user understand and decide on the parameters to use for the new mixed-type representation, and also to illustrate the usage of many of the routines included in this release for solution of mixed-type problems. It assists the user with appropriate sizing and scaling of fields, to help to avoid overflow situations, etc., when running GALOPP's One-pop or Manypops packages. The user can enter trial values for various parameters, and learn how many codes are available, etc.

When the user will use an application based on the "apphybxx.c" template, the program "mixsetup.c" is useful for defining the permutation subfields and position subranges. It can write a file which can be read by apphybxx.c to transfer this information.

tation used in the sample input file appmansq.in, having 5 perm fields and 11 extra fields, for a total number of 16 fields. It will create 6 permutation subfields and 5 position subranges, which are combined to generate 5 real "speedup" variables, each of which has a fixed number of real values in a specified range. The 0th perm subfield is "thrown away" to allow rotational invariance of the "extra" fields, so 6 are needed, rather than 5. The appmansq.c code assigns the first permutation subfield and first position subrange to the first speedup variable, etc. The permutation subfield and position subrange are combined using function "combineindices," with the permutation subfield as the major (more dominant) index. Thus, for the example files included, 5 perm fields and 11 extra fields were selected. That yielded cardinalities of perm subfields of 11, 10, 9, 56, 30, and 24 (remember, the 11 is "thrown away"). The position subranges were assigned so that the largest cardinality subrange went with the smallest cardinality permutation subfield, etc. -- i.e., 5 position subranges were created.

The inputs for automix are described at the top of the Onepop or Manypops input files with which they are associated, in a block of comments.

Apprally.c

The remaining examples use the more complex template, apphybxx.c. The next example application, a somewhat contrived one, is provided in the application **apprally.c**. In this problem, one is trying find an optimal solution to an (unusual) road rally problem, in which the parameters to be optimized are average speed to drive during daylight hours, engine size (horsepower, influencing fuel consumption), duration of rest period, and time to start rest period, but SUBJECT TO the path to be followed between the checkpoints, which the user may select. The shortest path does NOT necessarily yield the most points. Points are awarded for arrival at a specified elapsed time, using a specified amount of fuel, resting for a long duration, and having a low specific fuel consumption (in liters/100km). The GA actually solves this problem "blind," i.e., NOT using information about the distance between each pair of cities, but only about the total length of the tour after it is completed. SEE THE TOP OF FILE apprally.c for a full explanation of the point system, etc. The code also illustrates the use of many of the important functions provided in the mixedrep.c library and its associated routines.

Other Hybrid Examples to Explore/Illustrate Properties of Representation:

Three additional application examples are provided, entitled app1perm.c app1posn.c, and app1both.c, which illustrate the use of the order-based representation to solve a non-order-based problem -- namely, Goldberg's first example problem, which is solved in the usual way in file app1.c. It is essentially a "counting ones" problem, since each additional 1 in the quantity being raised to the power 30 causes an increase in fitness. It is a GA-easy problem. However, in this representation, the relationship between the fields on the chromosome and the final fitness is less direct. It is therefore instructive to use the new indices and reordering-type operators in this problem, for example, to compare the performance of the 4 order-type crossover operators included in this release.

File **app1perm.c** codes the problem in terms of the order-based tools, using only the permutation index to map to the single parameter which is raised to the

TANEOUSLY trading off other technology-related factors influencing the costs or performance of various placements AS A FUNCTION OF THE QUALITY ACHIEVED FOR THOSE PLACEMENTS.

Several example applications are provided:

Appmatch.c

The first example, **appmatch.c**, is the only one using the new, simpler template (appautmx.c) and the automix.c tool for specifying the representation. It defines a chromosome with numpermfields fields to be reordered, and with numpermfields real values associated. The antifitness function to be minimized (fitness is 1./ antifitness) is the sum of the differences between the ordered values in permarray[i] and i and 10% of the difference between permarray[i] and the real value in value[i]. In other words, the optimal solution would have the form:
0, 1, 2, 3, 4, ... and associated real values 0.0, 1.0, 2.0, 3.0, 4.0 ... .
This problem may easily be modified by the user (on one line) to make problems which are fairly easy or VERY hard, depending on whether the reals must come to match fixed values, or must match "moving targets" in the permarray fields. It is illustrative to play with this example, for various combinations of representations (precision, etc.), operators, selection pressures, etc.

The inputs to be provided to the standalone program automix in order to describe the representation used in the sample input files are described in comments at the top of the input files, appmatch.in and appmatc8.in, for Onepop and Manypops, respectively.

Appmansq.c:

The next example is of a manufacturing sequencing problem. In this problem, the answer sought is the best ordering of manufacture of n jobs, where each job can have its manufacturing time reduced by devoting more "effort" (at a higher cost) to it. Thus, the answers desired are the best order (a permutation of the jobs) and the best speedup for each job. The input parameters defining the problem are the "normal" time for each job, its shortest time (with maximum speedup), its cost per unit time of speedup, its time due for completion, and the cost per unit time for completion after the time due. The objective function to be minimized is the sum of the speedup costs and the total tardiness penalty, which is the sum of the time each job is late times its tardiness cost per unit time. The fitness, which must be maximized, is defined as the reciprocal of the cost (i.e., 1/cost). An auxiliary program, makmanuf.c, or another called makmanko.c, can provide a suitable set of input parameters in a file, with the latter program making problems with known optimum solutions. Program automix.c should be run to set appropriate perm subfields and position subranges.

The program automix.c, which writes a file which can be read by appmansq.c, is used to define the way in which the speedup parameters are encoded on the chromosome. In the example problem, which uses files appmansq.in, 5jobsko.par, and 5sub6prm.mix, there are 5 jobs to be sequenced, and therefore 5 speedup parameters, one per job. The automix program should be run and told there are 5 perm fields, 5 real variables, and the minimum number of distinct values for each real variable is 10. The minimum fraction of the values to be coded by the PERM index should be set to 0.8. This will create the represen-

The GALOPP System also uses either of two forms for the POSITION index:
2a) a (signed) long int variable to hold the position index, or
2b) a set of unsigned short integer subranges of the position index, such that the product of the subrange cardinalities is the cardinality of the entire position index.

Method 2a) limits the position index, which the author believes is not readily "decomposed" like the permutation index, to a maximum of about 31 bits. However, method 2b) removes this limitation, using multiword integer arithmetic, extending the range of the position index to be able to encode several thousand bits of information (and, if need be, easily modified by the user to encode an arbitrary amount).

If the application is written starting from template appautmx.c, the user is insulated from calling any of the routines (listed below) to gain access to the permutation fields and real-valued variables. This is the preferred way of using the hybrid representation, whenever it is adequate. To prepare the representation, the user merely runs automix.c (compiled on Unix via "source automix.mak") and entering the number of reordering fields wanted, the number of real variables wanted, the resolution for the real variables, and the fraction of the real variable to be controlled by the permutation (as opposed to the position) index of the hybrid representation (entering a number between 0.5 and 1.0, for example, 0.75 is a decent value for some applications).

If the application must be written based on template apphybxx.c (the older form), the user must examine his/her data needs for non-permutation variables, and then decide which routines to call, (1a, 1b, 2a, 2b routines, etc.) In order to help the user in making the decisions about decomposition of the permutation index and position index, two programs, called mixtutor.c and mixsetup.c, are provided. When mixtutor.c is "made" (compiled together with files mixedrep.c, bignum.c, bigcomb.c, and utility.c, with header file sga.h available in the same directory), mixtutor will tell you about limits and limitations of both indices, and let you experiment with various subfield and subrange definitions, checking the legality of your choices, etc. Mixsetup provides similar assistance, but also writes a file of parameters which can be read by applications based upon the template apphybxx.c. This file specifies the definitions of the permutation subfields and position subranges, as well as the number of extra fields, etc.

More about all of these indexing methods is provided in three appendices: PERMUTATION INDEXING, POSITION INDEXING and BIGNUM LIBRARY.

Description of the Approach, and Examples Appmatch.c, Apprally.c, App1perm.c, App1posn.c, App1both.c, and Appmansq.c

There are, of course, many ways to transform the positions and orders of the "extra" fields into "ordinary" parameters, and the tools provided include only four (which may also be combined in various ways to yield many others). The use of these methods allows one to solve in a single, unified search, problems involving reordering AND setting of parameter values. For example, in VLSI design, one might optimize placement or partitioning of components WHILE SIMUL-

available on the web server at the MSU GARAGe as soon as it is converted into a suitable format (postscript).

Detailed descriptions of the perm and position codes and the ranking algorithms (both are original, and believed to be marginally superior to those in the literature) are provided in the appendices of this User Guide, and an overview is given below.

Using the Mixed-Type or Hybrid Representation

While the theory behind the mixed representation of GALOPPS is not straightforward, using the representation is easy. Sample programs **appmatch.c and appmansq.c** provide easy examples of its use with the new automated mixed representation setup utility, **automix.c**. In essence, the user's chromosome, with its numfields fields in a particular order, is translated into an array **permarray[]** of numpermfields elements in the order specified on the chromosome, and a set of values for the desired number of real-valued variables, in array **value[]**. The user could create a new application beginning either from appmatch.c or from the "blank" template, **appautmx.c**.

The hybrid representation is NOT recommended, in its current form, for solving problems in which the number of real values needed, or the precision to which their values are needed, are high, relative to the number of fields for which an optimal ordering is sought. The chief aim of the approach is to free the researcher to add a few (epistatically involved) real-valued fields to large reordering problems, and to continue to be able to switch easily among the various genetic operators for solving reordering problems (uniform order-based crossover, partially matched crossover, order crossover, and cycle crossover are provided with GALOPPS, as well as swap and random sublist scramble mutation). If a problem has 50 or more fields to be ordered, and a handful of real values whose optimal values depend on the ordering selected, and if high precision in the real variables is not needed, then this rep **might** provide acceptable solutions. The author continues to examine alternative rankings, and alternative mappings from the ranks to the set of real values, which may better preserve the building blocks of the problem.

Several examples (see below) other than appmatch.c, developed before the automix tool was created, are much more difficult to understand, and may be of interest only for someone who wants to explore the theory of the representation. Such persons are invited to contact the author for assistance.

Coding/Decoding Methods for the Mixed-Type Representation

The GALOPP System provides two fundamentally different types of codes, permutation codes and position (or combination) codes. The user may use only one of these coding systems, or may use both simultaneously (they essentially divide the information space provided by "extra" fields into two disjoint parts).

The GALOPP System uses either of two forms for the PERMUTATION codes (or permutation index):

1a) a (signed) long int variable to hold the permutation index, or
1b) a set of long int variables containing a user-specified number of subindices which, together, index all permutations of "extra" fields.

