

AN INTRODUCTION TO
GALOPPS

**The "Genetic Algorithm Optimized
for
Portability and Parallelism" System**

(c)MichiganStateUniversity,1993,1994,1995,1996.

RELEASE 3.2 (July 31, 1996)

(Obsoleting GALOPPS Releases 2.05 -3.02 and ESGA/IPGAR Releases 1.0 -2.05)

Written by

**Erik D. Goodman, Professor and Co -Director
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 # 96-07-01

**GENETICALGORITHMS
RESEARCH AND APPLICATIONS GROUP (GARAGE)
INTELLIGENT SYSTEMS LABORATORY,
DEPARTMENT OF COMPUTER SCIENCE
AND
CASE CENTER FOR COMPUTER -AIDED
ENGINEERING AND MANUFACTURING
MICHIGAN STATE UNIVERSITY, EAST LANSING**

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
Erik D. Goodman
(C) Copyright, 1994, 1995, 1996, Board of Trustees, Michigan State University
East Lansing, Michigan 48824 USA

ESGA/IPGAS SYSTEM (on which GALOPPS IS BASED)

The Extended SGA and Island Parallel Genetic Algorithm System
Erik D. Goodman
(C) Copyright, 1993, 1994, Board of Trustees, Michigan State University
East Lansing, Michigan 48824 USA

GALOPPS SYSTEM NOTICE

NOWARRANTY

THE GALOPPS SYSTEM IS DISTRIBUTED UNDER THE PROVISIONS OF THE GNU GENERAL PUBLIC LICENSE (SEE FILE 'LICENSE' IN THE DOCS DIRECTORY). BECAUSE THE GALOPPS SYSTEM IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NOWARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE AUTHOR OF THE GALOPPS SYSTEM PROVIDES THE GALOPPS SYSTEM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS 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 GALOPPS 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 AUTHOR OF THE GALOPPS SYSTEM, AND/OR ANY OTHER PARTY WHOM ANY MODIFY AND REDISTRIBUTE THE GALOPPS 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-CNOTICE

**(Covering the software from which the ESGA/IPGA System, and
then eventually, the GALOPPS system, was originally derived)**

NOWARRANTY

BECAUSE SGA -C IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NOWARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE AUTHOR OF SGA -C PROVIDES 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 AUTHOR OF SGA -C, AND/OR ANY OTHER PARTY WHOM ANY 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.

(ThisPageIntentionallyLeftBlank.)

TABLE OF CONTENTS

Topic	Page
CopyrightPage.....	InsideFrontCover
NoWarrantyNotice.....	iii
TABLE OF CONTENTS.....	v
DisclaimerandAcknowledgements.....	1
HARDWARE/OPERATING SYSTEM REQUIREMENTS.....	2
BACKGROUND INFORMATION.....	2
WHAT ELSE IS COMING OUT WITH GALOPPS?	3
ReleaseHistoryandBugFixes,ESGA2.0 -GALOPPS3.2.....	4
ANoteOnUserInterfaces,etc.....	8
NEW AND ENHANCED CAPABILITIES OF GALOPPS 3.2 FROM GOLDBERG'S SIMPLE GENETICAL ALGORITHM --IN BRIEF.....	9
HOW TO GET STARTED RUNNING GALOPPS... ..	13
WRITING YOUR OWN APPLICATION.....	15
OVERVIEW OF RELEASE 3.0'S ISLAND (COARSE -GRAIN) PARALLEL CAPABILITIES.....	22
FILES USED IN GALOPPS.....	24
Files Created by the GALOPPS System During Execution.....	24
NEW CAPABILITIES OF GALOPPS.....	26
Graphical User Interface.....	26
Differing Alphabet Sizes (Field Lengths).....	26
Multiple-Field Mutation Operator.....	27
Boltzmann Scaling of Fitnesses.....	27
Optional Automatically Controlled Selection.....	27
Falkenauer's Grouping Genetic Algorithm Representation and Operators (GGA)..	28
Threaded GALOPPS.....	28
Subpopulation-Level Inversion Operator and Related Support.....	28
Multiple Representations --Different Rep for Each Subpopulation.....	30
Command Line Parameter Overrides.....	30
Additional Selection Methods.....	31
Stochastic Universal Sampling.....	31
Linear Ranking, Followed by SUS.....	31
Representing Non -Binary Chromosomes (Alphabet Size > 2).....	32
Restructuring and Addition of More Crossover Operators for "Value -Based" Problems.....	33
Six Operators and Other Tools Added for Solution of Order -Based (or Permutation -Type) Problems.....	33
Quiet Mode, for Reduced Output under App Control.....	34
New Fitness Scaling Methods.....	34
Window Scaling.....	34
Linear Scaling.....	33
Sigma Truncation.....	34
Optional Elitism.....	35
Tools for Monitoring Convergence of Populations.....	35
Percentage of Ones at Each Locus.....	35
Measurement of Resemblance to Best Individual.....	35
DeJong-Style Crowding, ... Replacement, Niche Formation.....	36

IncestReduction --AForm ofMatingRestriction.....	36
EnrichedApplication -DependentCallbackFunctions.....	37
OperatorUsageAutomaticallyDocumentedtoOutput.....	37
User-SuppliedInitializationofPopulationsandPost -Initialization "Cleanup"Supported.....	37
OptiontoCountandReduceObjective FunctionCalls.....	37
MigrationCapabilitiesSignificantlyEnhanced andMultipleRepresentationsSupported.....	38
CheckpointandRestartCapability.....	39
NEWFORMATFORINPUTFILES.....	39
SampleofOptionalInputFileFormat.....	40
SpecificationsforInputFiles --forOnepopandManypops.....	41
ExplanationofInputforaManypopsRun.....	43
ISLANDPARALLELISM --GALOPPS/MANYPOPSFORSIMULATIONOF MULTIPLESUBPOPULATIONS.....	44
PrinciplesofGALOPPS/Manypops.....	45
GeneralFormatforaMasterFile.....	46
RUNNINGMANYPOPSONMULTIPLEPROCESSORS.....	47
NEWEXAMPLEPROBLEMFILES.....	48
Approyrd.c --Holland'sRoyalRoadProblem.....	48
App0to9.c --ANon -BinaryAlphabetDemonstrationProblem.....	49
Appbtsp.c --BlindTravelingSalesmanProblem.....	49
Appdemoi.c --DemoProblemforInversionOperator.....	49
Appdifrp.c --DemoProblemforDifferentRepresentationsandInjection Architecture.....	49
Appbpp.c --DemoofGGRepresentationforBinPackingProblem.....	49
Appxxxx.c --"Blank"TemplateforDevelopment ofNewGALOPPSApplications.....	50
CONTENTSOFUSER'SAPPLICATION -SPECIFICFILE(APPxxxx.C).....	50
CODEDISTRIBUTIONFORMAT.....	58
HowtoPreparetheGALOPPSSystemforSolving YOURProblem.....	60
Compiling/LinkingtheSystem.....	61
ModulestoCompile.....	61
UPDATESANDBUGREPORTING.....	63

APPENDICES

APPENDIXONE --AUXILIARYFILES	
ListingofAllAuxiliaryFilesProvidedwiththeGALOPP System,Release3.0,bytheProblem FilesTheyAccompany.....	64
APPENDIXTWO --CONTENTSOFTHEFILETYPESWRITTENBYGALOPPS...	68
APPENDIXTHREE --EXCERPTSFROMSGA -CV1.1RELEASEDOCUMENT.....	74

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 this 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 SGAC, is distributed under the terms described in the accompanying file "NOWARRANT", in accordance with the guidelines of the GNU General Public License. **That means that this code has absolutely NOWARRANTY implied or given, and that the author assumes no liability for any damage resulting from its use or misuse.** The contents of the NOWARRANT files for both GALOPPS and the original SGAC code are also printed for your convenience on the second page of this document. A copy of the GNU Public License is available in subdirectory docs, in file "license".

This **WARNING** is intended to be **REAL**: the many features of GALOPPS mean that there are many possible combinations of features which may never have been tested, or which may contain bugs which were not discovered even though they affected test runs. At each new release, bugs in former releases have been found (and corrected), and the "last" bug will probably never be found. As with any GA code, the things can **appear to work** while doing something quite different from what was intended. Before investing large amounts of time and/or resources in applications based on this (or, the author believes, ANY) GA code, the users should **VERIFY** the correctness of the operations with the features selected. The author found several significant bugs, for example, in the original SGAC code from which this software was initiated, even though that code has been available for several years.

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 thank 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, Assoc. Prof., Computer Science, and Co-Director of the GARAGE, for advice and assistance.

with the packaging, benchmarking, profiling, and other improvements made in Versions 2.05 and beyond. Bill's persistence in getting the djgpp 'C' compiler packaged up for distribution with PC versions of GALOPPS and lilgp has been a great help.

The author also appreciates the helpful suggestions made by members of Computer Science 941 (Genetic Algorithms) taught by Bill Punch in the fall semester, 1994, and especially thanks Dan Judd, who furnished code for four of DeJong's classical GA test functions and reported several bugs. The author thanks Prof. Rich Enbody (Computer Science) and his class, who did PVM parallel implementations of GALOPPS and wrote Unix -based graphical user interfaces for the system in fall semester, 1994; Prof. Enbody and Vera Bakic, who developed a PVM implementation of GALOPPS 3.0 for a MPP (or distributed workstation network). The author is grateful to Brian Zulawinski, who actively explored and extended GALOPPS in the areas of scheduling (and binpacking), and implemented the GGA (invented by Emanuel Falkenauer) under GALOPPS. The author thanks Leslie Thomas Wall Hoppensteadt, who greatly improved the checkpoint file handling process in Release 3.0, and who built in the command line parameter overrides, using code written originally for lilgp (another product of the MSUGARAGe) by Doug Zongker. Leslie Hoppensteadt also revised much of the code for reading inputs. Yizheng Feng, with the assistance of Prof. Min Pei, wrote the new Tcl/Tk -based Graphical User Interface for GALOPPS 3.2 (Unix distribution).

HARDWARE/OPERATING SYSTEM REQUIREMENTS:

This software is written in 'C', and 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'), Microsoft C, and djgpp. It has also been tested on Macintosh computers, under the Metrowerks Code Warrior compiler. The 'C' code for all of these systems is identical, and allows the user to select via one or two "#defines" in file `sga.handexternal.h` whether or not ANSI -style in-line prototype declarations are used, depending on the compiler to be used. The users 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 optimizations should be done, etc., are needed by Borland C++**). Some editing of make files may be needed to accommodate the locations of standard include files, etc., on the user's system.

Beginning with Release 3.0, and ending with Release 3.2, the author is distributing with a PC version of GALOPPS a 'C' compiler, djgpp, which, like GALOPPS, is distributed under the GNU General Public License, and which makes GALOPPS and lilgp very practical to use for fairly large problems on a PC. The djgpp compiler treats all installed PC memory as flat, removing the need for extended memory, etc., and also providing paging transparently to the user, just as if in a Unix environment, but running under DOS. Therefore, the user can run GA problems requiring many MB of RAM without difficulty. However, future releases will not include djgpp, as other compilers now typically also provide access to all of RAM.

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 GALOPPS system. The code has been extended many fold from the original **SGA-C (documented in Appendix Three of this manual)**, 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. Many of the operators, measures, concepts, etc., used here and NOT included in SGA-C are defined in Chapters 1 -4 of Goldberg's book. Ideas and algorithms defined there are NOT, in general, repeated in detail in this guide, so the users should refer to Goldberg's book for answers to questions not found here.

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). (All Russian members of the GARAGE also belong to the ICAD/GA Consortium, but not all members of the ICAD/GA Consortium are funded to conduct research as members of the GARAGE.) The prospective user might also want to consult the **SGA-C documentation included as Appendix Three** 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 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 archives server in the Genetic Algorithms Research and Applications Group (GARAGE), Michigan State University (ftp to isl.msu.edu, look in subdirectory pub/GA for the latest GALOPPS release in both zipped tar format and as a directory of files. Web users may use //isl.msu.edu. For more information, contact the author by email: goodman@egr.msu.edu. The Case Center's Sister Center in Moscow, and Russian headquarters for the GARAGE, is the AI/CAD Laboratory of Moscow State Bauman Technological University, V.L. Uskov, Director (email uskov@aicad.isrir.msk.su). Dr. Uskov can also provide assistance to users or other interested parties in Russia and the CIS. Russian -language versions of GALOPPS are also available from the Russian GARAGE, but are usually on one release delayed from the English -language version.

WHAT ELSE IS COMING OUT WITH GALOPPS?

Graphical User Interface is available in Tcl/Tk for Unix users --see GARAGE website.

Threaded version of GALOPPS 3.01 is available --see GARAGE website.

PVM GALOPPS Release 3.1 continues to be current

DAGA2 --a2 -level GA based on GALOPPS --will be available "shortly".

DAGA2 does ON-LINE competition among multiple subpopulations solving a problem, using different crossover, mutation, selection operators, different representations, different operator probabilities, parameters, etc. It uses PROGRESS in solving the problem as the "fitness function" for each subpopulation, and good combinations of operators, representations, parameters, etc., spread through the set of subpopulations via genetic operations (crossover, mutation, selection) treating subpopulations as individuals. DAGA2 can dynamically track the solution process, changing operators and rates as the problem solution progresses.

RELEASE HISTORY AND BUG FIXES (ESGA/IPGA 2.00 --> GALOPPS 3.2):

GALOPPS 3.2, July 15, 1996

This is a major new release, with several added capabilities, and many minor improvements to I/O, completeness of information reported, and documentation. Primary new capabilities are:

A GRAPHICAL USER INTERFACE for Unix users, implemented in Tcl/Tk, which allows easy, form -fill-in driving of One pop and Many pops, and provides GRAPHICAL output of key variables under user control. It also allows graphical input of migration relationships in Many pops, and writes/reads input files compatible with the "standard", non -graphical user interface. See the GARAGe webpage (<http://isl.msu.edu/GA>) to download it. Full documentation for the GUI is still in preparation.