"extra" fields to the n already needed on the chromosome. The contents of the n "city" fields are numbered from {0, n-1}, and the "extra" fields from {n, n+m-1}. The number of extra fields added must be enough to enable using ordinary reordering operators (Partially Matched Xover, Uniform Order-Based Crossover, Order-Based Crossover, Cycle Crossover, etc.) to "shuffle" all the fields (n+m), and then:

1) Pull from the chromosome in their order of appearance all fields with numbers less than n, which ARE the CITY permutation represented on this chromosome.

2) Pull from the chromosome all fields in {n, ..., n+m-1}, in their order of appearance, which yields a permutation of the set {n, ..., n+m-1}. The code just subtracts n from each of these then, putting them back in {0, ..., m-1}. A ranking algorithm (developed by the author) of that permutation yields A NUMBER in the range [0, m!-1], or, if preferred, a SET of numbers whose product of ranges is m!, which can then be mapped to a set of real values in any of a number of ways (and how to best do it is a very open question).

3) Starting from an array of n+m-1 0's, put a one in any position occupied by an EXTRA field on the chromosome. This amounts to a particular combination of n+m things taken m (or n, of course) at a time. Note that this combination is INDEPENDENT of the ORDER of the extra fields (2 above) and the "city" fields (1 above). Then use a combination ranking function (developed by the author) to map this to a (typically) huge integer, using the extended-precision (multiword) "bignum" representation, if needed to avoid overflow.

Thus, the algorithm has translated the positions and order of the m extra fields on any chromosome into a set of independent integer variables, whose product set has cardinality m! * (n+m)!/(m!*n!) or (n+m)!/n! possible values. These integers may be combined, if desired, in many different ways, or may be used individually to code sets of reals, or one index may be discarded, etc. The number of bits of information represented by these two values together, however, approaches (sometimes quite closely) the total number of bits added to the chromosome for the m "extra" fields.

Obvious troubles:
1) mapping these integers back to the real ranges needed is tricky, maybe troublesome; and

2) after the fields are mapped to discretized real ranges, does a point mutation (say a 2-field swap) or other operator wreak havoc on most of the real values?:

> **1) The difficulty of defining a representation is largely alleviated by use of a tool provided with the GALOPPS system: automix.** It allows the user to specify how many reals are needed, what precision is required, and what type of indices (permutation only or a mixture of permutation and position indices) should be used. It then produces the input needed to specify the representation for applications developed using the template appautmx.c.
>
> 2) The second issue is discussed in a paper on hybrid representations which will be

MIXED-TYPE ORDER-BASED (PERMUTATION) AND VALUE-BASED (NON-PERMUTATION) PROBLEMS -- SIMULTANEOUS SOLUTION

(Including Many Routines Useful for Encoding, Decoding, etc., Ordinary Parameters into Permutation Fields, Breaking Fields Into Subfields, etc.)

"Hybrid" problems, involving a simultaneous solution for an optimal ordering of some objects or events, and for values for some ordinary variables, can be addressed with the NEW experimental tools being developed for the GALOPP System. The initial set of tools included may be useful when the values which are optimal for the variables are dependent in some way on the order of the objects which are being permuted. The full set of order-based or permutation-type operators, including uniform order-based crossover, cycle crossover, order crossover, partially matched crossover, random sublist scramble mutation, and swap mutation, is also available for use in solving these "mixed-type" or "hybrid" problems. It is NOT necessary to develop SPECIAL OPERATORS which treat various parts of the chromosome differently, for example. Tools are provided to allow simultaneous solution for the optimal order of the fields actually representing order-based information, AND for the optimal values of the non-permutation "ordinary" variables. (Two example applications, apprally.c and appmansq.c, illustrate this process.) To date, analysis of the "building blocks" of this system is not completed, but it appears to work well at least at relatively modest problem sizes for a class of problems (exploring this domain for solving manufacturing scheduling/planning problems is a topic of our current research in this area). As the work progresses, we will introduce better and far easier to use tools exploiting this new representation and what we have learned about it. The general approach is described below:

By adding a few "extra" fields to be permuted like the others in a permutation problem, it is possible to search not only for the optimal ordering of the permutation fields, but also for optimal values for some real-valued parameters. For example, if one wants to find the optimal order for visiting 10 cities, one can represent each city with 4 bits on a chromosome, for a total of 40 bits. If one adds a few additional non-city fields (with codes from 10-15, for example), then one can use the usual permutation-type genetic operators to find the OPTIMAL rearrangement of the 16 fields. However, the POSITIONS of the SIX EXTRA fields among the 16 total fields, and the ORDERING (PERMUTATION) of the SIX EXTRA fields among themselves can also be translated into a set of "ordinary" parameters. For this example of 6 extra 4-bit fields (chromosome length 64 instead of 40), one can easily, with the tools provided, "recapture" 720 permutation codes and 8,008 position codes, or 5,765,760 distinct codes, which is about 22 of the 24 extra bits of information introduced. These 5 million codes can be "cut up" into subranges for mapping to integer fields, which can, in turn, using tools provided, easily be transformed into linear ranges of fixed-point real numbers (which Goldberg calls a "multiparameter, mapped, fixed-point" coding).

IN OTHER WORDS, here's how it works (since it's not very intuitive):
Given n cities to permute, and, say, 3 numerical variables for which values must be found to a precision of, say, one part in 2**5, the idea is to add m

TOOLS AND OPERATORS FOR SOLUTION OF ORDER-BASED (OR PERMUTATION-TYPE) PROBLEMS, INCLUDING SIX OPERATORS ADDED TO THE GALOPP SYSTEM

(NOTE: See also "mixed-type" problems, below)

Genetic algorithms are frequently used to find solutions to problems which are essentially problems of permuting (or reordering) a set of fields. Scheduling of production facilities, solution of shortest path problems, partitioning and placement for electronic circuits, etc., are common examples. Release 2.05 added facilities for solving such problems as part of the basic GALOPP System. If the user specifies that the problem is a permutation-type problem, the user must select appropriate operators in the makefile (Unix systems) or project file (Borland C, etc.). For permutation-type problems, four different crossover operators are included: uniform order-based crossover (uobx.c), order crossover (ox.c), cycle crossover (cx.c), and partially matched crossover (pmx.c). The first of these is described in Davis, Handbook of Genetic Algorithms, while the other three are described in the Goldberg book. A choice of two mutation-type operators for permutation problems is also provided -- swapping of two randomly selected fields (swap.c) or random sublist scramble mutation (scramble.c), which is described in Davis, Handbook of Genetic Algorithms. The user is referred to both Goldberg and Davis for discussions of the merits of these various operators for various types of problems.

Two routines in utility.c, int2ithruj and ithruj2int, are the lowest-level routines used to encode and decode individual fixed-length integer fields to and from specified bit ranges on the chromosome. Higher-level routines are also available for simpler access, when the application permits (getfield and putfield for individual fields, and chromtointarray and intarraytochrom for unloading/loading the whole chromosome from an int array (numfields and fieldlength are global variables used in these cases). A special routine, initpermpop(), in startup.c (single populations) or initsubp.c (multiple subpopulations), performs random initialization of all fields so that each possible value appears on the chromosome exactly once.

An example file, appbtsp.c, implements a solution to the Blind Traveling Salesman Problem... that is, a TSP in which the solver does not know (or does not make use of) the distances between individual city pairs, but only receives at the end of the tour a measure of the total length. This routine illustrates the use of the facilities provided for solving such problems, and should serve as the basis for the user's development of his/her own file, starting from the "generic" appxxxxx.c file.

REPRESENTING NON-BINARY CHROMOSOMES (ALPHABET SIZE > 2)

Releases 2.30 and later of GALOPPS support the use of non-binary alphabets to represent solutions on the chromosome. That is, bits are grouped into fields n-bits long, of which only some subset may be legal values for the field. For example, if alpha_size (the cardinality of the alphabet) is set to 6, then each field is 3 bits long, but the only legal bit combinations in each field are the binary strings for 0 - 5_{10}. That is, 110_2 and 111_2 are NOT legal values in any field. The user triggers the use of this type of representation by responding NO to "permproblem?" and any integer > 2 for alpha_size. (Alpha_size = 2 provides the usual binary representation.)

Implementation of this feature involved initialization of the populations (there is a new function called in initsubp.c and startup.c, and superuniform initialization of these chromosomes is not allowed), and development of new variations of the traditional crossover and mutation operators.

Crossover (oneptx.c, twoptx.c, and unifx.c) for this representation are all performed only at the boundaries between fields. This prevents generation of any illegal codes, and preserves the fields as the basic atoms of any building blocks. Crossover probability continues to be a per-chromosome probability.

Mutation is done on a per-field basis (mutation rate is per-FIELD (or per LOCUS, if you prefer), just as it is for a binary representation, where a bit is a field). When a field is to be mutated, its value is changed at uniform random to a different one of the legal values. It is forced NOT to remain the same (just as in the binary case), but to remain legal. (Of course, more than one mutation at a time on the same chromosome could conceivably return it to its former value.)

A new example application file illustrating the use of a non-binary alphabet is provided, **app0to9.c**. This application searches for the single best string of length numfields, of the form:

 0,1,...,m-1,0,1,...,m-1, ... 0,1,...k,

where m is the user-specified alphabet size. There is no upper limit imposed on either k or m, although it will likely not work with 2**m > MAX_UINT.

locus, it assigns the first child the allele from parent one with probability 50%, making a new, independent decision for each locus. Child two always receives its value for any locus from the parent NOT used in child one. While uniform crossover often works promotes very rapid search on very simple "counting ones" problems, for example, it renders the concept of *linkage* completely inoperative, so should NOT be used in any system using an INVERSION operator to evolve better linkages. (Such an inversion operator is expected to be available in the next release of GALOPPS.) Uniform crossover also is NOT recommended by this author for problems in which a "natural" representation offers the hope that linkage might help to preserve co-adapted sets of alleles which are relatively close on the chromosome.

CHECKPOINT AND RESTART CAPABILITY:

GALOPPS, in either single-population or multiple-population (parallel) modes, features a checkpoint/restart facility, which allows the user to run for a specified interval, after which GALOPPS saves its state in checkpoint files on the disk, and exits. The user may then restart the program and specify a restart from the saved checkpoint files. The user may examine the population, and may change any parameters, including popsize. If popsize is increased, new individuals to fill the empty slots are generated at random, as at initiation of a run. If the user requests printing of the chromosome strings, it is also done at initialization time.

Checkpointing and restarting are actually accomplished with a pair of files, with suffixes .ckp and .ind, which contain header (program state) information and the population (individuals), respectively. The files share a common prefix, and are always written. If the user specifies no checkpoint filename, they are written to sgackp.ckp and sgackp.ind. When the program is started, the user is allowed to enter a restart file prefix, and if none is entered, the program gets its input from the keyboard (or file) as before.

The user may use restart files as a means of seeding a population with the results of an earlier run. If needed, the user could also create "simulated" restart files by hand for reading in as initial populations. In all cases, if popsize is larger than the number of individuals provided in a file, the program uses random generation to fill empty population slots.

In multiple-population (parallel) mode, the checkpoint files are first written to disk with the suffixes .neu (for header files which will become .ckp files) and .new (for files of individuals, which will become .ind files). At the end of each full cycle, GALOPPS renames each .new-suffixed file with a .ind suffix, and each .neu-suffixed file with a .ckp suffix. This is done to allow OTHER subpopulations to read individuals from neighboring subpopulations and to get the SAME CYCLE's results, regardless of whether the subpopulation being read from is processed BEFORE or AFTER the receiving subpopulation. (Of course, when subpopulations are calculated by more than one PROCESS (on a Unix workstation) or by more than one PROCESSOR (networked PC, workstation, or whatever), there is no longer any attempt to keep the subpopulations "in sync," but the file renaming still operates, for the sake of any subpopulations handled by the same PROCESS.

them, rather than a dissimilar individual. That gives the DISSIMILAR individuals (presumably fewer in number) a BETTER CHANCE to survive. Since FITNESS REALLY MEANS INFLUENCE ON THE NEXT GENERATION (survival of self or producing offspring similar to oneself), crowding produces a DENSITY-DEPENDENT FITNESS FUNCTION, in effect. However, unlike the concept of "fitness sharing," it does not require calculation of average Hamming distances among ALL members of a population, for example.

Since crowding actually implicitly ALTERS THE TRUE FITNESS FUNCTION, the user should expect that different settings for crossover rate, mutation rate, scaling factor, etc., may be needed for effective search than if crowding is not used.

Incest Reduction -- A Form of Mating Restriction

When DeJong-style crowding is used (crowding_factor is set greater than 0), the user is also asked whether or not to use incest reduction. If it is turned on, then a mechanism (developed by Goodman) to reduce the fraction of crossovers between very similar chromosomes is invoked. In this scheme, after the individuals are selected which will survive or produce offspring into the next generation, using whatever selection method the user has chosen, pairs for crossover are picked by choosing the first parent at uniform random from the list of survivors and breeders, then choosing (nominally 3) possible candidates for the other parent, again at uniform random from the list of survivors and breeders. Then the Hamming distance of each candidate from the first parent is calculated, and the one with the greatest Hamming distance is picked for the crossover. Only the two parents actually used are eliminated from the list of eligible parents for the next crossover selection, or for survival (possibly mutated) after crossover is completed. This scheme structures pairs for crossover so as to promote diversity, while preserving all members of the selection-biased pool (or their offspring) into the next generation. It is expected to be especially useful for problems in which good building blocks can exist relatively independently of one another, allowing them to be combined with high probability (the royal road function is such an example).

Restructuring and Addition of More Crossover Operators

The crossover and mutation operators are in separate files, allowing the user to select them independently at compilation time. As an alternative to single-point crossover (the only crossover provided in the original SGA-C), two-point crossover and uniform crossover have been added for manipulating binary representations. These three operators are now in files oneptx.c, twoptx.c, and unifx.c, respectively. The user must select one of these in the makefile (or one of the others, for a permutation-type problem -- see below), and a mutation operator (normally, bitmutat.c for non-permutation problems).

Two-point crossover treats the chromosome as a ring, and places genes from one parent between two randomly selected points on the chromosome, with the remainder of genes coming from the other parent. Two-point crossover is considered by many to be superior to one-point crossover for most applications, and its use is recommended.

Uniform crossover treats each locus as independent of all others, so for each

which are closer to the best individual. The user defines (via input) the criterion for "good"... entering a multiple of the standard deviation. This multiple of the standard deviation is added to the mean to determine a lower fitness limit for being considered "good." The number entered may be a positive or negative floating point number, and will typically be between +3. and -3. For example, 1.0 will result in all individuals at least 1.0 standard deviations above mean fitness to be in the "good" group. An entry of 0.0 means all with higher than average fitness are "good." An entry of -3. means that essentially EVERYONE is "good." A special entry, 7.0, is used to disable this convergence calculation and reporting. Otherwise, it is calculated and reported at the same intervals as the count of ones in each locus, another convergence-related measure.

DeJong-Style Crowding, to Foster Niche Formation

DeJong-style crowding, as described in Goldberg's book, has been added as a means of allowing "niching" of the population -- that is, allowing several distinct groups of individuals (in different "niches") to develop and persist in the population, with lessened pressure by the GA for all to converge toward a single type of individual. This technique is intended to help reduce premature convergence of the population, allowing it to more effectively explore the domain of a multi-modal function. It may be particularly useful for very difficult functions, in which runs of many generations are expected to be necessary to find a global optimum.

DeJong crowding uses an integer "crowding_factor" specified by the user. If it is set to 0, crowding is turned off, and the crossover operation proceeds as usual, with offspring replacing their parents. If crowding_factor is set to 1, then each individual produced by crossoverreplaces a randomly selected individual from the population ALREADY SELECTED according to relative fitness for reproduction or survival into the next generation. If crowding_factor is set greater than 1 (usually to 2, 3, or other small integer), crossover is modified to work as follows:

> Prior to any crossover or mutation, normal fitness-weighted selection (with or without sigma truncation and linear scaling) is used to select the tentative members of the next generation, newpop. Crossover is then performed on pairs of individuals selected at UNIFORM RANDOM from this set (since fitness has already been used to bias the SELECTION FOR SURVIVAL). After a pair of individuals is selected for crossover, the children are calculated as usual. Then, for each child, "crowding_factor" members of the set of tentative survivors are selected (at uniform random), and HAMMING DISTANCE of each chromosome (i.e., number of DIFFERENT BITS) from the child is calculated for the crowding_factor individuals. The child then REPLACES whichever survivor it was CLOSEST TO in Hamming distance. Mutation is then applied to all members of the new population at the specified rate, and fitnesses are calculated for new individuals.

The idea of crowding is that children will tend to replace individuals to which they are SIMILAR, so that, as more individuals of a similar genotype arise in the population, the chances increase that their offspring will replace one of

ONE copy of the current generation's best fitness individual appears in the next generation's population. If turned off, 1) for some selection methods (but NOT for suselect.c), random chance may fail to allow the best individual to be selected for crossover or reproduction; or 2) chance may cause all copies of the current best individual to be subjected to crossover and/or mutation, perhaps resulting in NO copies of it in the next generation, in spite of its higher-than-average fitness. Thus, fitness of best individual of current population could actually decrease, if elitism is not employed. However, depending on the circumstances, some may wish to avoid it, to insure absolute adherence to the sampling probabilities given by the fitness distribution, etc.

Enriched Application-Dependent Callback Functions

In support of the many new functions added to SGA's original set, and in the spirit of the design of the original, the application file templates, appxxxxx.c, appautmx.c, and apphybxx.c, within which the user can develop the code needed to solve a particular problem using GALOPPS, have been enriched and extended. The user is given the opportunity to inject problem-specific code at many points, without modifying the GALOPP System. Thus, most applications can be coded simply by modifying the single file appxxxxx.c. "NOP" model functions for all of the user callbacks have been precoded in appxxxxx.c (for "ordinary" problems) and in appautmx.c and in apphybxx.c (for "hybrid" or mixed-type problems), and any which are not needed may simply be left "as is."

Option to Count and Reduce Objective Function Calls

For greater efficiency of operation, generate.c was modified so that theobjective function can be called ONLY when a chromosome has been subjected to a genetic operation. For time-varying or stochastic problems, or any others for which the evaluation of a chromosome may vary each time it is evaluated, a flag (stochastic) has been added to retain calling of the objective function for every chromosome in every generation, at the user's option.

Tools for Monitoring Convergence of Population(s)

Percentage of Ones at Each Locus:

A facility was added for calculating the proportion of 1 bits present in the population at each locus. Using this measure, which can be calculated and reported at a user-specified interval, it is easy to determine when most individuals have the same values for particular loci, and thus judge the degree of convergence of the population, and determine hyperplanes in which it may be difficult for the population to explore.

Measurement of Resemblance to Best Individual:

This convergence measure, developed by Erik Goodman, calculates a non-standard measure of the diversity of the population as the GA progresses. It compares the chromosomes of all individuals defined as "good" by the user against each other and against the best individual of the current generation. It tabulates the number of good individuals which are closer to the current best individual than to any of the other good individuals, and then calculates the percentage

dow scaling is off, the user may elect either sigma truncation or linear scaling or both (sigma truncation followed by linear scaling of the result).

Window Scaling

Window scaling, similar to that provided in GeneSYS or GAucsd, is provided. The user may elect no window scaling (-1), or may set scaling_window to any integer from 0 to a #defined maximum (currently 19). The fitness is scaled by subtracting from the raw fitness (init_fitness in the code) the LOWEST fitness of any individual in the past scaling_window generations (0 means current generation only; 1 means current plus immediately preceding generation, etc.). Of course, this window scaling differs from GENESIS in that GENESIS subtracts the maximum fitness seen in scaling_window generations, because it uses "anti-fitness", which it tries to minimize; in contrast, GALOPPS uses true fitness, which it seeks to maximize. The user may turn window scaling on or off during a run -- it keeps the needed minimum fitness history whether or not it is active, and operates correctly from first activation, even when files are restarted after a checkpoint halt.

Linear Scaling

Linear scaling is provided as described in Goldberg's book, using scalemult (user input) as the ratio of best fitness to mean fitness. If maintaining the user-specified ratio between best and mean fitness would cause some fitnesses to become negative, the slope of the scaling line is readjusted so that the worst fitness individual receives a scaled fitness of 0, while the mean fitness is maintained. A value of -1 causes linear scaling NOT to be performed.

Sigma Truncation

Sigma truncation, as described in Goldberg's book, has been added. Thisallows the user to specify a multiplier of sigma, the standard deviation, which the sigma truncation feature in file statisti.c uses to alter the fitness function as follows:

> The standard deviation, sigma, of the fitnesses of the current generation is calculated. The fitness, f, is transformed to sigma-truncated fitness, f', using:

f' = max(0.0, f - (fbar - sigma_trunc * sigma)),

where fbar is the mean fitness of the current generation, and sigma_trunc is the multiple of sigma specified by the user. That is, all of the transformed values of f' below 0.0 are truncated to 0. The resulting values (in newpop[j].fitness) are then available for linear scaling, if desired. If the user specifies sigma_trunc = 0.0, sigma truncation is turned off. NOTE: sigma truncation does not preserve the mean fitness of the population.