Allowing user to define different ALPHABET SIZES (cardinalities) for different fields, and automatically treating them correctly (crossover only at field boundaries, mutation only to allowable values), even if subpopulation -level inversion is in use. To use different alphabet sizes, set *alpha_size=1* in the input file, and supply value for array *field_sizes[]* in function *app_set_field_sizes()* in the *appxxxx.c* file. GALOPPS automatically decodes the chromosome properly, and remaps variable length fields under an inversion operation, and transforms migrants into the correct inversion pattern (and field boundaries) for the receiving subpopulation. (Introduced at user request --thanksto Dragan Savic at University of Exeter.)

Introduction of a new MULTI_FIELD MUTATION OPERATOR. Single field (or single-bit, for binary reps) mutation is very ineffective for problems with much "deception", and a spatially correlated mutation operator, which can change several adjacent fields at once, can be helpful. User specifies (in the input file) the frequency of subjecting a chromosome to a multi -field mutation (probability per chromosome) and (in function *app_set_options()*) the expected value of number of fields possibly altered (this mutation changes at RANDOM some fields in a computed range of loci, but some may NOT be altered).

Introduction of BOLTZMANN SCALING for control of selection via a single parameter, *beta*. Setting *beta* to 0 makes all individuals equally likely to be chosen, while large *betas* (3, for example) give almost all probability of selection to the most fit individual. Between the extremes, any desired level of selection pressure can be achieved.

Optional AUTOMATICALLY CONTROLLED SELECTION. The pioneering theoretical work in applying statistical mechanics (or maximum entropy assumption) to GA's (see Shapiro and Pruegel-Bennet, for example) has produced a

method for dynamically controlling selection so as to achieve relatively high rates of improvement of mean fitness without strongly reducing the diversity of the population. A method for control of selection, using dynamically altered Boltzmann scaling, is provided in GALOPPS 3.2.

Introduction to GALOPPS of a specialized representation and operators for manipulating it, the GROUPING GENETICALGORITHM. This GGA representation, developed by Emanuel Falkenauer, greatly enhances performance of a GA for attacking many grouping and grouping-related problems, as are frequently found in scheduling, partitioning, packing, and similar problems. A problem directory has an example of a GGA problem (for bin packing) together with the specialized operators and encoding/decoding routines to use the representation. It should be regarded as a `alpha-test` (experimental) code, but it is included because of interest to some users.

Improved REPORTING of input configuration and internal parameter settings, plus more convenient (and optional) printing of integer fields for non-binary problems.

A THREADED version of GALOPPS, enabling users on multiprocessor systems to speed up the running of either One pop or Many pops runs, using the pvms system, without user intervention.

GALOPPS 3.2 fixes a bug (in 3.0 and beyond) which did not allow the user to specify an output filename without also specifying an input filename. While this may represent an unusual method of usage, it now works correctly.

GALOPPS 3.2 fixes a bug (in 3.0 and beyond) in the demo in `vexample`, restoring code in `appdemoi.c` which was inadvertently deleted and which defeated the second-order selection mechanism which the example was to demonstrate.

GALOPPS 3.01, 3.02, August 1, 1995

This is a minor bug-fix and documentation improvement release. The new checkpoint file routines introduced in 3.0 were found to interfere with EXTENDING of checkpointed One pop runs, and that is now fixed in file `mainone.c`. Release 3.0's makefiles showed an error (on some systems) because file `master.c` was accidentally included in the compilation when "make One pop" is typed. That has been fixed in all of the makefiles in 3.01. A few examples of extending runs have been added to subdirectories `work` and `examples/app`. A comment in many of the app files might have misled the user into thinking the user needed to malloc utility fields used by user's application, so that has been corrected to indicate that GALOPPS automatically mallocs the utility fields, so long as the user specifies a non-zero length (bytes) in routine `GetUtilitySize()` (in user's `appxxxx.c` file). The manual, `guide301.ps`, has been updated to include these changes, and the section on getting started with GALOPPS and writing your own GALOPPS application have been extended and improved. Release 3.02 was a repackaging for PC users, with no extended capabilities.

GALOPPS 3.0 is distributed in a new form, as a directory `galopps3.0`, with several subdirectories. Unix users may load the tarred, compressed file. PC users may use that if they wish, but an (otherwise identical) form is available with the DOS-type end-of-line terminations, which is necessary for some compilers. PC (DOS or Windows) users

who want to run problems requiring more than 640KBRAM should also download the djgpp 'C++' compiler, a PC port of the GNU g++ compiler, providing up to 128MB of "flat" RAM and up to 128MB of paging disk (whatever is available on your machine). Makefiles which are usable by djgpp are also included (in both versions). (See directions for use below.) However, for debugging new applications, the user may want to use his/her "usual" tools. For Borland 'C' users, some example ".prj" files are provided (in the work\ subdirectory and the examples\app\ subdirectory. Before compiling with Borland 'C', be sure to specify under "options" that include files are also located under subdirectory 'galopps3.0\include', and when making a new project, the user must add in the source files from subdirectory 'galopps3.0\src'. Specify the problem file's subdirectory as the place to put compiled objects (output from compiler).

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 part of this code for resolution 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 thru j2int for reading of integer fields from chromosomes were particularly significant errors).

The Beijing 2.01 Release of ESGA/IPGA (February 16, 1994) corrected not only the bugs discovered in the eoriginal SGA-C, but also several bugs in earlier alpha test releases of the code from November, 1993 - January, 1994. It added a print of the random number seed to the program output, generalized example problem app1perm.c, and added an additional example problem. Modules updated included 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 makefiles for the mixed representation files were also modified. The new example file added was called app1both.c.

Release 2.05 of ESGA/IPGA (subsequently renamed GALOPPS 2.05) 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 included: mainpga.c, mixtutor.c, statisti.c, and apprally.c. Also, a new callback ack(app_new_global_best_report()) was added to every app -type file and template. This was 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 in quiet mode.

GALOPPS 2.20 corrected 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. AIII/O file formats changed in 2.20, due to the many additional features introduced (see New Features below).

GALOPPS 2.25 corrected a serious bug in migration among subpopulations. Users using parallel subpopulations at all were urged to go to 2.25, as the 2.20 version had a serious bug in migration of individuals among subpopulations.

GALOPPS 2.30 included several bug fixes to the 2.25 release, including a bug fixed in uniform crossover.

GALOPPS 2.31 differed 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 (One pop) and initsubp.c (Many pops). The user who had already created new application functions just needed to add a blank (dummy) callback to their app.....c file, as illustrated in any of the app files accompanying 2.31.

GALOPPS 2.32 differed from 2.31 only in the rewriting of files appmansq.ctouse automix, and of appmatch.c and appautmx.cto improve their understandability. Their input files were also augmented with comments about the automix runs which should precede them. On Oct. 26, 8pm, Release 2.32 was updated slightly with 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 have been affected by any of these changes, except to see slightly more output than before when "quiet" is set to 1 or 2.

In GALOPPS 2.35, a bug in crossover rate, when crowding is used, was fixed. Several other small problems were remedied, in files mainmany.c, checkwt.c, checkrd.c, generate.c, random.c, sgapure.h, and 24 into 8.mst. Program input was reorganized. The sequence in which inputs are requested, by both OnePop and ManyPops, was 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 the second and later subpopulations, which were not, in fact, used, is no longer necessary. This reorganization allowed 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 cannot be read by Release 2.35.

In GALOPPS 2.35, user -created app files require alteration of only the initialization portions (addition of a parameter to app_init(), and adding of "blank" functions app_user_init_pop() and app_after_random_init() to the app files).

GALOPPS 2.36 fixed a small number of bugs found in 2.35, having to do with uniform crossover evaluation counting, crossover rates when crowding was used, a bad value in an example.mst file, and some irregularities with global parameters under certain restart conditions, which required a small change to the checkpoint header files (.ckp). Therefore, previous checkpoint files are not compatible with those of Release 2.36 and beyond.

GALOPPS 2.37 contained a few bugfixes to Release 2.36, including two to the app_user_init_pop feature added at Release 2.35, one involving file locking when multiple processes or machines are used, one sometimes causing faulty initialization of the global.stt file, and one affecting dynamic resizing of populations. The order of reading checkpoint file prefix and restart file prefix in OnePop has been revised to match that of ManyPops.

Release 3.0 contains many enhancements. File locking has been greatly simplified from the user perspective: only the global statistics file still has a companion locking file, z.....stt. The ckp and ind files are now "self-locking", and multi-population runs on multiple processors may be run without the program creating many z.....lock files. The structure of the command line invoking the program has been enhanced. The (optional) input filename is now preceded by -i, output filename by -o, etc. Other parameters may now be OVERRIDDEN for the run by specifying their values (for all subpops or only one) on the command line (see description of this feature).

The addition of optional inversion for use with all but permutation representations means that if 'perm problem' is false, the input file must contain the probability (per generation) of inversion, as described below under New Features. Multiple subpopulation runs (on ONE processor or on many) now allow each subpopulation to use a different representation for the problem solutions; the user must only supply 'C' code to map from those representations to the "standard" one used in the fitness function, and to map migrants from one representation to any other to which migration will occur. Since number of fields, alpha_size, and choice of random or superuniform initialization may now change from one subpopulation to another (even within the same process), the order of the input information had to be changed, so all .in files were revised to reflect this change (a .txt file in the distribution tells how to update any existing applications you may have written to comply with the new input file structure). Documentation in the code is improved, as is this user guide. The USER, in the app code, must provide any "meta level" code to control the inversion rate, in order, for example, to allow competition among subpopulations using different inversion patterns. A simple example of a method to do that is provided in directory examples/demoinv.

If you have begun using any of the earlier releases, the author urges you to convert to the latest release. The newest release has been tested more thoroughly than any of the earlier releases, some of which were interim development releases only. The capabilities of Release 3.0 and beyond, and their level of system independence and testing, are much better than earlier releases. Conversion from recent releases should involve only MINOR changes in user-written code -- in any app file the user has written. The easiest way to find the changes to be made is probably to diff the appxxxx.c file you used in creating your application with the appxxxx.c file in the new release, and then make corresponding changes in YOUR app file. However, an additional file, changes.txt, documents the changes to be made in a user's app file from release 3.0 to 3.2.

Release 3.01, August 1, 1995, is a minor bug-fix release of 3.0, to remedy a problem introduced when the I/O system was improved. The problem occurred if a user tried to EXTEND (restart) a run done under OnePop, and did not occur neither ordinary runs or using extension of ManyPops runs. An error for compiling OnePop in the makefiles in the example subdirectories and work directory was also corrected (file master.c should NOT be linked for

compiling Onepop, to avoid a multiple definition error). Users not using Onepop need not install Release 3.01, although it does also include three additional example input files illustrating extensions of runs, and a few minor corrections in this user's manual. It also clarifies in the comments in the app?????.c files that the user need NOT malloc utility fields... they are malloc'd automatically by the system, using the length information provided in the user callback GetUtilitySize().

ANOTE ON USER INTERFACES, ETC.:

A NEW GUI is available for Unix users of GALOPPS 3.2. See the GARAGe website (<http://isl.msu.edu/GA/>) for information about and/or to download the new graphical user interface (beta test). A PC version of the GUI (which is written in Tcl/Tk) should be available in Winter, 1997 (depending on release of versions of Tcl/Tk for various Windows, Windows 95, and Windows NT environments). 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). GALOPPS is the ENGINE, which was designed to be easy to embed in a GUI. The GUI simply writes the input files (which can be in a human-readable format for easy development) and displays the results from the output or checkpoint files, for example. Hooks (via app callbacks) are provided for modifying a run in process, etc.

For information about these and related developments, please send email to goodman@egr.msu.edu indicating that you want to be included on the list of GALOPPS users. Alternatively, consult the World Wide Web page of the Genetic Algorithms Research and Applications Group at MSU (<http://isl.cps.msu.edu/GA/>) for progress reporting on their availability, or contact goodman@egr.msu.edu

Please monitor the web page of the MSU Genetic Algorithms Research and Applications Group (MSUGARAGe) for further information.

NEW AND ENHANCED CAPABILITIES OF GALOPPS 3.0 FROM GOLDBERG'S "SIMPLE GENETICAL ALGORITHM" IN BRIEF:

(NOTE: For information on how to get started running GALOPPS, see the next sections, "How to Get Started Running GALOPPS" and "Writing Your Own Application.")

(For more DETAILED descriptions of the new features cited below, see the section "Additional Tools for Traditional Problems" (p.xx)

(For more DETAILED explanation of parallel operation (on one computer or on many), see the sections "Overview of GALOPPS 3.0's Island (Coarse -Grain) Parallel Capabilities," "Files Used in GALOPPS," and "Files Created by GALOPPS System During Execution"). Alternatively, Unix users may see the separate documentation and code for PVM GALOPPS or Threaded GALOPPS (on the GARAG website, <http://isl.msu.edu/GA>).

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 for breeding from among newly generated offspring) rather than **steady-state**. 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. For example, offspring need not replace only their own parents, etc. The users 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, replacement strategies, parameter settings, etc. **The user is counseled NOT to use optional features beyond the Simple GA (for example, crowding, incest reduction, sigma truncation, migration crowding, rank - based selection, etc.) without understanding what they do and whether or not they are appropriate for the problem the user is solving. An inappropriate choice MAY make the solution much more difficult, rather than easier.** GALOPPS does not decide on appropriate parameter settings or operators for a particular configuration -- that is up to the user. The sample applications merely ILLUSTRATE the use of various features, and ARE NOT OPTIMIZED for best performance -- rather, use of various features is sprinkled among sample applications largely based on when the feature and the application were developed and under going test. The author had **no interest in pursuing optimal tuning** of any particular sample application, but rather sought to illustrate ways in which GALOPPS may be used.

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 - Cv1.1 version) include

(*** indicates a capability NEW in Release 3.0 - 3.2. Other items were new in Releases 2.0 - 2.25.)

*** Added a GRAPHICAL USER INTERFACE for Unix users, implemented in TCL/TK, which allows easy, form - fill-in driving of One pop and Many pops, and provides GRAPHICAL output of key variables under user

control. It also allows graphical input of migration relationships in Many pops, and writes/reads input files compatible with the "standard", non-graphical user interface. As soon as users of other systems have tk available on their systems, they may begin using this new GUI.