Optional Elitism

By setting or resetting a compile-time flag, elitism, in the user's app(xxxxx).c problem definition file, the user may use or not use elitism. If elitism is elected, the generation process is modified to INSURE that AT LEAST

ADDITIONAL TOOLS FOR TRADITIONAL PROBLEMS

Additional Selection Methods:

Stochastic Universal Sampling:

An additional form of selection, stochastic universal sampling (file suse-lect.c) has been added, and is recommended over srselect.c and rselect.c. (See Baker, Proc. 2nd Int'l Conf. on GA, pp. 14-21.) because of its small SPREAD about the desired distribution and its freedom from bias. It is faster to compute than all of the other methods, for a serial machine, but does not readily lend itself to parallelization of the calculations for a single population, as Stochastic Remainder Sampling, for example, does. GALOPPS continues to provide rselect (roulette wheel), srselect (stochastic remainder sampling), and tse-lect (tournament selection), as well as ranking (described below).

*** Linear Ranking, followed by SUS:

Whitley (ref.) describes and documents many benefits of using fitness ranking, rather than relative fitnesses, as the basis for selection for survival and mating. For problems in which control of the rate of convergence is at all difficult, ranking is generally more tractable other methods. It is similar in some ways to tournament selection, but has a much smaller spread, of course. It is most useful when GA's in general are most useful: for multimodal problems of high dimensionality. Ranking can be used by compiling in file rnk-slect.c as the selection routine.

Quiet Mode, for Reduced Output Under App(xxxxx).c Control

A new variable, "quiet", has been added to control output. A user might to do very long runs and be interested only in the final result, or in output when a specific events occur. To assist with controlling the amount of output, "quiet" may be set to any integer 0 - 3, from program input or during the run in response to testing in the user's appxxxxx.c code). If quiet == 0, all normal program output is presented to the screen or output file. Quiet == 1 eliminates most normal output (except attaining of new best individuals) after the copyright notice is printed, and at each generation, whenever quiet is not 0, the function app_quiet_report() is called. Here, the user may examine any global variables, etc., reset quiet to another value, or print specific things. Quiet == 2 suppresses all but end-of-cycle reporting, and quiet == 3 suppresses all output except from app_quiet_report (the user may simply choose to leave all output suppressed until the run terminates, and then use a restart of the program from the checkpoint file to examine the results and/or continue the run).

Fitness Scaling Methods:

Three forms of scaling have been added: window scaling, sigma truncation, and linear scaling. Selecting window scaling disallows the other two, but if win-

Format For Master File" in this manual.

9. Run the executable, which will be called Onepop or Manypops.
10. Answer the questions as they appear on the screen. For details on the meanings of parameters, consult this manual as needed. If you are not restarting from a run already completed, you may leave restartfileprefix blank (just "return"). If you want to preserve the checkpointfiles at the end of the run, it is best to specify a prefix for their names (checkptfileprefix, up to 8 characters for Onepop runs, up to 6 characters for Manypops runs).

(If desired, you may read the input from a file and record the output to a file, by specifying two file names on the line invoking the program. See the section on "New Format for Input Files" for details on input, or simply record the interactive process to guide you in generating an input file. If you use the parameter names in the file, as recommended, then if the program finds something other than what it expects, it will tell you what it found and what it expected, making it easy to correct the file.)

Other System:

3. See the section "Modules to Compile" for instructions on how to compile the code to run your particular problem.
4. If your compiler does not accept ANSI-style prototype declarations, you must edit files "external.h" and "sga.h" and comment out the lines "#define PROTOTYPES_ACCEPTED" and "#define SECONDARY_PROTOTYPES_ACCEPTED" per the instructions at the top of external.h.
5. Set any compiler options (FP coprocessor or not, which processor, etc.) to fit your system.
6. Compile the modules described in "Modules to Compile".
7. If running Manypops, you should already have created a (or use an existing) master file, with extension .mst. It will tell each subpopulation what its neighbors are, and what to read from each at the beginning of each cycle. Format of the master file is described in the section "General Format For Master File" in this manual.
8. Answer the questions as they appear on the screen. For details on the meanings of parameters, consult this manual as needed. If you are not restarting from a run already completed, you may leave restartfileprefix blank (just "return"). If you want to preserve the checkpointfiles at the end of the run, it is best to specify a prefix for their names (checkptfileprefix, up to 8 characters for Onepop runs, up to 6 characters for Manypops runs).

(If desired, you may read the input from a file and record the output to a file, by specifying two file names on the line invoking the program (or supplying two arguments under the RUN menu of the BC system). See the section on "New Format for Input Files" for details on input, or simply record the interactive process to guide you in generating an input file. If you use the parameter names in the file, as recommended, then if the program finds something other than what it expects, it will tell you what it found and what it expected, making it easy to correct the file.)

srselect.c, suselect.c, tselect.c, rnkslect.c) to the one desired, for crossover method (oneptx.c, twoptx.c, unifx.c, uobx.c, ox.c, cx.c, pmx.c) to one appropriate for the problem type, and for mutation method (bitmutat.c, swap.c, scramble.c) to one appropriate for the problem type, as desired.

6. Use Build All or Make to compile the code.
7. If running Manypops, you should already have created a (or use an existing) master file, with extension .mst. It will tell each subpopulation what its neighbors are, and what to read from each at the beginning of each cycle. Format of the master file is described in the section "General Format For Master File" in this manual.
8. Exit Borland C++ (if desired) and run the executable, which will be called the same as the project file name, except with extension .exe.
9. Answer the questions as they appear on the screen. For details on the meanings of parameters, consult this manual as needed. If you are not restarting from a run already completed, you may leave restartfileprefix blank (just "return"). If you want to preserve the checkpointfiles at the end of the run, it is best to specify a prefix for their names (checkptfileprefix, up to 8 characters for Onepop runs, up to 6 characters for Manypops runs).

(If desired, you may read the input from a file and record the output to a file, by specifying two file names on the line invoking the program (or supplying two arguments under the RUN menu of the BC system). See the section on "New Format for Input Files" for details on input, or simply record the interactive process to guide you in generating an input file. If you use the parameter names in the file, as recommended, then if the program finds something other than what it expects, it will tell you what it found and what it expected, making it easy to correct the file.)

Unix System:

3. If your compiler does not accept ANSI-style prototype declarations, you must edit files external.h and sga.h to comment out the lines "#define PROTOTYPES_ACCEPTED" and/or "#define SECONDARY_PROTOTYPES_ACCEPTED" as described in the header of external.h.
4. Copy the appropriate file to filename "makefile" (file name to copy is similar to sample problem file, but beginning with "mak" and ending in ".one" for Onepop and in ".man" for Manypops -- SEE APPENDIX FOUR to get the exact name for the Onepop and Manypops makefiles for the selected application.
5. Edit the compiler options in the makefile for any desired optimization, etc. (FP coprocessor or not, which processor, etc.) to fit your system.
6. Edit the options indicated in the makefile for choice of selection method, crossover type, and mutation type, to ones appropriate for the problem at hand. Exit the makefile.
7. Use make to compile the code. (An alternative to steps 4-7 is to simply edit the make file to be used (which loses its original values) and use make -f makfilename.one, for example) to generate the run image.)
8. If running Manypops, you should already have created a (or use an existing) master file, with extension .mst. It will tell each subpopulation what its neighbors are, and what to read from each at the beginning of each cycle. Format of the master file is described in the section "General

MUST then UNLOCK the file by opening the z.... file and writing a 0 to it.

The .ind file must also be LOCKED before the .new file is RENAMED to .ind.

There is some complexity in the initial creation (or finding in existence) of the companion lock files, but that is handled largely by trying to lock the file, and if that fails initially, concluding that the companion z.... file does not exist, and creating it. Of course, later in the run, the companion file MUST already exist, so a failure has a different interpretation later in the run. In general, if a file cannot be locked to read a migrant, the program inserts such a message in the output stream (file, usually), and then ignores that migration. If the all_pops.stt file can't be locked for updating, the user is warned (again, in the output stream) that all subsequent global statistics are probably corrupted, but the run continues. However, if the file to restore a subpopulation for further execution cannot be locked, the process has little alternative but to give up with an appropriate message, since it really can't go on without initializing the subpopulation. Because of these differences in file types and situations, the locking mechanism has a specifiable "persistence" -- how many times it will try to lock the file before giving up -- and this parameter is set to try to minimize failures due to "coincidence" while avoiding lengthy delays to other processors if a single process "dies."
For obvious reasons, the user should avoid specifying any file prefix beginning with "z" to GALOPPS, since files starting with "z" cannot be locked by GALOPPS.

HOW TO GET STARTED RUNNING GALOPPS

NOTE: When GALOPP is run using more than ONE process, then differences in timing of migrations, etc., from process to process, mean that choosing the same SEED for the random number generator NO LONGER guarantees that the results will be the same. This effect is even more obvious when multiple processors are used on a single problem.

0. (Read sections of the manual? for description of capabilities, etc., to see what you might want to use? If unfamiliar with GA's, read Goldberg's book, Chapters 1-4.)
1. Copy all files distributed with the GALOPP System Release 2.35 from its distribution medium into a directory called GALOPPS2.35.
2. Select a sample problem (objective function definition, etc.) to run (example problems all have file names which begin with "app" and end in ".c").

DOS System, Using Borland C++, Onepop or Manypops:

3. Open the corresponding project file (similar name to sample problem, but ending in .prj -- SEE APPENDIX FOUR to get the exact name for the Onepop and Manypops project files for the selected application.)
4. Set the compiler options (FP coprocessor or not, which processor, etc.) to fit your system. Select the LARGE memory model. If your compiler DOES NOT accept ANSI-type prototype declarations, edit the files sga.h and external.h and comment out the lines "#define PROTOTYPES_ACCEPTED" and/or "#define SECONDARY_PROTOTYPES_ACCEPTED" as described in external.h.
5. Replace in the project file the file name for selection method (rselect.c,

"internal" restorations of checkpointed subpopulations during a run, AFTER THE INITIAL READING OF THE restartfileprefix files, are performed using the checkptfileprefix file names. Of course, if the user does not want to PRESERVE the restartfileprefix files for later performing OTHER experiments, the user may specify checkptfileprefix and restartfileprefix to be the SAME, and then the restartfileprefix files will be overwritten during execution. If the user leaves the restartfileprefix empty, the program expects to start from interactive user input or the corresponding text input file (specified on the command line after the program name). In that case, if checkptfileprefix is blank, the default "sgackp" is created automatically. On a restart, If the user leaves the checkptfileprefix empty, the restartfileprefix is used as the checkptfileprefix as well, overwriting the previously saved state. If this is not desired, the user should specify different prefixes.