- *** Allowing user to define different ALPHABET SIZES (cardinalities) for different fields, and automatically treating them correctly (crossover only at field boundaries, mutation only to allowable values), even if subpopulation-level inversion is in use. GALOPPS automatically remaps variable-length fields under an inversion operation, and transforms migrants into the correct inversion pattern (and field boundaries) for the receiving subpopulation. To use different-length fields, simply input alpha_size as 1 (a flag), and specify the desired alphabet size for each field in array field_sizes[], found in function app_set_field_sizes() in all application (app.c, app1.c, appxxxx.c, etc.) files. GALOPPS will handle the fields automatically, even if they are subsequently reshuffled by a subpopulation-wide inversion operation. Migrating individuals are automatically translated into the format of the receiving subpopulation.
- *** Introduction of a new MULTI_FIELD MUTATION OPERATOR. Single field (or single-bit, for binary reps) mutation is very ineffective for problems with any "deception", and as a spatially correlated mutation operator, which can change several adjacent fields at once, can be helpful. Use rs specifies frequency of subjecting a chromosome to a multi-field mutation (probability per chromosome) and the expected value of number of fields possibly altered (this mutation changes at RANDOM fields in a computed range of loci, but some may NOT be altered). Currently, this operator is used by substituting "multimut.c" for "bitmutat.c" in the makefile in the /work or /examples/??? directory. In a future release, all operators will be loaded at compile time, and the user will select among them during herun, so that both sorts of mutation could be in use simultaneously. Once multimut.c is loaded, the user specifies the average width (number of fields) to be mutated at random (value may return to current value) as avg_mut_width, a global in app_set_options() in the user's sappfile. The probability of doing a multi-mutation is specified PER CHROMOSOME, currently as "pmutation" in the normal input file, which multimut.c knows to interpret as a per chromosome, not per field, rate.
- *** Introduction of BOLTZMANN SCALING for control of selection via a single parameter, beta. Setting beta to 0 makes all individuals equally likely to be chosen, while large betas (3, for example) give almost all probability of selection to the most fit individual. Between the extremes, any desired level of selection pressure can be achieved.
- *** Optional AUTOMATICALLY CONTROLLED SELECTION. The pioneering theoretical work in applying statistical mechanics (or maximum entropy assumption) to GA's (see Shapiro and Pruegel-Bennet, for example) has produced a method for dynamically controlling selection so as to achieve relatively high rates of improvement of mean fitness without strongly reducing the diversity of the population. A method for control of selection, using dynamically altered Boltzmann scaling, is provided in GALOPPS 3.2.
- *** Introduction of a specialized representation and operators for manipulating it, the GROUPING GENETIC ALGORITHM. This GGA representation, developed by Emanuel Falkenauer, greatly enhances performance of a GA for attacking many grouping and grouping-related problems, as are frequently found in scheduling, partitioning, packing, and similar problems. Subdirectory /examples/GGA has an example of GGA problems (appbpp.c for bin packing) together with the specialized operators and encoding/decoding routines to use the representation. It should be regarded as an alpha-test (experimental) code, but is included because of interest to some users.
- *** A THREADED version of GALOPPS, enabling users on multiprocessor systems to speed up the running of either One pop or Many pops runs, using the pvms system, without user intervention.
- *** Setting flag different_reps now allows the user to utilize DIFFERENT REPRESENTATIONS of the problem in different subpopulations, by filling in one 'C' callback routine to map a chromosome from an arbitrary subpopulation's rep to the "standard" rep used in the objective (fitness) function, and one routine to map a

chromosome from any subpopulation's rept to any other subpopulation's rept to which individuals will migrate. (Added in Release 3.0.)

***Migration of individuals among subpopulations includes many options, including crowding versus the receiving population, incest reduction from the donor population against the best individual of the receiving population, user-written migration code, and transformation of migrants from one representation to another, allowing the user to use a different representation for each subpopulation, if desired. This capability readily allows for such parallel architectures as the "injection architecture" described in the recent publications of MSU's GARAGE. It allows for increasingly refined representations on lower levels of a tree of subpopulations, and also allows for inversion on the subpopulation level, for example.

***Command line overrides of all input parameters are now allowed, making it easier to conduct many replicated experiments changing only one or a few parameters or random seed between runs, without having to generate a dozen many different input files.

***The preliminary checkpoint files (suffixes .new and .neu) used in earlier releases for locking and synchronization purposes are no longer needed. They are replaced by new .ckp and .ind files which employ alternating buffers, and are read/written with care to preserve file consistency. Multiple -process statistics (.stt) files continue to use the old locking scheme, as they have multiple writers.

***Inversion operators and associated support for inversion of entire subpopulations are provided in 3.0. (See Additional Tools for Traditional Problems for more description).

***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) problems.

Capability for simulation (on one computer) of interacting, "island" parallel subpopulations, with many methods for interchange of migrants..

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

Improved capability for user -supplied initialization of the population.

Optional SHORT form of the "master" file, xxxxxxxx.mst, which specifies for each subpopulation of a Many pops run which types of individuals are to migrate in from which other subpopulation s. If this migration pattern is to be the SAME (relative to the positions of 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.

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

Checkpoint and restart capability. --for one population and for multiple subpopulations. One pop's checkpointing was extended in Release 3.0 to allow checkpointing every k generations, rather than just at the end (maxgen) specified by the user.

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. Then 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 to $\frac{2}{\log_2(\text{population size})}$.

*** A rich 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 in putto:

0 invokes "kid replaces parent" as in SGA;

1 causes kid 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 kid 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 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" in putto 0 (full output), 1 (most outputs 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 Many pops 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 reused for all

subpopulations. Only one random number seed is used, and random numbers continue in this sequence determined by that seed throughout all subpopulations handled by the process.

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

New features are described in more detail in later sections of this manual.

HOW TO GET STARTED RUNNING GALOPPS

NOTE: GALOPPS 3.2 is available in several forms, for the convenience of the user community. For Unix users, it is available in a gzipped, tarred form on the GARAG website, at <http://isl.msu.edu/GA/software>. For PC users, it is available in an ARJ -archived form in DOS -formatted files, and including the DJGPP 'C' compiler for DOS, which can be used to compile it for the PC architecture, making available all memory plus paging disk. DOS users who want to develop code and/or compile in their usual 'C' environment may use their 'C' compiler (Borland C, Microsoft C, etc.) on the DOS files, and may delete the djgpp compiler, if desired. The directions in this guide's section on compiling GALOPPS may be used to compile using any other 'C' compiler. Makefiles are supplied with each application, for those systems which can use them. Some users will need to "tailor" the makefiles for their particular software/hardware environment (changing 'include' file directories, compiler names and options, etc.). Other forms of GALOPPS to be released during summer, 1996, include PVM GALOPPS 3.2 (for Unix only -- superseding Release 3.1); a new, threaded version of GALOPPS (as yet unnamed), and a multi-level GALOPPS-compatible package called DAGA 2. A new Tcl/Tk -based Graphical User Interface (GUI) is available in Unix GALOPPS Release 3.2 on the website as of August 1, 1996.

GETTING STARTED:

0. If unfamiliar with GA's, read Goldberg's book, Chapters 1 -4. It describes the basic structure of the OnePop program. Then read sections of this GALOPPS User Guide? (who, me?) for description of capabilities, etc., to see what you might want to use?
1. Copy all files distributed with the GALOPPS System Release 3.2 for Unix (or the GALOPPS 3.2 for djgpp or for DOS) from its distribution medium into the directory under which you wish to have your directory galopps3.2 made. Directory galopps3.2 will include subdirectories src, include, work, and examples; subdirectory examples will include many subdirectories (one per example). The work subdirectory is where you will create your own applications. ('C' code of the DOS version is nearly identical to the Unix version... only the line -ending characters are system-specific, and the makefiles used instead of gcc instead of cc, and the Tcl/Tk -based GUI code is removed.) (The djgpp release includes the djgpp compiler and a directory for galopps3.2.) Unzip and unpack the routines as usual, if required (per read me files). (Users who will use the djgpp compiler need to make the appropriate modifications to the PC's sautoexec.bat and config.sys files, as described in the file "install.txt" which accompanies the release.)
2. If your compiler does not accept ANSI -style prototype declarations, you must edit file external.handsga.h in directory galopps3.2/include/ to comment out the lines "#define PROTOTYPES_ACCEPTED" and/or "#define SECONDARY_PROTOTYPES_ACCEPTED" as described in the header of external.h. The header(.h) files for GALOPPS are located in subdirectory galopps3.2/include/.
3. Change to directory galopps3.2/src and edit makefile. Edit the compiler options in the makefiles so the CCLine gives the command for invoking your compiler (cc, gcc, etc.). Edit the CFLAGSLines for any desired optimization, etc. (FP coprocessor or not, which processor, debug flag -g or not, etc.) to fit your system. (Note: the version of djgpp distributed with this release of GALOPPS does not compile correctly on some hardware if the -O (optimize) flag is set, so -g should be used, or another distribution of djgpp or some other 32-bit compiler used.) If you want to look at the GALOPPS system's source code lines while debugging your application, turn off optimization and turn ON debug (-g option on some compilers, for example). Exit the makefile and type "make". This will use makefile to compile (but not link) all of the "core" 'C' source

files used by GALOPPS applications. Some compilers complain about not having linked the program THAT IS NORMAL, as linking is not desired as part of the make in/src. --

4. Select a sample problem (objective function definition, etc.) to run (example problems all have filenames which begin with "app" and end in ".c"). The table in Appendix One describes which input files are to be used with which applications, and specifies any auxiliary programs which must first be built and run to create sample files specifying problem parameters and data, etc. (if needed). Information about values to specify when running those auxiliary programs for data creation for the example input files is in comments at the top of the input files or in Appendix One.

5. Change to directory galopps3.2/examples/(whatever_example_you_want)/. Edit the options indicated in the make file there just as above, and, if needed or desired, you may alter the choice of selection method, crossover type, inversion type, and mutation type, to ones also appropriate for the problem at hand (Note you specify an inversion type in the make file even if inversion will not be used, to avoid a linking problem). --

6. Use "make Onepop" to compile and link the code for a single -population run, producing an module Onepop, or use "make Manypops" or simply "make" to create module Manypops. Users of djgpp must perform an additional step: coff2exe Manypops or coff2exe Onepop, which convert the output of djgpp into a .exe file. (Alternatively, the djgpp user may do debugging, etc., using another djgpp utility, run32, which is described in the documentation included with djgpp.) **NOTE: If you will run GALOPPS from the new Graphical User Interface (GUI), you should have downloaded the Unix form of Release 3.2 and should follow the instructions for compiling and invoking the GUI at this point (see file "readme.gui" included with that release). It will help you to create new .mst files for migration, define properly formatted input files, and watch the output as your GALOPPS problem runs.**

7. If running Manypops, you should already have created a (or use an existing) master file, with extension .mst. (GUI users are helped to create it.) 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. general

8. Run the executable, which will be called Onepop or Manypops. (For a listing of command -line options, type, for example, an illegal field, such as Manypopsx, and the program will echo the legal parameters.)

9. Answer the questions as they appear on the screen. (At first, you may want to use sample input files, and to look at the output in a file. See next paragraph.) For details on the meanings of parameters, consult this manual as needed (see section "Explanation of Parameters for a Manypops Run," for example). If you are not EXTENDING a run already completed, you may leave restart file prefix blank (just "return"). If you want to preserve the checkpoint files at the end of the run, it is best to specify a prefix for their names (a checkpoint prefix, up to 6 characters).

(If desired, you may read the input from a file and record the output to a file, by using the "-i input filename" and/or "-o output filename" options on the line invoking the program. See the section on "New Format for Input Files" for detail on input, or simply record the interactive process to guide you in generating an input file. If you specify the parameter names in the file, as recommended (so that a typical line is of the form "p mutation=0.05", for example), then if the program finds some parameter name other than what it expects, it will tell you what it found and what it expected, making it easy to correct the file.

NOTE: In GALOPPS Release 3.0 and beyond, IF YOU ARE READING INPUT PARAMETERS FROM A FILE, you may also specify on the command line the values of parameters in the file to be OVERRIDDEN for this run. GALOPPS will then ignore the corresponding values in the input file, and will use the values given on the command line. The OUTPUT file will properly document the actual values used. For the format of the command line, simply type Onepop -? or Manypops -?, and the format specification will be echoed for you.

The Graphical User Interface for GALOPPS3. 2 eliminate the need to create input files and master files manually, allowing you instead to fill out a form for to specify the input and to make a graph specifying the exchange protocol. Then the GUI creates properly formatted input and master files compatible with the examples included here.

NOTE: the values for GA parameters in the sample input files are NOT suggested settings for you to use. They may not always even be appropriate for the application being run, but are simply choices which are acceptable to the program. Use, for example, the schema theorem, Goldberg's recent results on population sizing, Shapiro's results on selection pressure, your problem-specific knowledge, and testing to determine reasonable settings for the parameters. Regarding communication topology for parallel subpopulations, I'd recommend keeping the migration rates low, balancing premature "contamination" of all subpopulations with non-globally-optimal immigrants (reducing global search effectiveness -- all subpopulation tending to search the same space) against lack of recombination of good suboptimal individuals. Hierarchical or "tree-like" communications, in which some "donor" subpopulations receive NO immigrants, but furnish "uncontaminated" migrants to other subpopulations toward the root of the tree, have been found to be very effective for many problems. Use your intuition and observation of results. **Don't** use inversion unless you see a clear justification for it. If you use inversion, you **MUST** supply in a callback the logic for selecting FOR or AGAINST particular inversion maps... the "base" logic just keeps inverting subpopulations with the specified frequency, regardless of how well any particular map aids the search process. See example demo in v for ideas of how selection over inversion maps might be implemented.

NOTE: GALOPPS' MANYPOPS may be run on ONE or MANY processors (workstations, PC's, etc.). For more detail on running MANYPOPS on MORE THAN ONE PROCESSOR, see the section "Running Many popson Multiple Processors."