B) Manypops also writes a single file called "xxxxxx99.stt", where xxxxx is the checkpointfileprefix specified by the user. It contains information about ALL subpopulations in the master file, regardless of what process or processor is doing their calculations. It is the place in which, at the end of each cycle, each process(or) READS the current "all-populations" state, increments the variables by the numbers IT calculated during its just-completed cycle, and then WRITES BACK to all_pops.stt the new "all-populations" state. Information contained in the .stt file includes the raw totals needed to calculate on-line performance, off-line performance, and total number of evaluations performed to date, as well as the genotype of the BEST INDIVIDUAL found so far by ANY subpopulation in any process(or).

File Locking in Manypops -- "z-Files"

Because Manypops may be run simultaneously as more than one process in a processor, or more than one processor, all working on the same problem as defined in the master file (xxxxxxxx.mst), it is possible for one process to be trying to write a certain file while another is also trying to read or write it. Manypops therefore implements a machine-independent form of FILE LOCKING to prevent loss or corruption of data. ***Locking actually operates only when a process is told that it is NOT running ALL of the subpopulations (numpops != finishpopnum - startpopnum + 1). If all are being run in one process, locking is unnecessary, and is bypassed.*** Locking is used on the following file types: .mst, .ind, .ckp, .stt. It is not needed for .new or .neu files. It works as follows:

When a process wants to use file abcdef00.ind, for example, which another might also be using, it first checks file zbcdef00.ind, a "companion" file. If the z.... file contains 0, the file is "unlocked." If the file contains another number, the process "backs off" a process-specific and varying amount of time before attempting again to lock the file. When trying to lock the file, the process opens it, reads it, and if it is 0, rewinds it, writes the process's unique number to it (actually, the lowest number subpopulation it "owns", plus one), then READS it back. It performs the read several times (just "killing time"), and if, at the end, the number it reads MATCHES the number it WROTE, it concludes that it has "LOCKED" the file. It closes the z.... file, then opens the file abcdef00.ind, does any reading or writing it likes, then closes it. It

arbitrarily, so long as it does not conflict with any of the naming conventions (suffixes) used by GALOPPS for other purposes, and so long as the operating system will accept it.

b) The checkpoint header file, which records the state of the Onepop program when the specified number of generations have been computed. It has the suffix ".ckp", and the rest of the name is specified by the user at run time (or in the input file, if used, in the "checkptfileprefix" field, maximum of 8 characters).

c) The checkpoint individuals file, which records all individuals in the population when it is checkpointed. It has the suffix ".ind", and the rest of the name is specified by the user at run time (or in the input file, if used, in the "checkptfileprefix" field). Thus, for example, the user would end up with a pair of files called "c1p322.ckp" and "c1p322.ind", if the checkptfileprefix "c1p322" was specified.

Manypops creates the files above, PLUS:

A) The multiple subpopulation module, Manypops (actually just the main program in file mainmany.c), writes the three file types above, but slightly differently, plus several more. Manypops allows migration of individuals among subpopulations at the end of each "cycle", which is one OR MORE generations of EACH subpopulation which a particular process is responsible for calculating. Migrating "individuals" are read from the ".ind" files written by their neighbors as checkpoint files. However, in order to provide a "fair" environment in which subpopulations numerically higher than a given subpopulation are not "disadvantaged" vis-a-vis subpopulations numerically lower (which would already have been calculated for another cycle), all migrating individuals are taken from ".ind" files, but NEW .ind files are not created until the END of the cycle for the process which writes them. DURING the cycle, they are created with the file suffix .new (for files of individuals) and .neu (for the "header" files storing program state information). Then, at the end of a cycle, the program renames ALL of the .new files it "owns" to .ind files, and all the .neu files to .ckp files. (This is done with careful LOCKING of the files, as explained below.)

Thus, for each subpopulation it is responsible for from the master file, a Manypops process writes .new and .neu files during the cycle, and renames them to .ind and .ckp, respectively, at the END of the cycle. Note that cycles are COMPLETELY asynchronous among multiple processes running on the same CPU (on a Unix system, for example) and among VARIOUS CPU's (on a workstation or PC network, for example).

The file name prefix for .new, .neu, .ind, and .ckp files for Manypop runs must be no longer than 6 (SIX) characters, since Manypops adds a 2-digit subpopulation number to this prefix in making the names for the various checkpoint files.

NOTE: When a run is to be RESTARTED from a set of checkpoint files already written, the file prefix for RESTARTING from is given as the restartfileprefix in the input file (or in response to that question if working interactively). THAT NAME will be used ONLY for RESTARTING ONCE, and all checkpoint files WRITTEN by the new run will be written to the checkptfileprefix file names. The many

WorldWideWeb page of the Genetic Algorithms Research and Applications Group at MSU (//isl.cps.msu.edu/GA/) for progress reporting on their availability, or contact goodman@egr.msu.edu (in U.S.), uskov@aicad.isrir.msk.su (in former Soviet Union), or (less reliable) gabuaa%bepc2@scc.slac.stanford.edu in China. Availability is expected in the December/January timeframe.

A version of Manypops which will do faster (OS-specific) file locking will also be available in late November, 1994. Please monitor the web page of the MSU Genetic Algorithms Research and Applications Group (MSU GARAGe) for further information.

FILES USED IN GALOPPS

Files involved with the GALOPP System are described in six places:

1) Files CREATED by the GALOPP System during execution are described in THIS section, below, and in detail in APPENDIX FIVE.

2) Files used for compiling a version of the GALOPP System are described in the subsection entitled "Modules to Compile" in the "CODE DISTRIBUTION FORMAT" section.

3) The format of the "master" file (suffix .mst) is described in the section entitled, "General Format for Master File."

4) The formats for (optional) input files are described in the section entitled, "New Format for Input Files."

5) All new example problems are described in the section entitled, "New Example Problem Files." This includes a description of any auxiliary programs provided to create test data or specifications for the representation for use with the particular example problem.

6) A brief explanation of each file SUPPLIED with the GALOPP System is provided as Appendix Four.

Files Created by the GALOPP System During Execution

NOTE WELL: Since GALOPPS is intended to be portable, it restricts file names specified by the user OR created by the system to 8 characters, plus a period, plus a 3-letter suffix or extension, for DOS compatibility. Therefore, even when running on a Unix or other system which permits long file names, LONG PRE-FIXES cannot be specified for the intermediate files the program writes (see restrictions below). The user is free, however, to specify on the command line an output file name AS LONG as the system in use will accept, so that results may be labeled conveniently on Unix systems, for example.

Onepop:

During execution, GALOPPS/Onepop creates only three files:

a) The output file specified as the third element of the command invoking the Onepop code (if one is given). This name may be selected by the user

nearly identical mates, and helping to combine building blocks for better global search.

Optional "elitism".

Several fitness scaling methods.

*** Improved "Quiet" mode operation, for reduced output under app control. User can set "quiet" input to 0 (full output), 1 (most output suppressed), 2 (only "milestone" outputs recorded), or 3 (no output except saving of populations, statistics in checkpoint files).

Many new sample applications.

*** An improved format for input of problems from files. Release 2.0 and beyond allowed for an optional keyword on each line in an input file, allowing easy checking for correctness and automatic reporting of what parameter the program expected and what it found. In Release 2.20, a SHORT form is also added for input to Manypops runs, if the parameters for all subpopulations to be calculated by one process (or processor) are to be identical. In that case, the user specifies them for one population, and those values are used for all subpopulations. Only one random number seed is then used, and random numbers continue in the sequence determined by that seed throughout all subpopulations handled by the process.

A limited capability for "seeding" of populations (see checkpoint/restart).

and many other features. They are described below.

NOTE: for anyone interested in **BIN-PACKING** or related applications, we have created a bin-packing application under GALOPPS, based on Emanuel Falkenauer's Grouping GA (GGA) representation. It is extremely fast and effective, solving problems involving packing thousands of objects into hundreds of bins TO OPTIMALITY in a few minutes on a PC. However, it was heavily optimized (hash tables, etc.) and written for "internal" consumption, so is not included in the general distribution because it would be very difficult for us to "support" right now. However, with that caveat, email the author after October 30, 1994, if you would like to get access to a copy of it.

A NOTE ON USER INTERFACES, ETC.:

The GALOPPS system is FILE-DRIVEN, for system independence, and does not include a graphical user interface bundled with it (for window-based specification of input and graphical display of output, for example). A number have been written, but all are operating-system-specific. Thus, GALOPPS is the ENGINE, which was designed to be easy to embed in a GUI. The GUI must simply write the input files (which can be in a human-readable format for easy development) and display the results from the output or checkpoint files, for example. Hooks (via app callbacks) are provided for modifying a run in process, etc. A number of GUI's are currently in development for GALOPPS 2.30 and beyond -- including some based on Motif/Xwindows and others on Microsoft Windows 3.1. If you want a copy of such a system-specific GUI, please consult the

neighbors) for all subpopulations, then the user may now specify it only for subpopulation 0, and the program will do corresponding migrations for all other subpopulations.

*** Performance measures: on-line, off-line, current best and best ever, all measurable within a single subpopulation, within a single cycle on one processor (or process, in a workstation), and lumped for all subpopulations of a problem.

*** Convergence measuring tools, including "lost" and "converged" loci and "percent converged" for all loci, plus callback functions for terminating a run or reinitializing all or part of a population or subpopulation based on its performance or convergence measures.

A non-standard measure of convergence based on percentage of "good" (above some standard-deviation-based fitness cutoff) which are closer to some other "good" individual than to the current optimum individual.

Checkpoint and restart capability.

*** Two additional selection methods: stochastic universal sampling, suselect.c (see Baker, Proc. Second ICGA); and linear ranking (see Whitley) followed by stochastic universal sampling of the resulting probability distribution.

*** Addition of many more genetic operators, including two-point crossover, uniform crossover, uniform order-based crossover, cycle crossover, order crossover, partially matched crossover (pmx), random sublist scramble mutation, two-field swap mutation, and GGA crossover and mutation. The non-permutation operators operate on binary or larger alphabets.

*** Superuniform initialization (optional), which (for binary representations, only) guarantees that a population initially contains ALL of the possible combinations of length less than or equal $\log_2$ (population size).

*** A richer callback structure for defining user applications without needing to alter any of the routines of the GALOPPS system, in the same spirit as the original SGA.

An optional DeJong crowding-type replacement capability. Setting crowding_factor input to:
0 invokes "kid replaces parent" as in SGA;
1 causes kids to replace population members selected at uniform random from among those already selected according to fitness for reproduction and/or survival into the next generation, and
cf>1 causes kids to replace most similar (Hamming distance) of cf randomly selected individuals from among those already selected according to fitness for reproduction and/or survival into the next generation (DeJong crowding).

*** Optional Incest Reduction (only when also using DeJong-style crowding) helps to pair for crossover chromosomes which differ significantly from each other, preserving diversity, reducing "wasted" crossovers among

NEW AND ENHANCED CAPABILITIES OF GALOPPS 2.35 -- IN BRIEF:

GALOPPS was initiated from the Simple Genetic Algorithm as described in Goldberg's book, and inherits some of its characteristics -- for example, it is **generational** (i.e., the next generation is calculated entirely from the current generation without drawing individuals from newly generated offspring). However, in many other ways, it has been extended to allow CHOICES, some of which differ from the single possibilities offered by the original SGA. The user should be familiar with genetic algorithms before using GALOPPS (or any other GA which offers many choices of parameters, methods, etc.), because the problem-solving ability of the GA often depends STRONGLY on appropriate selection of, and harmonization among, the many possible representations, operators, selection methods, parameter settings, etc. GALOPPS does not decide on appropriate parameter settings or operators for a particular configuration -- that is up to the user.

Major ways in which the current system has been extended or improved from the SGA as documented in Goldberg's book (and from the SGA-C v1.1 version) include:

(*** indicates a capability NEW in Release 2.20 - 2.35. Other items were new in Releases 2.0, 2.01, or 2.05.)

*** Representation of problems involving non-binary alphabets, using new crossover and mutation operators restricted to generating only legal values (mutation) and crossing over only at boundaries between fields, and a new initialization routine to generate only legal values in such fields.

Solution of order-based (permutation-type) and mixed (order-based AND non-order-based) problems (see extensive description below).

Capability for simulation (on one computer) of interacting, "island" parallel subpopulations.

*** Capability for running island parallel subpopulations on MULTIPLE processors (workstations or PC's), with migration among the subpopulations in the various processors, so long as processors can read/write from a common file system.

*** Capability for running island parallel subpopulations using MULTIPLE processes on a single workstation, with migration among the subpopulations controlled by the various processes.

*** Complete rewrites of bit-by-bit mutation and of uniform random initialization of binary chromosomes, reducing execution times of these routines 10-30 fold, particularly when hardware floating point is not available.

***Improved capability for user-supplied initialization of the population.

*** New optional SHORT form of the "master" file, xxxxxxxx.mst, which specifies for each subpopulation of a Manypops run which types of individual are to migrate in from which other subpopulations. If this migration pattern is to be the SAME (relative to the positions of

Release 2.35 additions and alterations have been described in the previous section.

If you have begun using any of the earlier releases, the author urges you to convert to the latest release. Release 2.35 has been tested more thoroughly than any of the previous releases, some of which were interim development releases only. The capabilities of Release 2.35, and its level of system independence and testing are much better than earlier releases.

The Beijing 2.01 Release of ESGA/IPGA (February 16, 1994) corrected not only the bugs discovered in the original SGA-C, but also several bugs in earlier alpha test releases of the code from November, 1993 - January, 1994. It adds a print of the random number seed to the program output, generalizes example problem app1perm.c, and adds an additional example problem. Modules updated include generate.c, report.c, tselect.c, mixedrep.c, master.c, utility.c, startup.c, initsubp.c, random.c, app1perm.c, and sgafunc.h. The example make-files for the mixed representation files were also modified. The new example file added is called app1both.c.

Release 2.05 of ESGA/IPGA (subsequently renamed GALOPPS) corrected a very few additional bugs, made the input file I/O code more portable, provided some additional tools for working on hybrid problems (reordering/parameter value problems), and added a new manufacturing sequencing example problem.

Besides additions, several output formats were changed in Release 2.05. Routines altered include: mainpga.c, mixtutor.c, statisti.c, and apprally.c. Also, a new callback (app_new_global_best_report()) was added to every app-type file and template. This is intended to make it easier for the user to turn on "quiet" mode, particularly for parallel runs, and see output only when new global best individuals are found, but to be able to print out whatever information is desired (from the new callback) at that time, even if in quiet mode.

Release 2.20 corrects a severe bug in the optional crowding mechanism which was introduced earlier. Crowding had not been tested extensively, and did not work properly before Release 2.20. All I/O file formats changed in 2.20, due to the many additional features introduced.

Release 2.25 corrected a serious bug in migration among subpopulations.

Release 2.30 corrected a bug in uniform crossover, and introduced non-binary fields (alpha_size > 2) for non-permutation problems.

Release 2.31 added a new app callback function, app_after_random_init(), called from startup.c and initsubp.c, to enable users to "adjust" or redo the initialization of the population for their particular application, if needed. Dummy (empty) functions app_after_random_init() were added to every application file. If the user had written any new application files, they required MODIFICATION by adding this dummy routine (see any of the sample applications).

Release 2.32 included a rewriting of 3 application files: appautmx.c, appmansq.c, and appmatch.c. Their input files have also been augmented to include comments about how to run automix to create the files needed to run the sample input files. The updating of the PostScript version of the guide was completed, and an ASCII text version was dumped from it.

Release 2.32 also included corrections to the guides, slightly improved logic for the "quiet" mode printing (at user request), and corrected comments and better input files for 3 hybrid-type example files, app1perm.c, app1posn.c, and app1both.c. Users preparing their own application files should not be affected by any of these changes, except to see somewhat more output when quiet is set to 1 or 2.

collection of processors of different speeds may be used, with the only effect being that the number of evaluations done to find each emigrant will differ strongly among processors, unless the user chooses to "balance" that by doing less frequent migration from processors which run at slower speeds.

The file locking mechanism was described above in 2), and is most important when multiple processors are utilized. Accounting (for example, number of evaluations before a new "best" individual was found, on-line performance, etc.) is also done as described in 2) above.

Since the parallelism of GALOPPS is accomplished without the need for direct inter-process communication, it is easy to run truly parallel subpopulations even on a number of PC's linked by a typical PC network (Novell net, LANtastic, Pathworks, etc.), or to run simulated parallel subpopulations on a single machine. It is the simplicity of this scheme which renders it so portable. This portability was a key design goal, since this software was developed to support U.S. collaboration with GA researchers in Russia and China, where the typical hardware available is much more likely to be individual or networked PC clones.

The GALOPPS code is so portable and system-independent that parallel subpopulations for a given problem can be run on an arbitrary mixture of PC's, workstations, or any other computer sharing a common file system, so long as the word lengths match (this affects the format of the files they read and write). Any differences in machine speed may be taken into account by the user, if desired, by assigning processors different numbers of subpopulations, or different subpopulation sizes, or other such decisions, so as to keep the evolution among the subpopulations "loosely" in synchronization; however, the operation of the code does not require that to any extent. The rationale for doing it is to keep from "overwhelming" the search of slower processors by introducing to them "advanced" individuals from a faster processor, thereby "poisoning" their own search and likely leading to local premature convergence. However, a rough balancing is certainly all that is indicated, and that is rather easy to achieve.

RELEASE HISTORY AND BUG FIXES -- ESGA/IPGA 2.0 --> GALOPPS 2.32

The Beijing 2.0 release of the ESGA/IPGA system, distributed on January 31, 1994, was prepared during October, 1993 - January, 1994, while the author was conducting research and teaching a graduate-level course on genetic algorithms during his sabbatical leave at Beijing University of Aeronautics and Astronautics, Beijing, China. Students in the class used parts of this code for solution of problems, but no large-scale, comprehensive test program was undertaken. Several bugs were found and repaired in the SGA-C code which served as the starting point for this development (a "stuck at" fault in the initialization of chromosomes, improper rate and counting of mutations, and errors in ithruj2int for reading of integer fields from chromosomes were particularly significant errors).

for simulating. Then these processes run in parallel (of course, the OPERATING SYSTEM is now doing the "checkpointing" or context-switching). The operation of the subpopulations is NO LONGER SYNCHRONOUS among processes, but GALOPPS was designed with this asynchrony in mind. Simultaneous access to the same file is prevented by a simple File Locking protocol, which allows only ONE process to write to (or even read, actually) a file at any time. Attempts to lock files which are locked by another process result in randomly delayed retries. In the event of many consecutive failures, the process either continues (with a notation to that effect) without performing that particular migration (in the event of migration attempts), or fails, if it cannot continue without the file (such as a restart file for one of its own subpopulations). The aborting of one process typically results in "freezing" of the emigrants it provides to its neighbors, but does not interrupt the continued progress of the remaining processes.

Accounting (e.g., reporting the BEST individual found in ANY subpopulation, or tracking the number of evaluations performed in ALL subpopulations before a new "best" individual was found, or the "on-line performance" across ALL subpopulations, etc.) is more difficult in this asynchronous environment. GALOPPS does this by maintaining a file, all_pops.stt, which is updated by ALL processors and ALL processes which are running one problem (i.e., from one file directory using one .mst file). At the start of each cycle, a process reads the current "global" information from the all_pops.stt file, and then uses those values for all reporting it does during the CYCLE (which may be as short or as long as the user sets it to be). Then, at the end of the cycle, it locks all_pops.stt, reads the (probably new, updated by other processes or processors) values from the file, updates its local copies, and writes the new "totals" back to the all_pops.stt file, then unlocks it. Thus, WITHIN A CYCLE, a process is ignorant of things which happened in OTHER processes during its cycle, but "catches up" again at the end of the cycle, and provides the information which all OTHER processes will need to "catch up" with ITS work whenever THEY next complete a cycle. This reporting is "asymptotically correct," and the only improvement possible would be to do this updating "more often," but how often it is done is under user control, so should not be an issue.

The **user must start the process which simulates subpopulation 0 before any other processes** to allow for proper zeroing (at beginning of run) or loading (during a restart) of the cumulative statistics file (which has suffix .stt).

3) **"TRUE" parallel execution:**
Several processors (computers) each run their own copy (or copies, using mode 2) above, as well) of GALOPPS/Manypops, each simulating as many subpopulations as desired. GALOPPS/Manypops allows simulation of ANY number of subpopulations on ANY number of processors (actually up to 99, with default file naming conventions). Again, all cooperating processors must share a common file directory. The MASTER file (suffix .mst) determines the neighbors and migrations, which are conducted asynchronously. A

QUICK OVERVIEW OF 2.35'S ISLAND (COARSE-GRAIN) PARALLEL CAPABILITIES

GALOPPS provides hardware parallelism for genetic algorithm users, even for users of networked PC's, and can also provide simulated parallelism on a single processor. It is designed as an (either true or simulated) coarse-grain- (or "island-") parallel Genetic Algorithm (GA). It consists of two main programs, with executables called (by the Unix makefiles) Onepop (main program in file mainone.c) and Manypops (main program in file mainmany.c), for simulating single populations and multiple (parallel) subpopulations, respectively. (Note that the executables are named according to the "project" name when Borland C++ is used, so the names Onepop and Manypops will be replaced by such names as app1sga.exe, approyrp.exe, etc. All of the project names are listed in Appendix Four.)