WRITING YOUR OWN APPLICATION:

In general, you must first decide how you want to represent each possible solution in the search space on the chromosome. GALOPPS presents two primary alternatives for what the chromosome represents, and operators to go with each choice. The first form is a "permutation problem" or "perm problem" -- i.e., one in which the chromosome consists of *numfields* integer fields, and each integer in the range [0, *numfields*-1] appears exactly once on the chromosome. The second form is an "optimal values" problem, in which the chromosome is a concatenation of a set of integers (which may, in turn, represent discretized real numbers), and each integer field has a finite range. These "optimal values" problems are described below (in a -c), and then the permutation problem is described:

a) (*alpha_size*==2, *permproblem*==n) A string of *numfields* BITS, which the user may interpret in ANY MANNER desired (*alpha_size* is set to 2 for binary, and *numfields* is the number of bits represented. The user may use any subset of the bits to encode any variable desired, and the appropriate operators are one-point crossover, two-point crossover, or uniform crossover, and either bit-wise mutation or a spatially correlated multi-bit mutation operator (multimut.c). Any of the routines --ithruj2int (for several contiguous bits), getfield (for any one bit), or chromtointarray (for all of the bits into an array of ints, one bit per int field) -- may be used to get the values from the chromosome.

b) (*alpha_size*>2, *permproblem*==n) A set of *numfields* equal-length integer fields, each ranging in value from 0 to (*alpha_size* - 1), each of which represents, directly or indirectly, the VALUE of some variable. These may be fetched as ints from the chromosome one at a time (using getfield) or all at once (using chromtointarray). Additional functions in file utility.c (for example, indextofloat()) may be used to convert some or all of the ints into a discretized set of real numbers, for example. For this method -- that is, whenever *alpha_size*>2 -- the crossover operators will only operate at the boundaries BETWEEN fields, and the mutation operators will generate only ints less than *alpha_size*. The same operators as above (oneptx.c, twoptx.c, or unifx.c; and bitmutat.c or multimut.c) are available in GALOPPS for this type of problem.

c) (*alpha_size*<2 (typically 1, but immaterial), *permproblem*==n) (New in GALOPPS3.2) A set of *numfields* not-necessarily-equal-length integer fields, field *i* ranging in value from 0 to (*field_sizes*[*i*] - 1). Each represents, directly or indirectly, the VALUE of some variable. Their respective sizes (cardinalities) are defined by the user in function app_set_field_sizes() in the user's appxxxx.c file. They may be fetched

exactly as when all are the same length, one at a time (using `getfield`) or all at once (using `chromtointarray`). Additional functions in `utility.c` (for example, `indextofloat()`) may be used to convert some or all of the ints in a discretized set of real numbers, for example. For this method -- that is, whenever `alpha_size != 2` -- the crossover operators will only operate at the boundaries BETWEEN fields, and the mutation operators acting on field `i` will generate only ints less than `field_sizes[i]`. The same operators as above (oneptx.c, twoptx.c, or unifix.c; and bitmutat.c or multmut.c) are available in GALOPPS for this type of problem.

d) (`permproblem == y`) Each of the `numfields` ints in the range `[0, numfields-1]` appears exactly ONCE on each chromosome, in any of the `numfields!` possible arrangements. The problem is to find which permutation of these `numfields` fields produces the optimal value of the fitness function. These values are conveniently fetched either one at a time (`getfield`) or all at once (`chromtointarray`). The appropriate operators to use for these permutation -type or reordering problems are: one of `{uobx.c, pmx.c, cx.c, ox.c}` and one of `{swap.c, scramble.c}` (all are described in Goldberg's book or in Davis' *Handbook of Genetic Algorithms*). These operators all generate only legitimate permutations of the original set of numbers `[0, numfields-1]`. NOTE: INVERSION cannot be used for permutation problems, as it is for rearranging chromosomes in which the order of appearance of values on the string can be changed *without affecting the fitness*. Thus, `permproblem == y` defeats inversion (but an inversion operator must still be loaded in the makefile, in order to satisfy the unused call in the code).

ONCE YOU HAVE DECIDED WHAT FORM OF CHROMOSOMAL REPRESENTATION TO USE, YOU SHOULD:

1) Copy the "blank" template `appxxxx.c` in directory `galopps3.2/work` to a new filename. `cof` of your choice (for example, `appmine.c`).

2) Create a new makefile (copying an old one, and inserting the name of the app file to be compiled and linked on the line beginning "APP=") for compiling your problem. In the makefile, choose the type of crossover, mutation, selection, and inversion operators for your problem and the representation you have chosen.

3) Edit your new file, `appmine.c`. The only things you absolutely MUST do, for ANY problem, are to

A) specify in function `objfunc()` HOW THE FITNESS is to be calculated. That is, you must make `objfunc()` into a function which has as domain the chromosome of an individual ("critter" \rightarrow chrom" is a pointer to the chromosome), and as range, its fitness (a field called `critter` \rightarrow `init_fitness` in GALOPPS). In other words, GALOPPS passes to the user, in `objfunc`, a pointer (`critter`) to the structure "individual" which defines the individual to be evaluated. One field of the individual is a pointer to its chromosome. The chromosome is a set of contiguous unsigned integers, the first of which is pointed to by `critter` \rightarrow `chrom`. (`critter` is the local name for the individual in the population whose fitness is now being evaluated.) To define this mapping, the GALOPPS system includes several routines to help you to "decode" the chromosome into its constituent fields (`chromtointarray`, `getfield`, and `ithruj2in t` -- see below). **All of these functions are in a file called `utility.c`, in the `src` directory.** NOTE: If you are using inversion (`pinversion > 0`), you CAN IGNORE that fact in defining the fitness function -- all chromosomes are temporarily "remapped" to the "standard" order used in the fitness function (as defined automatically in the array: `this_pop_map_to_std`) BEFORE they are passed to the fitness function `objfunc()`.

B) Choose the appropriate or desired options for the problem, in function `app_set_options()`. Those include such flags as whether the fitness function is stochastic (different values when called again with same chromosome), whether you are supplying your own code for migration or for initialization of new individuals, whether you will use inversion during the run, whether all subpopulations in a Many pops run use the same representation, etc. If you use variable `-lengthfields`, you must also set `field_sizes[]` in function `app_set_field_lengths[]`.

Briefly speaking, there are 3 "families" of routines in `utility.c` available to help you DECODE the chromosome into the set of variables it represents for use in the fitness function (and any user `-defined` output functions, if needed) --

each routine's "inverse" is also present, in case you also need to "manufacture" chromosomes with particular values. **ALTERNATIVELY**, you may "tell" GALOPPS in your input that the chromosome is just a string of *numfields* bits, and then use your OWN routines called from `objfunc()` to decode the chromosome however you want.

(NOTE: New in GALOPPS 3.2 -- many of these routines also exist in a second form, for use when *alpha_size* < 2 (in this case, the user -specified *field_sizes[]* (in `app_set_field_sizes()`) is used to determine *field_lengths[]* (in bits), and then *field_ends[]*, the ending bit positions of each field on the chromosome). Then, since the user may also be using inversion, or another representation in which the fields appear in different places or have different lengths in different subpopulations, the user must also be able to pass the APPROPRIATE *field_ends[]* array for the subpopulation in which the individual originates to the decoding routines (`getfield`, `chromtointarray`). For that special case (when `different_repsis` TRUE), the users should call, instead, `getfield_uneq_flds()` or `chromtointarray_uneq_flds()`, as illustrated in the example problems `diffreps` and `demoinv`. Those routines accept a *field_ends[]* array (which GALOPPS automatically updates) for each individual who is migrating, and the user provides the `odetotransform` migrants in `app_transform_migrants` whenever `different_repsis` TRUE but `USING_INVERSION` is FALSE (GALOPPS handles the inversion case automatically.)

The three chromosome decoding families "builtin", from the most "atomic" to the most compact, are:

ithruj2int(int i, int j, unsigned int* from): Treats the sequence of bits in bit positions *i* through *j* (where bits are numbered in range [1, *lchrom*) of the chromosome pointed to, as a nonnegative binary integer and puts it into an integer variable. Using this function, the user can "decode" arbitrary binary strings of varying lengths into integer variables, any of which may then be mapped into real (float or double) variables over arbitrary ranges, if desired, using a few lines of 'C' code (or provided functions like `indextofloat()` in file `utility.c`). Using this lowest level decoding function, the user has essentially complete control over the decoding of the chromosome at the bit level. This form of decoding is usually used only with "binary" chromosomes -- i.e., when the input *alpha_size* is set to 2. This routine is used in all three of the examples from Goldberg's book, in subdirectories `app`, `app1`, and `app2`.

Example:

```
int intval;
intval = ithruj2int(1, 10, critter -> chrom);
(This code would take the value in bits 1 - 10 of the chromosome
critter->chrom and put it into intval.)
```

getfield(int n, int nbroffields, unsigned* chrom): The chromosome is always defined as a set of *numfields* fields (binary or larger), whether or not they all have the same length, and this function may be used to fetch the contents of the *n*th field (0 < *n* < *numfields*), with one exception -- see EXCEPTION in next paragraph. The user specifies *numfields* (and *alpha_size*, for non-permutation problems) to GALOPPS as part of the input stream, and then passes to `getfield()` parameters *numfields* and the number of the field wanted, together with the pointer to the chromosome. (Of course, for permutation problems, *numfields* also determines the range (and therefore size) of each field, since the fields must contain all integers in [0, *numfields*-1].) Note that whenever *alpha_size* != 2 or *permproblem* == y(TRUE), GALOPPS crossover operators will break the chromosome only at field boundaries, and mutations will be performed so as to generate only LEGAL values (in range [0, *alpha_size*-1], or, when *alpha_size* < 2, in range [0, *field_sizes[i]*-1]).

Example:

```
int intval;
intval = getfield(7, numfields, critter -> chrom)
(This code will fetch the value in field 7 (of numfields total) from critter -> chrom, placing it into intval.)
```

EXCEPTION: `getfield()` fetches fields using the global array *field_ends* defined for the current population (which is calculated by GALOPPS based on user input). However, if the chromosome to be decoded is coming into this subpopulation from another one or is a saved "best" individual, then the `iflag`

different_reps is set, then the array *field_ends* may not match the array in use when the chromosome was evaluated. To assist with this situation, GALOPPS 3.2 provides two things: 1) it saves the *field_ends* array with the individuals recorded as "bests", and 2) it sends the *field_ends* array for the sending population along with any migrant sent to another subpopulation. However, to use that information, the user must call a substitute for *getfields()*, called *getfields_uneq_fields()*, using the following form:

```
getfields_uneq_fields( n, numfields, critter -> chrom, proper_field_ends );
```

where *proper_field_ends* is replaced by the name of the *field_ends* array in use (passed along) when the chromosome was evaluated. This information, together with the stored *map_to_standard* array prevailing when the chromosome was first evaluated, allows a correct interpretation of the chromosome at a later time or when it enters another subpopulation.

chromtointarray(int* intarray, unsigned* chrom): If the user wants to "unload" the chromosome into a set of int fields, then rather than fetching each field one-at-a-time, the user can provide an array of ints, and *chromtointarray()* will fill the array with the int values coded by the fields on the chromosome. If *alpha_size* == 2, then each int will contain 0 or 1; if *alpha_size* > 2, then the ints will be in the range [0, *alpha_size* - 1]. When *alpha_size* < 2, each field has its own upper limit, *field_sizes[i]*.

Example:

```
intarray[10];
chromtointarray(array, critter -> chrom);
```

(This code places all of the int values of *critter -> chrom* into *array*, converting the chromosome, essentially, from a string of adjacent bit fields within a set of contiguous unsigned ints into an array of int values, which are easily manipulated in calculating the fitness.)

NOTE: There is a corresponding function, *chromtointarray_uneq_fields(array, critter -> chrom, proper_field_ends)* for use when *alpha_size* < 2 and *different_reps* == y is in use. It performs analogously to function *getfields_uneq_fields()* described above.

YOUR_OWN_CODE(): (writing your own decoder, if you prefer)

Tell GALOPPS *alpha_size* is 2 (binary), and the chromosome is *numfields* BITS long. Then you may supply your own code for decoding the chromosome from its set of unsigned ints to your fields, or use any of the three calls above to copy the chromosome into intermediate int structures you define.

ANY OF THE ABOVE FUNCTIONS IS TYPICALLY USED TO "DECODE" THE CHROMOSOME FIRST INTO A SET OF INT VARIABLES. THEN THE USER TRANSFORMS THEM INTO WHATEVER FORM IS NEEDED IN THE FITNESS FUNCTION (INT, FLOAT, BOOL, DOUBLE, OR WHATEVER IS APPROPRIATE).

IT IS USUALLY BEST TO DECLARE ANY VARIABLES YOU WILL NEED TO USE IN CALCULATING FITNESS OR FOR HOLDING OUTPUT INFORMATION TO BE STATIC VARIABLES AT THE TOP OF YOUR APPXXXXX.C FILE, OUTSIDE ANY FUNCTION DEFINITIONS, SO THEY MAY BE USED FROM ANYWHERE WITHIN THAT FILE, BUT CANNOT INTERFERE WITH SIMILARLY NAMED VARIABLES ELSEWHERE IN GALOPPS. SEE 6) BELOW FOR METHOD STORED IN APPLICATION -SPECIFIC PARAMETERS, ETC.)

NOTE on decoding ints to float or double variables: there are multiple ways to do this, and the user may easily bias the resulting float variable, or cause the mid-range value to be unattainable, for example, if care is not given regarding the desired mapping of both endpoints, whether the ranges should have an even or odd number of discrete values, etc. The interested user is referred to the discussion in the GADigest during April, 1996 (approx.) on this topic.

4) Using the decoded variables, calculate the fitness of this chromosome, and place that value in the individual's *critter -> init_fitness*. This is the "raw" or "unscaled" fitness, and GALOPPS will use it to calculate *critter -> fitness*, the "scaled" fitness used in selection, according to the scaling options you select in the input file.