The Onepop and Manypops modules share ALL but the main program and initialization code file in common, including using a COMPLETELY identical "app".c file defining the user's problem to be solved. In addition, Manypops also uses code in file master.c and a MASTER data file (suffix .mst) to define the neighboring subpopulations of each subpopulation, and how many individuals are to migrate in from each neighbor each cycle, and whether these include the best individual and/or some number of randomly selected individuals. Onepop is run ONLY when one wants to use a SINGLE population for the problem. Manypops can be run in many modes, all of which allow simulation of more than one subpopulation.

The **ISLAND PARALLELISM** of Manypops (in mainmany.c) is provided in THREE ways:

1) **"Serial" simulated parallelism:**
   One can simulate any number of subpopulations using only a SINGLE PC or workstation, using GALOPPS/Manypops. In this case, each subpopulation is simulated, in turn, for some number of generations called a CYCLE, and at the end of the cycle, that subpopulation is CHECKPOINTED to two files, for later restart. Each population receives one turn per cycle; at the beginning of each population's turn, and according to the master table (originally from a specified file with suffix .mst), it reads one or more individuals from each of its declared neighboring subpopulations. The frequency of this interchange is entirely under user control (determined by the number of generations per cycle, independently settable for each subpopulation). All other GA-related input parameters (population size, crossover rate, etc.) can also be independent among the various subpopulations -- only the compile-time options are constrained to be the same (of course) in all subpopulations.

2) **Multiple-process simulated parallelism:**
   On a single Unix workstation, or on any other system which allows the user to run more than one copy of GALOPPS/Manypops at the same time, with access to the same file directory, the user may run any number of copies of GALOPPS at the same time, each as a separate user process. All copies share the use of a single MASTER file (suffix .mst), which tells all the processes what must be communicated among the subpopulations, whether they are within a process or in separate processes. Each process, in the input file it reads, is told WHICH SUBPOPULATIONS (by number) IT is responsible

the second and later subpopulations, which were not, in fact, used, is no longer necessary. This reorganization allows a new, simpler description of the input files (in intempla.one and intempla.man), but altered ALL sample input files, the ".h" files, and all sample application files, as well as the contents written to the checkpoint header file. Checkpoint header files written by older versions of GALOPPS may not be read by release 2.35, as variables were added to and eliminated from the file format. App files require alteration of only the initialization portions (addition of a parameter to app_init(), and adding of app_user_init_pop() and app_after_random_init(), if updating from a release prior to 2.31).

User-Supplied Initialization of Populations and Post-Initialization "Cleanup" Supported:

Four types of "standard" initialization of chromosomes are furnished with GALOPPS (random values between 0 and alpha_size-1 in each field, random binary (a special case of the preceding one), superuniform initialization of binary chromosomes, and random permutations of n values among n fields). In order to accommodate the needs of users of representations which need to impose additional constraints on initialization, two new callbacks have been added to the app files: user_supplied_init_pop(starting_guy_index) (new in 2.35) and app_after_random_init() (new in 2.31). The first is called INSTEAD of one of the normal four random chromosome generators, if the user sets the flag "user_supplied_initialization" to TRUE in app_init (the default is FALSE). This then requires that the user fill oldpop[starting_guy_index] through oldpop[popsize-1] with valid chromosomes, generated however the user desires. The user may wish to do this "from scratch," or by copying one of the standard routines (from startup.c or initsubp.c) into user_supplied_init_pop() in appxxxxx.c, then modifying it to the requirements of the user's particular representation. The second callback, app_after_random_init(), (introduced in Release 2.31) is called after ANY (including user-supplied) random initialization of chromosomes has been performed, but before the statistics have been run on the initial population. The user may perform any desired transformations, replacements, etc., on the population, to complete the initialization.

Global Statistics Redone:

Since 2.20, the handling of the global statistics file, for parallel (Manypops) runs, has been completely rewritten, so that runs may be restarted using different restartfilenames from the checkpointfilenames used for the first part of the run. This enables conveniently restarting from a particular state, over and over, with the global statistics tracked properly for each of the restart runs (i.e., cumulative from the values at the end of phase one, ignoring subsequent runs made and "rolled back" from there).

Non-Binary Fields Added to "Ordinary" (Non-Reordering) Representation:

An option has been added to work with non-binary alphabets (for example, 6 possible alleles at each locus). Initialization has been rewritten to generate only legal values, and crossover and mutation operators (oneptx, twoptx, unifx, and bitmutat) have been rewritten to perform only legal operations, at field boundaries. Input files (if used) must now contain alpha_size (cardinality of alphabet) and numfields, instead of lchrom (length of chromosome in bits) for all non-permutation representations. (Permutation reps did not use lchrom.)

Russian/American Joint Education/Research Consortium for Intelligent CAD/CAM/CAE and Genetic Algorithms, including members at Moscow State Bauman Technological University, Moscow Aviation Institute, Nizhny Novgorod State University, Penza State University, the Institute of Computation of the Russian Academy of Sciences, and Taganrog State University of Radioengineering, and (2) a consortium of Chinese universities conducting joint research on genetic algorithms with the author and his colleagues at Michigan State University, including the Beijing University of Aeronautics and Astronautics, Tsinghua University, Zhejiang University, the Academia Sinica (Chinese Academy of Sciences), and Beijing Union University. The software is available to others upon request, or via anonymous ftp from the archive server in the Genetic Algorithms Research and Applications Group (GARAGe), Michigan State University. For more information, contact the author by email: goodman@egr.msu.edu. The Case Center's Sister Center in Moscow, the AI/CAD Laboratory of Moscow State Bauman Technological University, V. L. Uskov, Director, can also provide assistance to users or other interested parties in Russia and the CIS.

OVERVIEW OF RELEASE 2.35

For information on getting started running GALOPPS, see "How to Get Started Running GALOPPS" on page 16.

WHAT'S NEW FROM 2.20 to 2.35?

BUG FIXES:

GALOPPS2.30 included several bug fixes to the 2.20 release: If you are using parallel subpopulations at all, you MUST go to 2.30, as the 2.20 version had a serious bug in migration of individuals among subpopulations. Another bug was found in uniform crossover; a bug in DeJong-style crowding was already fixed in release 2.20.

GALOPPS2.31 differs from 2.30 only in that an additional user callback function, app_after_random_init() was added to each application file, called from startup.c (Onepop) and initsubp.c (Manypops). The user who has already created new application functions must simply add a blank (dummy) callback to their app......c file, as illustrated in any of the app files accompanying 2.31.

GALOPPS2.32 differs from 2.31 only in the rewriting of files appmansq.c to use automix, and of appmatch.c and appautmx.c to improve their understandability. Their input files were also augmented with comments about the automix runs which should precede them.

GALOPPS2.35 includes restructuring of I/O and additional capabilities, but no bugs were reported or found between the release of 2.32 and 2.35.

NEW FEATURES:

Input Reorganized:

The sequence in which inputs are requested, by both Onepop and Manypops, has been organized more rationally. Especially in Manypops, any inputs which cannot differ among subpopulations are now requested only once, and used for all subpopulations "owned" by the process. The entry of random number seeds for

ence, for advice and assistance with the packaging, benchmarking, profiling, and other improvements made in Versions 2.05 and beyond. The author appreciates the helpful suggestions made by members of Computer Science 941 (Genetic Algorithms) in the fall semester, 1994. The author also thanks Prof. Rich Enbody (Computer Science) and his class, who are doing PVM parallel implementations of GALOPPS and writing Unix-based graphical user interfaces for the system in fall semester, 1994. Finally, the author is grateful to Brian Zulawinski and Kevin Schaffer, who have been actively exploring and extending GALOPPS in the areas of scheduling (and bin packing) and permutation indexing, respectively.

HARDWARE / OPERATING SYSTEM REQUIREMENTS:

This software has been run on DOS-based, MS-Windows-based, Macintosh, and many Unix-based systems. It has been tested on Sun, Hewlett Packard, and NeXT workstations, using a variety of compilers. On PC systems, it has been tested using Borland C++ (Release 3.1) (but not making use of functionality beyond standard 'C') and using Microsoft C. It has also been tested on Macintosh computers, where it compiles and runs, but it has not been used extensively in that environment. The 'C' code for all of these systems is identical, and allows the user to select via one or two "#defines" whether or not ANSI-style in-line prototype declarations are used, depending on the compiler to be used. The user should specify the characteristics of the compiler and the hardware in use via compilation options before compiling the code (for example, **what processor, whether or not a coprocessor is present, what memory model is to be used, what optimization should be done, etc., are needed by Borland C++**).

BACKGROUND INFORMATION:

This software was developed from the starting framework of the Simple Genetic Algorithm (SGA) system described in David Goldberg's book, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Mass., USA, 1989. This was done in order that the beginner to genetic algorithms can read Goldberg's superb description of the theory and organization of a genetic algorithm, and understand the code in his book, as a stepping stone to understanding the more complex and powerful structure of the GALOPP System. The code has been extended many fold from the original SGA-C, and many sections have been rewritten for greater efficiency (speed), but the organization of the core GA is parallel to that of the original SGA in many respects.

The best way to become familiar with genetic algorithms in general, and with the SGA organization in particular, is to read Chapters 1-4 of Goldberg's book, perhaps along with several chapters of Holland's pioneering work, Adaptation in Natural and Artificial Systems (UM Press, 1975, or MIT Press, 1990). (Copies of both of these books, and several other books and conference proceedings, as well, are furnished to all universities joining the Russian/American Joint Education/Research Consortium for Intelligent CAD/CAM/CAE and Genetic Algorithms (the "ICAD/GA Consortium") and to the universities in China collaborating with the Genetic Algorithms Research and Applications Group of Michigan State University). The prospective user might also want to consult the SGA-C documentation included as Appendix Six of this document, in order to see what was done in translating the system from Pascal to 'C'. All later enhancements and extensions are those of the author, and are described below.