5) Then examine the "control variables" in function *app_set_options()*. There, you select the following:

the expected (average) number of adjacent fields to be set at random when multmut is done on a chromosome (occurs with probability *pmutation* PERCHROMOSOME when multmut.c is linked in the makefile),

whether or not to use elitism (always keep the single best individual found),

whether or not the fitness function is stochastic (or dynamic -- has different values if called more than once with the same genotype) (if so, chromosomes are re-evaluated every generation, to "sample" their average values),

user_supplied_initialization (if 1, YOU must supply code to initialize the population "legally" in routine *app_user_init_pop()* below),

call_app_user_does_migration (if 1, YOU must supply the code to migrate an individual among subpopulations, below, in *app_user_does_migration()*, (using the system -provided version as a guide, presumably)

using_inversion (if 1, the program will use and report the inversion maps, even if the probability of inversion, *pinversion*, is currently set to 0.0; if flag *using_inversion* is left at its default (0) value, it is set automatically if you set *pinversion* > 0.0 at the start of a cycle). Allows YOU to control the inversion process, suspending inversion in a subpopulation by setting its *pinversion* to 0, but still using its current inversion map,

different_reps -- automatically set to 1 if you are using *using_inversion*; if you are not using *using_inversion*, then IF you set *different_reps* to 1, YOU must supply the code to transform individuals, when they migrate, from their representation in the donor population to their representation in the recipient population, in function *app_transform_migrant()* below. In that case, you must also supply code, *in_app_user_transform_to_std()* below, to transform a chromosome from the current subpopulation's representation to the "standard" representation used by the objective function, *objfunc()*.

You may also look through the remainder of the *appxxxxx.c* file to see some of the other capabilities provided to you by use of the callback routines. You may "cleanup" the population after an initialization, watch for certain criteria and then reinitialize or alter subpopulations, output desired information when new global best individuals are found, do printing when *inquiet* > 0 modes, and many other things. Unless your application requires input, output, or tracking of other variables, additional globals, etc., you need not use these other callback functions. If you must do some of your own input, see 6) below. If you want to do some application-specific output, see 7) below. If your application must preserve certain application-specific variables with each individual, see the information on using the utility field in Goldberg's book... it is handled similarly here, but automated somewhat. You simply need to use the callback routines *GetUtilitySize()* and *GetMaxUtilitySize()* in your app file to tell GALOPPS the number of bytes of utility field each individual has, so that it can save them. Then you can "cut up" those bytes however you like (using a struct, or in some other fashion). GALOPPS will malloc them, record and restore them when the population is checkpointed, etc., automatically. (See example files *app2* .*cor* and *proyrd.c* for examples of the use of utility fields). Otherwise, proceed to 8) below.

6) There are several callback functions in your *app.....c* file from which you may do **any input of parameters or data** you require for your application, depending on when you want to read them in and whether they should be read when a run is extended, or recorded with each subpopulation, etc. The most common places to do I/O are in *app_read_prob_params*, *app_data*, and *app_init*. Look at each for information on which you want to use (i.e., when it is called). To keep your input consistent with the rest of the GALOPPS I/O, you may call functions *ffscanf()*, in file *ffscanf.c*, to read one input value per line. You should call *toread* from the stream *infp*, which is automatically redirected to be keyboard or input file, depending on what you specified on the command line invoking the GALOPPS run. See files *mainone.c* or *mainmany.c* for examples of calls to *ffscanf()*. By using it, you are allowed to (optionally) include keywords on input lines, just as is done in GALOPPS example input files. You may use *ffscanf.c* to read floats, ints, doubles, strings, etc., just as you normally would use *fscanf* to do. If you will do

extensive input of tabular data, then reading the data directly from a data file (with filename read using `ffscanf()`, for example) is suggested, as is done in many example GALOPPS problems.

Where to do your input :

app_read_prob_params():

This is called once, at the beginning of any run (whether starting a new run only from input data or extending a previous run from saved checkpoint (restart) files). It is called before the "standard" inputs like *numfields*, *alpha_size*, etc., are read. Therefore, the user cannot use this callback to read information which depends on those values. It is useful for reading in tables of parameters defining the problem, etc.

If `Onepop` is used, this routine is called at the beginning of each RUN (One pop allows multiple runs in one "job.")

User-defined values read here are usually defined as static variables at the top of the application file, so their values, once read in, are available to all the user-supplied routines in the file. Note that they apply to ALL subpopulations in Many pops. So long as they are read in this callback, they will be read even when a run is EXTENDED by initiating GALOPPS and supplying saved checkpoint (restart) files. Therefore, parameter values read here need NOT be written to any checkpoint file for restarting. HOWEVER, if the user wants to SAVE values of any of these variables (if, for example they changed during a run and capture the STATE of a run), they can be written to checkpoint files created by each subpopulation, by including `writes` and `reads` in `app_write_ckp_hdr()` and `app_read_ckp_hdr()`. Then they will be restored by each subpopulation (even DURING a run) at the beginning of each cycle, to their values at the end of the previous cycle FORTHATSUBPOPULATION.

app_data(REVISING):

This is called (perhaps multiple times) from two places in files `initsubp.c` (Many pops) or `startup.c` (One pop): when a run is initiated from the beginning (no restart file prefix is given), OR if the user EXTENDS a previous run, giving a restart file prefix, AND answers 'y' to the question about revising parameters. This callback is called once for EACH subpopulation, unless the user says "y" to the question "all_subpops_use_same_parameters?". In that case, it is called only once at the beginning of a run (or when a run is being EXTENDED).

1) `subpop_d` at a call site when the program is started "from scratch" with flag `REVISING` set to FALSE, telling it that the user just finished supplying the parameters (crossover rate, etc.) for this subpopulation, and the user may set any additional user-defined ones desired.

2) `subpop_data` calls it when a run is being EXTENDED, with flag `REVISING` set to TRUE, telling it that the user has just finished revising the normal parameters (crossover rate, etc.) for this subpopulation, and the user MAY revise any additional ones desired. The usual course, unless user will really change them, is to use the default logic:

```
if (REVISING) return;
```

```
to avoid re-reading user-provided parameter files, etc.
```

HOWEVER, if the user EXTENDS a run, supplying a non-blank restart file prefix, and does NOT choose to revise the GA parameters, then this routine is NOT called, so it is not a good place to do input of variables that are NOT preserved in a checkpoint file (see `app_init` below, or `app_read_prob_params` above for possible places to do the input).

app_init():

This routine is called by `initialize()` (in files `initsubp.c` or `startup.c`) for EACH subpopulation EACH CYCLE, whether a new run is being started, or extended, or just continuing from the restart files, and whether `all_subpops_use_same_parameters` is TRUE or FALSE. It is called AFTER individuals are allocated and after individuals are read from the `indfile` (on a restart), but before random initialization of all individuals (if initial start) or of individuals NOT read (if restarting). If doing I/O that needs to be done only ONCE when GALOPPS is started (from beginning/EXTENDING or restart), then the user may use the "first call" logic in this routine to

execute the input (or other) code only once. Otherwise, it will be done each time EACH SUBPOPULATION is initialized or is reloaded from its checkpointed state in EACH CYCLE.

7) Output may be done under varying circumstances from a number of callback routines in your app file. See those callbacks for descriptions of when they are called. You should probably write most output using file pointer `outfp`, which is the global output file descriptor used by GALOPPS, which is automatically redirected if an output filename is specified on the command line invoking GALOPPS.

8) If you use utility fields to store problem-specific additional information about each individual in the population (see Goldberg's book for description of this capability, and application `royalrd` for an example), you must also fill in the routines `GetUtilitySize()` and `GetMaxUtilitySize()` in your app file, so that the system can malloc and read/write the correct number of bytes for whatever data you are storing in the utility fields.

9) If your problem or approach requires you to perform many operations NOT provided in GALOPPS, or you wish to control the run automatically by monitoring certain parameters, and taking particular actions when particular thresholds are reached, etc., it is EASY to do WITHOUT modifying the GALOPPS code, by using the many available user callback functions in your app file (which you made from `appxxxx.c`, for example). Some examples are provided in the file (commented out), and some sample applications contain examples of the use of some of these callbacks. You may also construct your own operators (crossover, selection, mutation, inversion, etc.) by using the "standard" operators as an example and replacing them with your own file containing an operator with the same calling structure.

10) Compile, link, and run your program, just as you do the example problems.

11) **EXTENDING a GALOPPS run:**

Any GALOPPS run (One pop or Many pops) which has terminated normally, or has completed `ckptfreq` generations (One pop) or at least one cycle (Many pops) may be EXTENDED by restarting GALOPPS and specifying as restart file prefix the prefix used as checkpoint file prefix during the run completed or interrupted. For such a run, the user may restart INTERACTIVELY by typing One pop or Many pops, or may supply a simple input file containing the same information. Command-line parameter settings can be used to override the input values in an input file, but an input file OR interactive inputs must always be provided. The data for the population (or for each subpopulation) is RESTORED from the restart files (`.ckpand.indsuffixes`), and, unless modified new command-line values or by new user-supplied inputs (if `other_changes==yes` in the input file or in interactive dialog), the previous values are used. NOTE: it does not matter when in the cycle a run is interrupted -- GALOPPS will always begin extending the run from the last FULL CYCLE COMPLETED (Many pops) or last set of `ckptfreq` generations completed (One pop).

OVERVIEW OF GALOPPS 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 the 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.

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, whether these semigrants include the best individual and/or some number of randomly selected individuals, an option for guiding the selection of individuals to migrate using incest reduction, and an option for using `migration_crowding` to decide which individual to replace by migrants. (At compile time, the user may also decide instead to perform

themigrationwithuser-suppliedcode,bysettingaflaginapp_set_options()).OnepopisrunONLYwhenonewants 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 **ISLANDPARALLELISM** 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 return 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. This is particularly useful for debugging/testing code which will ultimately be run on multiple processors (see below). It also allows different sets of subpopulations (owned by different processes) to use different compile-time options (operators, selection methods, etc.) -- the user need only retitled different "versions" of Manypop to other names before invoking them. 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 processor or in separate processes. Each process, in the input file it reads, is told WHICH SUBPOPULATIONS (by number) IT is responsible 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 controlled by a simple File Locking protocol (different from releases previous to 3.0), which controls file access and prevents overwriting of information being read by another process. In the event of many consecutive failures to gain access to a file to read something, 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 (whenever a process does not control ALL of the subpopulations in use) by maintaining a file, with suffix .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 .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 the .stt file, 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 .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"withITSworkwheneverTHEYnextcompleteacycle.Thisreportingis"asymptotically correct,"andtheonlyimprovementpossiblewouldbetodothisupdating"moreoften,"buthow oftenitisdoneisunderusercontrol,soshouldnotbeanissue.

The **usermuststarttheprocesswhichsimulatesubpopulation0beforeanyotherprocesses** toallowforproperzeroing(atbeginningofrun)orloading(duringarunEXTENSION)ofthe cumulativestatisticsfile(whichhassuff ix.stt).

3) "TRUE" parallelexecution:

Severalprocessors(computers)eachruntheirrowncopy(orcopies,usingmode2)above,aswell ofGALOPPS/Manypops,eachsimulatingasmanysubpopulationsasdesired. GALOPPS/ManypopsallowssimulationofANYnumberofsubpopulationsonANYnumberof processors(actuallyupto99,withdefaultfilenamingconventions).Again,allcooperating processorsmustshareacommonfiledirectory.TheMASTERfile(suffix.mst)determinesthe neighborsandmigrations,wh ichareconductedasynchronously.Acollectionofprocessorsof differentspeedsmaybeused,withtheonlyeffectbeingthatthenumberofevaluationsdoneto findeachemigrantwilldifferstronglyamongprocessors,unless.theuserchoosesto"balance"that bydoinglessfrequentmigrationfromprocessorswhichrunatslowerspeeds.

Theglobalstatisticsfilelockingmechanismwasdescribedabovein2),andismostimportant whenmultipleprocessorsareutilized.Accounting(for example,numberof evaluationsbeforea new"best"individualwasfound,onlineperformance,etc.)isalsodoneasdescribedin2)above.

Since the parallelism of GALOPPS is accomplished without the need for direct inter-process communication, it is easy to run truly parallel subpopulation even on a number of PC's linked by a typical PC network (Novellnet, LANtastic, Pathworks, Windows NT, etc.), or to run simulated parallel subpopulation 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's.

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 computers sharing a common file system, so long as the word lengths and byte orders (big-endian vs. little-endian) 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 others 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" these arch 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 all that is needed, and that can be achieved.

(NOTE: When GALOPPS (Manypops) is run using more than ONE process, the 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. However, running multiple subpopulations from a SINGLE process still allows reproducing of the same run if the same random number seed is used.)

FILES USED IN GALOPPS

Files involved with the GALOPPS system are described in six places:

- 1) Files CREATED by the GALOPPS System during execution are described in THIS section, below, and in detail in APPENDIX TWO.
- 2) Files used for compiling a version of the GALOPPS 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 GALOPPS System is provided as Appendix One.

Files Created by the GALOPPS System During Execution

NOTEWELL: Since GALOPPS is intended to be portable, it restricts filenames specified by the user OR created by the system to 8 characters, plus a period, plus a 3-letter suffix or extension, for DOS /Windows compatibility. Therefore, even when running on a Unix or other system which permits long filenames, LONGPREFIXES 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 input or output filename 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 (at most) three files:

- a) The output files specified in a " -o " field of the command invoking the Onepop code (if one is given). This name may be selected by the user 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 runtime (or in the input file, if used, in the "ckp file prefix" field, maximum of 8 characters).
- c) The checkpoint individual 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 runtime (or in the input file, if used, in the "ckp file prefix" field). Thus, for example, the user would end up with a pair of files called "c1p322.ckp" and "c1p322.ind", if the checkpoint file prefix "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 the "old" buffer in those files, and the buffer pointer for reading is updated at the END OF EACH CYCLE. Each subpopulation also saves its "state" in a .ckp file, which is similarly divided into two buffers, and the buffer is updated at the END OF THE ENTIRE CYCLE, so that an extension of a run will restore a CONSISTENT set of states, even if the previous run was interrupted in mid-cycle.

Thus, for each subpopulation it is responsible for from the master file, a Many pops process writes .ind and .ckp files. 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 filename prefix for .ind and .ckp files for Many pops runs must be no longer than 6 (SIX) characters, since Many pops adds a 2 -digit subpopulation number to this prefix making the names for the various checkpoint files.

NOTE: When a run is to be EXTENDED from a set of checkpoint files already written, the file prefix used for reloading state and population data is given as the restart file prefix in the input file (or in response to that question, if working interactively, or overridden by the appropriate option on the command line). THAT NAME will be used ONLY for RESTARTING ONCE, and all checkpoint files WRITTEN by the new run will be written to the checkpoint file prefix filenames. The many "internal" restorations of checkpointed subpopulations during a run, AFTER THE INITIAL READING OF THE restart file prefix files, are performed using the checkpoint file prefix filenames. Of course, if the user does not want to PRESERVE the restart file prefix files for later performing OTHER experiments, the user may specify checkpoint file prefix and restart file prefix to be the SAME, and then the restart file prefix files will be overwritten during execution. If the user leaves the restart file prefix 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 checkpoint file prefix is blank, the default "sgackp" is created automatically. On an EXTENSION of a previous run, if the user leaves the checkpoint file prefix empty, the restart file prefix is used as the checkpoint file prefix as well, overwriting the previously saved state. If this is not desired, the user should specify different prefixes.

- B) Whenever more than one process is working on the same problem (i.e., $finishpopnum - startpopnum \neq popno$) Many pops also writes a single file called "xxxxxx99.stt", where xxxxxx is the checkpoint file prefix specified by the user. It contains information about ALL subpopulations in the master file, regardless of what processor or processors are 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). Since multiple processes are in use, this .stt file is LOCKED using an auxiliary lockfile, zxxxxx99.stt, which is created automatically by the program. When a process wants to use file abcdef00.stt, for example, which another processor or processor might also be using, it first checks file zbcdef00.stt, a "companion" file. If the z.... file contains 0, the file is "unlocked." If the file contains another number, the process "backoffs" 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.stt, does any reading or writing it likes, then closes it. It MUST then UNLOCK the file by opening the z.... file and writing a 0 to it.

There is some complexity in the initial creation (or finding in existence) of the companion lockfile, 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. 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.

For obvious reasons, the user must **avoid** specifying any file prefix beginning with "z" to GALOPPS, since files starting with "z" cannot be locked (and will be destroyed) by GALOPPS.

NEW CAPABILITIES OF GALOPPS

NEW FEATURES IN RELEASE 3.2

Graphical User Interface:

Release 3.2 features a tcl/tk-based (hence, currently, only for Unix users) GRAPHICAL USER INTERFACE, which allows easy, form-fill-in driving of Onepop and Manypops, and provides GRAPHICAL output of key variables under user control. It also allows graphical input of migration relationships in Manypops, and writes/reads input files compatible with the "standard", non-graphical user interface. Its use is documented in a separate, brief user guide, GALOPPS GUI User Guide, also available at the GARAGe web site <http://isl.msu.edu/GA>. A version of the GUI for a PC environment is expected for December, 1996.

Differing Alphabet Sizes (Field Lengths):

Release 3.2 allows the user to define different alphabet sizes (cardinalities) for different fields, and automatically treats them correctly (crossover only at field boundaries, mutation only to allowable values), even if subpopulation-level inversion is in use. (Of course, *permproblem* must be 'n' for this to be available.) The user signals the use of these user-defined field lengths by entering *alpha_size* as 1 (or any value < 2) in the input file. The user must then set the field sizes in function `app_set_field_sizes()` in the user's `appxxxx.c` file. Then the user simply uses the same tools as always (`getfields`, `chromtointarray`, for example) to decode the chromosome into integer variables, and then treats these varying-range integers as desired to define the fitness function. The new versions of `getfields`, `putfields`, `chromtointarray`, etc., all use the array `field_ends[]` (calculated by the program from array `field_sizes[]`) to tell them where each field starts and ends, so the user need not worry about it. Thus, for example, two fields may represent int variables with ranges [0,5] and [0,100], respectively, while ten others are used to map to floating point variables discretized to different resolutions (500 values for some, 1000 values for others, etc.). If inversion is in use, GALOPPS automatically remaps variable-length fields under an inversion operation, and transforms migrants into the correct inversion pattern (and field boundaries) for the receiving subpopulation. To use different-length fields, simply input *alpha_size* as 1 (a flag), and specify the desired alphabet size for each field in array `field_sizes[]`, found in function `app_set_field_sizes()` in all application (`app.c`, `app1.c`, `appxxxx.c`, etc.) files. GALOPPS will handle the fields automatically, even if they are subsequently reshuffled by a subpopulation-wide inversion operation. Migrating individuals are automatically translated into the format of the receiving subpopulation.

One complication arises when the user uses *alpha_size* == 1 AND uses different representations in different subpopulations (whether inversion or user-supplied differences). The routines `getfields` or `chromtointarray` for decoding the chromosome (or corresponding ones for encoding a new one, if needed) must be replaced by new ones which can accept a `field_ends[]` array argument as well, since the user may be decoding a chromosome which has a different map than the one for the current subpopulation (due to migration or to changes in a subpop's representation after a "best" individual was saved). Therefore, new routines, `getfields_uneq_fields()` and `chromtointarray_uneq_fields()` are supplied for that case. The user is referred to example subdirectory `diffreps` for an example of their use.

Multiple-Field Mutation Operator:

Release 3.2 includes a new mutation operator capable of making changes in several nearby fields at once. Single field (or single-bit, for binary reps) mutation is very ineffective for problems with much "deception", and a spatially correlated mutation operator, which can change several adjacent fields at once, can be helpful. The user specifies the frequency of subjecting a chromosome to a multi-field mutation (probability per chromosome) in the input file (or interactive input), and the expected value of number of fields possibly altered (this mutation sets to RANDOM values all fields in a computed range of loci, but that means that some may NOT be altered). The parameter governing the expected number of fields to set at random is in the user's `appxxxx.c` file, in `app_set_options()`, and is

called `avg_mut_width`. Its value is the average number of adjacent FIELDS (bits, if binary) to be set at random, legal values, all at once. The new value of some field(s) may be the same as the old value -- in fact, for binary fields, the probability of no change to any one particular locus is 50%. As with other operators, this operator is chosen by linking "bitmutat.c" in the makefile in the /work or /examples/??? directory. It may NOT be used for permutation-type problems *permproblem = 'y'*). In a future release, all operators will be loaded at compile time, and the user will select among them during the run, so that both sorts of mutation could be in use simultaneously. Once `multimut.c` is loaded, the user specifies the average width (number of fields) to be mutated at random (value may return to current value) as `avg_mut_width`, a global in `app_set_optiona()` in the user's app file. The probability of doing a multi-mutation is specified PER CHROMOSOME, currently as "pmutation" in the normal input file, which `multimut.c` knows to interpret as a per chromosome, not per field, rate.

BoltzmannScalingofFitnesses:

Release 3.2 allows the user to specify a *beta* value, the Boltzmann scaling constant, and scaled fitnesses (.fitness terms) are calculated from raw fitnesses (.init_fitness fields, really) using, for individual [i]:

$$.fitness[i] = \exp(beta * .init_fitness[i]) / \text{Sum_over_all_i}(\exp(beta * .init_fitness[i]))$$

Setting *beta* to 0. makes all individuals equally likely to be chosen, while large *betas* (3, for example) give almost all probability of selection to the most fit individual. Between the extremes, any desired level of selection pressure can be achieved. This simple Boltzmann scaling by a constant *beta* is invoked by selecting *autocontrol_selection* to 'n' (off) in the input, and then specifying a numerical value for *beta*.

OptionalAutomaticallyControlledSelection:

The pioneering theoretical work in applying statistical mechanics (or a maximum entropy assumption) to analysis of GA performance (see Shapiro and Pruegel-Bennet, for example) has produced a method aimed at dynamically controlling selection so as to achieve relatively high rates of improvement of mean fitness without strongly reducing the diversity of the population. A method for control of selection, using dynamically altered Boltzmann scaling, is provided in GALOPPS 3.2. This method has not yet been tested for effectiveness on a wide range of problems, but the user may find it interesting to explore. GALOPPS Release 3.2 allows the user to elect "*autocontrol_selection*" in the input stream (file or interactive input). The user must then specify (or use the automatic default for) *beta_s_tilde*. The automatic default is based on the work of Shapiro and Pruegel-Bennett, and is determined by using a recommended point on the graph of the ratio of the std. deviation of fitnesses (across one generation) versus *beta_s*. The value of *beta_s_tilde* is obtained by multiplying *beta_s* by $\sqrt{2 * \log \text{popsize}}$. From *beta_s_tilde*, *beta* is defined by:

$$beta = beta_s_tilde / \sqrt{\text{std. dev. of scaled fitnesses}}.$$

Beta, calculated in this way, tends to keep selection pressure roughly constant as the fitness distribution changes. If the suggested default for *beta_s_tilde* is accepted, Shapiro and Pruegel-Bennett suggest that it may tend to maximize the gain in mean fitness subject to not sharply decreasing the diversity of the population. Thus, if *autocontrol_selection* is y(es), GALOPPS calculates *beta_s_tilde* (based on *popsize*) once, and then rescales *beta* from it at each generation, using the current std. deviation of the scaled fitness distribution (after any migrants have entered the population).

If *autocontrol_selection* is chosen, the selection function in `suselect.c` must be compiled in. That is Stochastic Universal Sampling, which then acts on the "optimally" Boltzmann-scaled values to sample the population quite accurately according to the Boltzmann-scaled probabilities.

Falkenauer'sGroupingGeneticAlgorithmRepresentationandOperators:

Release 3.2 includes as an example an implementation (by Brian Zulawinski of the MSU GARAGe) of Emanuel Falkenauer's GGA (Grouping Genetic Algorithm). This GGA representation greatly enhances performance of a GA for attacking many grouping and grouping-related problems, as are frequently found in scheduling, partitioning, packing, and similar problems. Subdirectory /examples/GGA has an example of a GGA problem (`appbpb.c` for bin packing) together with the specialized operators and encoding/decoding routines to use the representation. It should be regarded as alpha-test (experimental) code, but is included because of interest to some users. Reading of one or more of Falkenauer's papers is suggested before attempting to use the GGA provided here.

Threaded GALOPPS:

A THREADED version of GALOPPS, written and tested on Sun workstations, enables users on multiprocessor systems to speed up the running of either Onepop or Manypops runs, using the pvm system, without user intervention. Available as a separately downloadable module on the GARAGe web site, <http://isl.msu.edu/GA>.

Features Added in Release 3.0 -3.02

Subpopulation-Level Inversion Operator and Related Support:

Release 3.0 included a major addition of capabilities for solution of the most difficult problems using value-based (i.e., NOT permutation-based) representations. Inversion is introduced, with a choice of two inversion operators: the "classical" operator, in which a starting position and a (subsequent) ending position on the chromosome are selected at random, and all fields between them are reversed in order, and a "circular" inversion operator, which treats the chromosome as a ring, and picks start and end positions at random on the ring, and reverses the segment in the positive direction from start to end, including passing "around the end" of the chromosome.

However, unlike the various reordering operators in GALOPPS for use in solving permutation-type problems, when inversion occurs, GALOPPS makes the change in order to ALL the chromosomes in the population (or in a single subpopulation, in the case of Manypops) and tracks the locations of the rearranged fields relative to their position in the original "standard" representation used in the fitness function, so that the inversions may be "undone" before chromosomes are actually evaluated in the fitness function. Thus, the user does NOT need to deal with inverted chromosomes in the fitness function, because the chromosome is transformed temporarily back to "standard" form before being passed to objfunc().

A major difficulty with the inversion operator in the past for most GA applications has been that it is difficult to meaningfully crossover individuals with different inversion patterns. GALOPPS overcomes this by doing inversion as an operation on an entire population or subpopulation. Then all matings are done within the same representation, averting the usual problems. The only potential problem is what to do when individuals in Manypops migrate from one subpopulation to another (which may have a different inversion pattern). GALOPPS handles that by transforming migrating individuals into the inversion ordering used in the receiving subpopulation. This is accomplished by first transforming the migrating chromosome back to "standard" form, using the donor's "map_to_standard", then using the INVERSE of the receiving subpopulation's "map_to_standard" to map the chromosome FROM standard to the receiving subpopulation's current representation.

The prospective user of inversion is CAUTIONED to read and understand thoroughly the inversion operator before deciding to use it. In many cases, use of inversion for simple function-optimization tasks will ONLY SLOW the search, producing no beneficial effects. It is useful ONLY for problems in which there is considerable epistatic (nonlinear) interaction among variables on the chromosome, but the user does NOT know enough about the nature of those interactions to "pre-arrange" the chromosomes so that variables related to each other can be placed close to each other on the chromosome. In that case, inversion may help speed the search, by (through random chance) placing two (or more) variables in close proximity to each other on the chromosome, which allows crossover to exploit their relationship more effectively. NOTE that inversion is a "second-level" operator: performing inversion has NO effect on the fitness of the individuals inverted, but rather affects the probability that various "good" schemata which have a long defining length in the "standard" representation (used by objfunc) will become short schemata in the inverted representation, and thus able to act as "Building Blocks." It is a corollary that the inversion capability provided by GALOPPS (and actually, the process itself, the author postulates) is useful ONLY when inversion patterns are judged at a higher (i.e., subpopulation) level. (Therefore, using inversion with Onepop is not recommended, although it is permitted.) GALOPPS permits this higher-level control (subpopulation-level selection on the inversion map) easily, as shown in the demonstration file, **appdemo1.c**. In that example, the inversion rate is independently controlled for each subpopulation, being changed to 0.0 or a user-input value, based on competition among the subpopulations. That is, the effectiveness of a subpopulation's inversion pattern is judged *vis-à-vis* the effectiveness of other subpopulations' patterns by seeing how well the crossover-driven search progresses with each particular

inversion pattern, and rewarding GOOD inversion patterns with a 0.0 inversion rate (i.e., FIXING the good pattern) so long as the subpopulation continues to outperform the other subpopulations. The user has a great deal of flexibility in how to control this, and only a simple example is shown in the file `apdemoi.c`.