In addition to being available for anonymous ftp and worldwide web distribution, this release is being distributed to two groups of universities: (1) the

AN INTRODUCTION TO GALOPPS

The "Genetic ALgorithm Optimized for Portability and Parallelism" System

DISCLAIMER NOTICE:

Modifications, extensions, enhancements, reimplementations, etc., of all preceding code from which this system is derived are the sole responsibility of Erik D. Goodman, Professor and Director, Case Center for Computer-Aided Engineering and Manufacturing, and member of the Genetic Algorithms Research and Applications Group, Michigan State University; and absolutely no claim is made as to the correctness, fitness or merchantability of this code or the code on which it is based, or the accuracy of the accompanying documentation, for any purposes whatsoever. While the author will be pleased to receive information regarding any bugs discovered, and may, at his option, choose to release revised versions of the code repairing such bugs, no promise or warranty is made that such bugs will be fixed. **All use of this code and documentation is at the sole risk of the user.**

This code, like the original SGA-C, is distributed under the terms described in the accompanying file "NOWARRAN", in accordance with the guidelines of the GNU General Public License. **That means that this code has absolutely NO WARRANTY implied or given, and that the author assumes no liability for any damage resulting from its use or misuse.** The contents of the NOWARRAN files for both GALOPPS and the original SGA-C code are also printed for your convenience on the second page of this document.

ACKNOWLEDGMENTS:

The author wishes to acknowledge the contributions of Mr. Wang Gang, graduate student, Beijing University of Aeronautics and Astronautics, to the writing of the 'C' code for some of the modifications and extensions of this system, and thanks him for help in debugging of some of the author's code, as well, through Release 2.05 of the ESGA/IPGA system.

The author would like to thank Prof. John Holland for introducing him to the concepts of genetic algorithms, in the Logic of Computers Group at the University of Michigan, 1968 - 1971, and for successfully introducing genetic algorithms to the world.

The author is grateful to Beijing University of Aeronautics and Astronautics, Beijing, China, and to Prof. Li Wei and other faculty members of the Department of Computer Science for providing him an excellent environment and facilities for use during his sabbatical leave, September, 1993 - February, 1994, during which time he began the development of this system. Special thanks are due to the author's colleague and friend, Prof. Pei Min, College of Automation Engineering, Beijing Union University, for his great help in making the stay in China both productive and enjoyable.

The author thanks the members of MSU's Genetic Algorithms Research and Applications Group (GARAGe), and especially Bill Punch, Asst. Prof, Computer Sci-

APPENDIX ONE -- COMBINATION INDEXING
A Method for Indexing and Calculating Indices of Combinations(n,m) 64

APPENDIX TWO -- PERMUTATION INDEXING
Permutations and Permutation Indexing: Methods of Encoding and Decoding 69

APPENDIX THREE -- INTRODUCTION TO THE BIGNUM LIBRARY FOR EXTENDED-RANGE POSITIVE INTEGER ARITHMETIC 70

APPENDIX FOUR -- AUXILIARY FILES
Listing of All Auxiliary Files Provided with the GALOPP System, Release 2.35, by the Problem Files They Accompany 74

APPENDIX FIVE -- CONTENTS OF THE FILE TYPES WRITTEN BY GALOPPS 80

APPENDIX SIX -- EXCERPTS FROM SGA-C V1.1 RELEASE DOCUMENT 87

MIXED-TYPE ORDER-BASED (PERMUTATION) AND NON-ORDER-BASED (NON-PERMUTATION) PROBLEMS -- SIMULTANEOUS SOLUTION.................................. 27
Using the Mixed-Type or Hybrid Representation................ 29
Coding/Decoding Methods for Mixed-Type Problems.............. 29
Description of the Approach and Examples Appmatch.c, Appmansq.c, Apprally.c, App1perm.c, App1posn.c, and App1both.c..... 30
Introduction to the Use of the Mixed-Type Tools.............. 33
Sample Code to Use a Mixed-Type Encoding..................... 37

NEW FORMAT FOR INPUT FILES...................................... 39
Sample of Optional Input File Format.................... 39
Specifications for Input Files -- for Onepop and Manypops.... 40
Explanation of Input for a Manypops Run...................... 42

ISLAND PARALLELISM -- GALOPPS/MANYPOPS FOR SIMULATION OF MULTIPLE SUBPOPULATIONS................................ 44
Principles of GALOPPS/Manypops............................... 44
General Format for Master File............................... 46

New Example Problem Files....................................... 46
Approyrd.c -- Holland's Royal Road Problem................... 47
App0to9.c -- A Non-Binary Alphabet Demonstration Problem..... 47
Appbtsp.c -- Blind Traveling Salesman Problem................ 47
Appmatch.c -- A Hybrid Representation Example Using Automix.. 48
Appmansq.c -- A Manufacturing Sequencing Problem............. 48
App1perm.c -- Goldberg's First Problem, but using a Permutation Index Representation.................... 48
App1posn.c -- Goldberg's First Problem, but using a Position of Extra Fields (Combinations) Representation.. 49
App1both.c -- Goldberg's First Problem, but using Both Permutation and Position(Combination) Indices Representation.. 49
Apprally.c -- A Road Rally Optimization Problem.............. 50
Appxxxxx.c -- "Blank" Template for Development of New GALOPPS Applications........................... 51
Appautmx.c -- "Blank" Template for New Hybrid Reps Using the Automix Utility to Specify the Representation...... 51
Apphybxx.c -- "Blank" Template for Development of new GALOPPS "Hybrid" or "Mixed" Applications............... 52

CONTENTS OF USER'S APPLICATION-SPECIFIC FILE (APPxxxxx.C)....... 52

CODE DISTRIBUTION FORMAT.. 59
How to Prepare the GALOPP System for Solving YOUR Problem.. 59
Compiling/Linking the System................................. 60
Modules to Compile... 61

UPDATES AND BUG REPORTING....................................... 63

APPENDICES

TABLE OF CONTENTS

Topic **Page**

Copyright Page....................................Inside Front Cover
No Warranty Notice.. ii
TABLE OF CONTENTS.. v

Disclaimer and Acknowledgements.................................... 1
Hardware/Operating System Requirements............................. 2
Background Information... 2

OVERVIEW OF RELEASE 2.35... 3
Bug Fixes and New Features Since 2.20.............................. 3
Quick Overview of 2.35's Island (Coarse Grain) Parallel Capabilities.................................5
Release History and Bug Fixes, ESGA 2.0 - GALOPPS 2.35.......... 7

NEW AND ENHANCED CAPABILITIES OF GALOPPS 2.35 -- IN BRIEF....... 10

A NOTE ON USER INTERFACES... 12

FILES USED IN GALOPPS... 13
Files Created by the GALOPP System During Execution.............. 13
File Locking in Manypops -- "z-files"............................ 15

HOW TO GET STARTED RUNNING GALOPPS................................ 16

ADDITIONAL TOOLS FOR TRADITIONAL PROBLEMS......................... 19
Additional Selection Methods...................................... 19
Quiet Mode, for Reduced Output under App Control.................. 19
Fitness Scaling Methods... 19
Window Scaling.. 20
Linear Scaling.. 20
Sigma Truncation.. 20
Optional Elitism.. 20
Enriched Application-Dependent Callback Functions................. 21
Option to Count and Reduce Objective Function Calls.............. 21
Tools for Monitoring Convergence of Populations.................. 21
Percentage of Ones at Each Locus.................................. 21
Measurement of Resemblance to Best Individual..................... 21
DeJong-Style Crowding, To Foster Niche Formation.................. 22
Incest Reduction -- A Form of Mating Restriction.................. 23
Restructuring and Addition of More Crossover Operators........... 23

CHECKPOINT AND RESTART CAPABILITY................................. 24

REPRESENTING NON-BINARY CHROMOSOMES (ALPHABET SIZE > 2)......... 25

TOOLS AND OPERATORS FOR SOLUTION OF ORDER-BASED (PERMUTATION-TYPE) PROBLEMS, INCLUDING SIX OPERATORS ADDED TO THE GALOPP SYSTEM............... 26

(This Page Intentionally Left Blank.)

GALOPP SYSTEM NOTICE

NO WARRANTY

BECAUSE THE GALOPP SYSTEM IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE AUTHORS OF THE GALOPP SYSTEM PROVIDE THE GALOPP SYSTEM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE GALOPP SYSTEM PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL THE AUTHORS OF THE GALOPP SYSTEM, AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE THE GALOPP SYSTEM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH PROGRAMS NOT DISTRIBUTED BY FREE SOFTWARE FOUNDATION, INC.) THE PROGRAM, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SGA-C NOTICE
(Covering the software from which the ESGA/IPGA System, and then eventually, the GALOPP System, was originally derived)

NO WARRANTY

BECAUSE SGA-C IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE AUTHORS OF SGA-C PROVIDE SGA-C "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE SGA-C PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL THE AUTHORS OF SGA-C, AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE SGA-C AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH PROGRAMS NOT DISTRIBUTED BY FREE SOFTWARE FOUNDATION, INC.) THE PROGRAM, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

HEREDITY

GALOPPS is a distant descendant of:
SGA-C, v1.1, modifications by Jeff Earickson, Boeing Company,
which was based on SGA-C, by Robert E. Smith, Univ. of Alabama,
which was based on SGA (in Pascal), (c) David E. Goldberg 1986,
All Rights Reserved,
as described in GOLDBERG, David E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, Massachusetts, USA, 1989.

GALOPPS
The "Genetic ALgorithm Optimized for Portability and Parallelism" System
(C) Copyright, 1994, Erik D. Goodman
Michigan State Univesity
East Lansing, Michigan 48824 USA

ESGA/IPGA SYSTEM (on which GALOPPS IS BASED)
The Extended SGA and Island Parallel Genetic Algorithm System
(C) Copyright, 1993, 1994 Erik D. Goodman
Michigan State University
East Lansing, Michigan 48824 USA

AN INTRODUCTION TO

GALOPPS

The "Genetic ALgorithm Optimized for Portability and Parallelism" System

(c) Erik D. Goodman 1993, 1994, Michigan State University.

RELEASE 2.35 (November 4, 1994)

(Obsoleting GALOPPS Releases 2.05 - 2.32 and ESGA/IPGA Releases 1.0 - 2.05)

Written by
Erik D. Goodman, Professor
MSU Genetic Algorithm Research and Applications Group (GARAGe)
Intelligent Systems Laboratory, Dept. of Computer Science, and
Case Center for Computer-Aided Engineering and Manufacturing

Professor, EE, ME
Director, MSU Manufacturing Research Consortium
Director, Case Center for
Computer-Aided Engineering and Manufacturing
Michigan State University, East Lansing 48824

TECHNICAL REPORT # 94-11-01

INTELLIGENT SYSTEMS LABORATORY,
DEPARTMENT OF COMPUTER SCIENCE
AND
CASE CENTER FOR COMPUTER-AIDED
ENGINEERING AND MANUFACTURING

MICHIGAN STATE UNIVERSITY, EAST LANSING