The start and end points for these segments to be inverted by an inversion operator are selected at FIELD BOUNDARIES, not between any two pairs of bits. If `alpha_size == 2`, of course, then the inversion can start and end anywhere, but if `alpha_size = 20`, for example, then the inversions will start and end at 5-bit field boundaries.

If your problem is simple enough that it can be solved within a few hundred generations, then it is UNLIKELY that you will benefit from using inversion. However, for problems with high degrees of nonlinear interaction (epistasis) among variables, and no "natural" arrangement of the variables, then if the problem is long enough to require many hundreds or thousands of generations to solve, the inversion operator MIGHT be useful in achieving success. Please note also that while inversion is usable with either one-point or two-point crossover, it is literally useless (and a waste of time) if uniform crossover is used, since uniform crossover treats fields for inheritance INDEPENDENT of their relative positions on the parent chromosomes. (Many have found that inversion is more useful for solving production-rule-set problems, etc., than for conventional function optimization problems. You should determine its efficacy before settling on the use of inversion for any particular problem.)

Note: If you begin using inversion, and subsequently set `pinversion` (the probability of inversion per generation) to 0., then you MUST ALSO set the global variable "using_inversion" to 1 (TRUE), for example, in `inapp_set_options()` in your `appxxxx.c` file. This will allow the migrations among subpopulations to proceed correctly, while producing no more NEW inversions of the subpopulations. The "using_inversion" variable is automatically set, as is the variable "different_reps," whenever `pinversion > 0.` at the start of a cycle.

Multiple Representations -- Different Rep for Each Subpopulation:

In Release 3.0, a new flag in `inapp_set_options()`, in file `appxxxx.c`, is "different_reps." If TRUE, it means that each subpopulation MAY have a different representation for its chromosomes. That can occur in two cases: the user is using INVERSION, or the user has supplied code (`inapp_transform_migrant`) which can translate any subpopulation's representation into the "standard" representation used in defining the fitness function (`inobjfunc()`) and user has supplied code (`inapp_user_transform_to_std()`) to transform a chromosome from the current subpopulation's representation onto the "standard" representation used by the objective function, `objfunc()`. If the user sets flag "using_inversion" to TRUE, or sets `pinversion > 0.`, then "different_reps" is automatically set by the system. But if the "different_reps" flag is set to TRUE (default is FALSE), and using_inversion is FALSE, this signals many popsthat special user-defined routines, `app_transform_migrant()` and `app_user_transform_to_std()`, must be called whenever an individual to migrate has been selected or the objective function must be called, respectively. During migration in this case, it is the job of the user to use the individual passed in the "donor population" representation to define and return the corresponding individual in the representation of the recipient population. When the fitness function is to be called, the user is first allowed to transform the chromosome's representation from its current form to the "standard" representation used in the fitness function. Migration will involve the user's making modifications to the chromosome and/or the utility fields (if used). Of course, the user may need auxiliary information from both subpopulations to know how to transform the migrating individual, and it is the responsibility of the user to obtain that information in whatever manner is appropriate. The remapped individual returned by `app_transform_migrant` is used by many popst to replace a member of `foldpop[]` in the receiving population, just as if the representation had not been changed. The only feature NOT allowed when `different_reps` is set is incest reduction during migration -- if `migration_incest_reduction > 1`, it is ignored when `different_reps` is TRUE, and individualstomigrate are chosen as if `migration_incest_reduction` were 0.

See the different_reps example (in subdirectory examples) for an illustration of the use of `different_reps`.

The use of different _reps allows the user to easily implement parallel GA architectures such as the MSUGARAGE's "Injection Island" architecture. In that approach, subpopulations are organized in a tree-like manner, with base level subpopulations using a representation which is more highly aggregated than the "leaf" nodes. Migration of individuals occurs ONLY in the direction toward the "leaf" nodes. When an individual is passed to a node with a more refined representation, the any value for a more coarsely aggregated variable in the "donor" subpopulation is simply placed into all the destination variables which correspond to the single donor variable. Then, when the receiving subpopulation does genetic operations, these values (which were forced to be aggregated in the donor subpopulation) may diverge from one another, providing a more detailed representation of the problem, but continuously "seeded" with potential "building blocks" from the coarser level. We have found that this Injection Island parallel GA architecture speeds our search significantly on some very difficult synthesis problems.

Command Line Parameter Overrides:

Release 3.0 introduces optional command line overrides for parameters. Earlier releases allowed only the name of an input file and an output file on the command line. In 3.0, those must be prefixed by "-i" and "-o", respectively, as in:

```
Many pops -i filename -o outfile name
```

However, Release 3.0 also allows many other things to be specified on the command line. Names of an input file, output file, master file prefix, checkpoint file prefix, and/or restart file prefix may be specified on the command line for any run, but **OVERRIDES** for parameter values may only be specified IF the user also specifies an input file on the command line. That is, parameter overrides override those specified in an input file, but may not be used if the user is setting the parameters from the keyboard at runtime. Parameters which may differ in each subpopulation may be specified to override the input file values for ALL subpopulations controlled by this process by giving only the parameter name, an "=", and the new value (for example, p_{cross}=0.4), OR, may be specified for any single population by giving the parameter name, the two digit subpopulation number, "=", and a value (for example, p_{cross}03=0.6, which applies only to subpopulation 6). A full listing of the command line options is given if the user enters, for example "Many pops x" or other illegal specification. The listing produced then is shown below (analogous options for One pop may be obtained by typing One pop x):

```
Usage: Many pops [ -i o c e m <filename|prefix> ] [ -p <tag> = <value> ] ...
```

File names:

- i <filename> file of input parameter values
- o <filename> file for runtime output
- c <file prefix> checkpoint file prefix (6 char max)
- e <file prefix> restart file prefix for extending a run (6 char max)
- m <file prefix> master file prefix for migration patterns (8 char max) (Many pop only)

Global parameters:

- p quiet = <output level, 0 -3>
- p npops = <# of population to be run> (Many pop only)
- p start pop num = <first population for this process/processor, 0 to npops -1> (Many pop only)
- p finish pop num = <last population for this process/processor, 0 to npops -1> (Many pop only)
- p n cycles = <# of cycles to be run> (Many pop only)

Population -specific parameters:

- p p cross = <% probability of crossover>
- p p mutation = <% probability of mutation>
- p p inversion = <% probability of inversion>
- p p crowding_factor = <crowding factor>
- p p pop size = <# of individuals/population>
- p p gens per cycle = <# of generations per cycle>

Add a 2 digit population number to tag to change the value of a parameter.

population -specificparameterforonlysinglepopulation,i.e.,
-ppcross02=0.05willonlyoverrideforpop#2

AllvaluespecifiedonacommandlineSUPERSEDEvaluespecifiedintheinputfiles,andtheLATESTvalue determinedforanyparame-teronthecommandlinesupersedesanyearliervaluesforitgivenonthecommandline.

Additional Selection Methods:

Stochastic Universal Sampling:

Anadditionalformofselection,stochasticuniversalsampling(filesuselect.c)hasbeenadded,andisrecommended oversrselect.candrselect.c.(SeeBaker,Proc.2ndInt'lConf.onGA,pp.14-21.)becauseofitssmallSPREAD aboutthedesiredistributionanditsfreedomfrombias.Itisfastertocomputethanalloftheothermethods,fora serialmachine.GALOPPScontinuestoproviderselect(roulettewheel),srselect(stochasticremaindersampling), andtselect(tournamentselection),aswellasranking(describedbelow).

Linear Ranking, followed by SUS:

Whitley(ref.)describesanddocumentsmanybenefitsofusingfitnessranking,ratherthanrelativefitnesses,asthe basisforselectionforsurvivalandmating.Forproblemsinwhichcontroloftherateofconvergenceisatall difficult,rankingisgenerallymoretractablethanothermethods.Itissimilarinsomewaystotournamentselection,but hasamuchsmallerspread,ofcourse.ItismostusefulwhenGA'singeneralaremostuseful:formultimodal problemsofhighdimensionality.Rankingcanbeusedbycompilinginfilesrselect.castheselectionroutine.

Representing Non-Binary Chromosomes (Alphabet Size > 2) (Note: Release 3.2 even allows different sized alphabets for different fields)

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. That is, 1102 and 1112 are NOT legal values in any field. Crossover, mutation, inversion, and initialization will "respect" these restrictions, allowing crossover to occur only at field boundaries, for example. The user triggers the use of this type of representation by responding `N` to "perm problem?" and any integer > 2 for `alpha_size`. (`Alpha_size=2` provides the usual binary representation.)

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 representations have never used `lchrom`.)

Implementation of this feature involved initialization of the populations (there is a new function called `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. The new inversion operators were developed with the non-binary alphabet capability enabled.

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 to a 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.)

Inversion constitutes a re-arrangement of FIELDS, regardless of the field length. Probability of inversion is expressed as a per-generation probability, since entire subpopulations are inverted at once.

A new example application file illustrating the use of an n-ary 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 $2^{*}m > \text{MAX_UINT}$.

Restructuring and Addition of More Crossover Operators for "Value-Based" Problems

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 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 very well, it does not promote 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.

Six Operators and Other Tools Added for Solution of Order-Based (or Permutation-Type) Problems

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 GALOPPS 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 `projectfile` (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`, `int2ithrujandithruj2int`, 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 (num fields and field length 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" `appxxxx.c` file.

QuietMode,forReducedOutputUnderApp(xxxxx).cControl

A new variable, "quiet", has been added to control output. A user might do very long runs and be interested only in the final result, or in output when a specific event occurs. 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 appxxxx.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 copy right 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 EXTEND (by restarting) the program from the checkpoint file to examine the results and/or continue the run). Periodic output is easily done by using code of the form:

```
if(gen%100)fprintf(.....)
```

for example, in app_quiet_report, producing an output only each 100 generations, if quiet==3.

New Fitness Scaling Methods:

Three forms of scaling have been added: window scaling, sigma truncation, and linear scaling. (Note: Boltzmann scaling has also been added, in Release 3.2, see above.) Selecting window scaling disallows the other two, but if window 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 now window scaling (-1), or may set scaling_window to any integer from 0 to a #define'd 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 (the author intends) 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 scale_mult (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. This allows the user to specify a multiplier of sigma, the standard deviation, which the sigma truncation feature in file statistics use 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:

$$f' = \max(0.0, f - (\bar{f} - \text{sigma_trunc} * \sigma)),$$

where \bar{f} is the mean fitness of the current generation, and σ_{trunc} is the multiple of σ as 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 function `app_set_options()` in the user's `sapp(XXXXX).c` problem definition file, the user may use or not use elitism. If elitism is selected, the generation process is modified to INSURE that AT LEAST ONE copy of the current generation's best fitness individual appears in the next generation's population. If turned off, random chance may fail to allow the best individual to be selected for reproduction unchanged, and chance may cause all copies of the current best individual to be subjected to crossover and/or mutation, perhaps resulting in NO copies of fit in the next generation, in spite of its higher-than-average fitness. Thus, the fitness of the best individual of the 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.

If the user elects elitism, survival of the best individual is also guaranteed by GALOPPS during migration operations (i.e., last copy of best guy is never replaced by a worse individual), and during partial re-initialization of a subpopulation (so long as at least ONE individual is NOT randomly replaced, the best one will be preserved).

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 an σ -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 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 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, An Alternative Method For Replacing Population Members with New Offspring (Can Be Used 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 (indifferent "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 crossover replaces a randomly selected individual from the population ALREADY SELECTED according to relative fitness for reproduction or survival into the next generation. That is, this case does no crowding, but it is simply an alternative strategy for placement of offspring (instead of simply replacing their parents, as in SGA).** 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 UNIFORMRANDOM 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 recalculated as usual. Then, for each child, "crowding_factor" members of the set of tentative survivors are selected (at uniform random), and HAMMINGDISTANCE of each chromosome (i.e., number of DIFFERENTBITS) 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 recalculated 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 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 for 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 users 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, the name mechanism (developed by the author) 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 *incest_reduction* (for example, 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). It is a form of negative assortative mating.

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 template, `appxxxx.c`, within which the user can develop the code needed to solve a particular problem using GALOPPS, has been enriched and extended. The user is given the opportunity to inject problem specific code at many points, without modifying the GALOPPS system. Thus, most applications can be coded simply by modifying the single file `appxxxx.c`. "NOP" model functions for all of the user callback have been precoded in `appxxxx.c`, and any which are not needed may simply be left "as is."

Operator & File Usage Automatically Documented to Output

At Release 2.35, code has been added to each crossover, mutation, and selection file (and inversion, in 3.0) operator file, so that GALOPPS can record for the user which operators were compiled in for the particular run being made. The names of the input file (if used) and of the master file (specifying migration among subpopulations in Many pops) are also printed at the beginning of the run. This is intended to simplify for the user the task of keeping track of the environment in which any run was made.

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.corinit.subp.c`) into `user_supplied_init_pop()` in `appxxxx.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 many desired transformations, replacements, etc., on the population, to complete the initialization.

Option to Count and Reduce Objective Function Calls

For greater efficiency of operation, `generate.c` was modified so that the objective function is normally called ONLY when a chromosome has been subjected to genetic operation (and at initialization, of course). 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.

Migration Capabilities Significantly Enhanced and Multiple Representations Supported

At Release 3.0, significant new features have been added to migration of individuals in Many pops. One level of tournament selection (with tournament size=3) is automatically applied in picking candidates for migration from a subpopulation, unless it is the best individual which is to be migrated. This is to insure that when high crossover and/or mutation rates are used, the many "lethals" produced are not so likely to be selected as immigrants. Note that immigrants from a subpopulation are picked from a population which has NOT yet been subjected to ANY selection since the previous generation's operations were performed. Therefore, a tournament of three for picking each candidate for migration helps to provide a fitness-biased probability of selection for migration. Note that, in the event that migration_incest_reduction is used (see below), EACH CANDIDATE for migration_incest_reduction is ALREADY the winner of a tournament of three based on fitness (and the actual migrant will be chosen based on Hamming distance from the best of the receiving subpopulation).

In addition to providing the above fitness bias for migrants, two new fields have been added to the neighbor lines in the .mst files, which control two new processes added to migration.

"Incest Reduction" During Migration:

The first is called migration_incest_reduction, and is a small integer value (usually [0 -5] individuals). When migration_incest_reduction > 1, then when a random individual (i.e., NOT requested as the donor population's BEST) is to be picked for migration, migration_incest_reduction individuals are randomly chosen (but fitness-biased) by making each the winner of a tournament of 3, see above) from the donor population, and each of their chromosomes is compared with the chromosome of the BEST individual of the recipient population. The one MOST DIFFERENT (largest Hamming distance) is chosen as the immigrant. This may have the desirable effect of reducing the likelihood of "re-importing" the same individual over and over from a neighboring subpopulation, and, in conjunction with crowding, may aid in maintaining diversity and in assembling different building blocks in the recipient population. Note: values of 0 and 1 are equivalent, and effectively disable incest reduction during migration. Note: If the flag different_reps is set (in function app_init, or because inversion is in use), migration_incest_reduction is automatically disabled (treated as 0).

Crowding During Migration:

The second new field is called migration_crowding_amount, and is also a small integer (typically [0 -5] individuals). It works similarly to crowding during normal selection: each immigrant is compared with migration_crowding_amount individuals selected randomly from the recipient population, and the individual whose chromosome is CLOSEST (smallest Hamming distance) to the immigrant is replaced. Depending on other parameters and the problem being solved, this may have the effect of favoring the formation of more than one "cohort" or cluster of individuals with relatively high fitness, but with somewhat distinct genotypes (perhaps representing regions of the solutions space with different local optima). This may aid the GA in combining building blocks, by allowing several distinct ones to persist, at least briefly, before selection eliminates the ones with marginally lower fitness.

User-Supplied Migration Code:

In Release 3.0, a flag variable, call_app_user_does_migration, has been added to the app files, in the app_init() function. If that variable is set to TRUE (default is FALSE), then when migration is to occur, instead of calling the usual migration routines, Many pops calls app_user_does_migration() (in the user's app.....c file), passing it a set of parameters likely to be useful to the user in performing migration. It is then up to the user to do the migration in any manner desired. The net effect should be to move some individuals from neighboring subpopulations (as set in the .mst file) to replace some set of individuals in the receiving population. Migration occurs for each subpopulation at the beginning of its execution in each cycle (except the very first), just after it has been initialized from the restart files, and before any genetic operations are performed. Migrants are introduced into population oldpop[], from which selection of individual stock to create the next generation is later made. It is suggested that a user wanting to use this feature examine carefully the "standard" migration code (in files master.c and checkrd.c) and "cannibalize" whatever portions are applicable. (See also other related (but different) capability for using different representations in

each subpopulation, which may supplant the need for using `call_app_user_does_migration`, by letting the users specify only the transformation code, not the code for selecting and placing migrants.)

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 individual slots 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.

*** At Release 3.0, checkpointing in OnePop has been extended to allow the user to specify an interval (every "ckptfreq" generations) between checkpoint file writing. This new parameter, `ckptfreq`, is read as part of interactive user input, or from an input file, if one is used. (Many pops have always written checkpoint files at the end of each "cycle", and that is not altered.) Checkpoint files continue to be written at the end of the run in any case, so long as termination is normal.

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 `gackp.ckp` and `gackp.ind`. When the program is started, the user is allowed to enter a restart file prefix to `EXTEND` a previous run, 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.

Checkpoint files are written to disk into one of two buffers in each file. At the end of each full cycle, GALOPPS changes the buffer pointer for reading. 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, and to allow consistent EXTENDING of interrupted runs.

SPECIAL NOTE FOR USERS OF THE PVM VERSION OF GALOPPS: See the separate guide to PVM GALOPPS for information on the handling of files in the PVM version, which is different from the "normal" version discussed here.

NEW FORMAT FOR INPUT FILE

LES

GALOPPS Release 2.05 and beyond includes an optional format for input files to GALOPPS. The user may optionally include one 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 a valuable 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 comment lines: `/*Comment text*/` are allowed in the file, but ONLY as stand-alone lines, not as "trailing" comments. -style

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 One pop*/
numberofruns=1
quiet=0
ckptfreq=10
ckptfileprefix =appckp
/*blank restart file prefix means will take input from this file, not*/
/*from a restart file --i.e., NOT EXTENDING a previous run*/
restartfileprefix=
permproblem=n
alpha_size=2
numfields=10
superuniform=n
maxgen=5
popsize=20
printstrings=n
pcross=.5
pmutation=0.01
pinversion=0.
autocontrol_selection=n
beta=
scaling_window= -1
/*Not 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/Many pops 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 subpopulation, in Many pops). If you elect crowding (>0), you may also elect incest reduction. For value `-based` (i.e., NOT permutation) problems, you may elect `pinversion > 0` (otherwise, specify `pinversion=0`). If you choose `autocontrol_selection=y`, then instead of beta, you enter `beta_s_tilde`. In each case, if you omit a parameter, when you run GALOPPS, 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 Many pops 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 `-style`, scaling method, etc.) In that case, the users 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 ONE POP AND MANY POPS

ONE POP INPUT SPECIFICATION

```
/*-----*/
/*appinputtemplate --sampleinputfileforsingle -populationrun(Onepop),*/
/*whichactuallydoesn'trunANYproblem.TheIFlinesDONOTAPPEARINAN*/
/*ACTUALINPUTFILE.Inallcasesinwhichadefaultvalueisshownatrun*/
/*time,ablanklineorblankinthevaluefieldcanbeusedtoindicate*/
/*acceptanceofthedefaultshown.*/
/*-----*/
numberofruns=[1 -?]
quiet=[0 -3]
ckptfreq=[1 -nnn]
checkptfileprefix=(blankorupto6characters,NOTstartingwith'z')
restartfileprefix=(blankorupto6characters,NOTstartingwith'z')
IF(restartfileprefixISBLANK)
permproblem=(y/n)
ANYINPUTFROMinputUSER'Sapp_read_problem_paramsgoeshere.
maxgen=nnn
IF(restartfileprefixNOTBLANK)
other_changes=(y/n)
IF(restartfileprefixISBLANK||othe r_changes)
/*Canspecifyorchangeanyofthevaluesbelow.*/
    alpha_size=nn
    numfields=nn
IF(notpermproblem&&alpha_size==2)
superuniform=(y/n)
popsize=nnn
printstrings=(y/n)
pcross=[0.,1.0]
pmutation=[0.,1.0](NOTE:isprobability/chromosomefor
permproblem==yorifusingmultimut.c,andprobability/fieldotherwise)
autocontrol_selection=(y/n)
IF(autocontrol_selection==y)
beta_s_tilde=[0., ~5.]
ELSE
beta=[0.,~5.]
scaling_window=[ -1 -20](NOTE:mustbe -1IFusingrank -based
selection)

IF(scaling_window<0)
sigma_trunc=[0. ~5.0]
scalemult=0.0or[1.0n ~2.n](NOTE:mustbe>1.0IF
usingrank -basedselection.
crowding_factor=[0 ~5](NOTE:0meanscrowdingoff)
IF(crowding_factor>0)
ince st_reduction=(y/n)
conv_sigma_coeff=[0. ~5.0]
convinterval=[0 ~100]
IF(tournamentselectioniscompiledin)
tourneysize=n
IF(restartfileprefixISBLANK&&popno==0)
USER -DEFINEDINPUTS,IFANY,fromapp_data,app_input,etc.,GOHERE
randomseed=.123
```

IF(numberofruns>1)
 (Thesamefields,beginningafternumberofrunline,gohereforeach
 subsequentrun.)

MANYPOPINPUTSPECIFICATION

```

/*-----*/
/*appinputtemplate --specsforsampleinputfileformulti -populationrun*/
/*(Manypops),whichactuallydoesn'trunANYproblem.TheIFlinesDONOT*/
/*APPEARINACTUALINPUTFILE.Inallcasesinwhichadefaultvalueis*/
/*shownatruntime,ablanklineorblankinthevaluefieldcanbeusedto*/
/*indicateacceptanceofthefaultshown.*/
/*-----*/
npops=nn
startpopnum=nn[0,npops -1]
finishpopnum=nn[startpopnum,npops -1]
ncycles=nnn
quiet=[0 -3]
checkptfileprefix=(blankorupto6characters,NOTstartingwith'z')
restartfileprefix=(blankorupto6characters,NOTstartingwith'z')
masterfileprefix=(blankorupto8characters,NOTstartingwith'z')
/*i.e.,startofnewrunorfirstcycleextendingapreviousrun*/
IF(restartfileprefixNOTBLANK&&cycle==0)
changes_to_this_pop=(y|n)
IF(changes_to_this_pop==y||(restartfileprefixISBLANK
&&(popno==startpopnum||NOTall_subpops_use_same_parameters))
{
/*Canspecifyorchangeanyofthevaluesbelow.*/
alpha_size=nn
numfields=nn
IF(notpermproblem&&alpha_size==2)
superuniform=(y|n)
genspercycle=nn
popsize=nnn
printstrings=(y|n)
pcross=[0.,1.0]
pmutation=[0.,1.0]/*(NOTE:isprobability/chromosomefor
permproblem==yorifusingmultimut.c,andprobability/fieldIFnot
apermproblem)*/
pinversion=[0.,1.0]
autocontrol_selection=(y|n)
IF(autocontrol_selection=='y')
beta_s_tilde=[0.,~5.0]
ELSE
beta=[0.,~5.0]
scaling_window=[ -1 -20]/*(NOTE:mustbe -1IFusingrank -based
selection)*/
IF(scaling_window<0)
sigma_trunc =[0. --5.0]
scalemult=0.0or[1.0n --2.n]/*(NOTE:mustbe>1.0IF
usingrank -basedselection.*/
crowding_factor=[0 --5]/*(NOTE:0meanscrowdingoff)*/

```

```

IF(crowding_factor>0)
incest_reduction=(y/n)
conv_sigma_coeff=[0.      --5.0]
convinterval=[0      --100]
IF(tournamentselectioniscompiledin)
tourneysize=n
IF(userreadsinputduringthisoperationfrom      app_data,app_init)
USER      -DEFINEDINPUTS,IFANY,(inapp_data,app_init,etc.)HERE
randomseed=.123
}
}

```

EXPLANATIONOFINPUTFORAMANYPOPSRUN

Upon starting the GALOPPS/Manypops, if the user started the program interactively (i.e., did not specify an input file on the command line) the user is asked the following questions. If an input filename was given (after `-i` option), program reads the same fields from that file. If an output file is specified (`-o` option), output will be written to that file; otherwise, it is written to the screen. The inputs requested are (NOTE the difference in inputs required below depending on whether the run is a NEW one (restart file prefix is blank) or an EXTENSION of a previous run (restart file prefix is NOT BLANK):

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 (restart 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)).

the file prefix (6 characters or fewer) for the new checkpoint file 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 (6 characters or fewer) for the restart files to initialize each subpopulation (if response is "xyz", then files xyz00.ind, xyz00.ckp, xyz01.ind, xyz01.ckp, ..., (and also xyz99.stt, of running populations from more than one process) 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 files specified in the command line, in the format described above under "New Format for Input Files (or --NOT recommended --as "naked" responses identical to keyboard input).

the file prefix (8 characters or fewer) for the master file (if none is specified, default file prefix "master" is used. If file cannot be opened or read, then the populations are assumed to be independent (no migration).)

IF a new run (not an extension)
 whether or not all subpopulations use the same parameter settings
 the problem type (permutation 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).

IF NOT a permutation problem (permproblem==0)
the size of each field, alpha_size (<2 means different sizes specified in
app_set_options)

the number of fields on the chromosome (numfields). For a binary
representation, this is the number of bits on the chromosome; for
non-binary problems, the number of fields. Not that for non-permutation
problems, the operations of crossover, mutation, etc. are done at field
boundaries only. For permutation problems, the number of fields determines
the set of integers to be permuted on the chromosome.

IF a permutation problem (permproblem==1)
the number of extra fields (>=0; >0 only for "hybrid" reps --0 for most GALOPPS users)

IF not a permutation problem && alpha_size==2
whether to use superuniform initialization or not

whether to complete the run without offering the user any further
opportunity to modify the parameters of the subpopulations (this
question is asked only in the first cycle).

Then, if a restart file prefix was given (EXTENDING a previous run), and all_pops_use_same_parameters==y, then
the run begins. If different subpops used different parameters, then the user enters the GA parameter s for each
subpopulation in turn (from gens per cycle through random seed (first subpop) or conv interval (subsequent subpops),
and the program executes the specified number of cycles. Not 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). Files of "individuals" written by
GALOPPS 3.2, are labeled with the suffix ".ind"
A similar procedure is used to save program state, in .ckp files written at the end of each subpopulation's
execution in the cycle.

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 subpopulation 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 t

facility of GALOPPS/Onepop may be used to initialize runs of Manypops, and vice versa (given the proper choices of filenames).

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.

On MULTIPLE processors, the subpopulations are run with exactly the same sort of intercommunication and execution, but 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 in the section "Running MANYPOPS on Multiple Processors."

Regardless of the mode of execution (one processor 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, UNLESS the "different_reps" flag is set (see that section). 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 $n+1$ (modulonpops, the number of subpopulations). Details follow the example:

NOTE: New, in Release 3.0: migrants which are to be chosen "randomly" (i.e., not picked because they are the current best of subpopulation) are now PRE-SELECTED. For each "random" individual to migrate, three are chosen at uniform random, then tournament-selected (i.e., the best of the three is picked). Thus, the migration process acts as if migrants were chosen from a population which has already undergone some selection. (This is important, because otherwise, offspring of crossovers and mutations in the donor population which have terrible fitness might be picked to migrate -- the new ones have NOT YET been subjected to any selection.) If the user has selected to use migration_incest_reduction, then EACH of the individual to be examined for possible migration is already the winner of a three-way tournament selection.

GENERALFORMATFORAMASTER(.mst)FILE

FULLYGENERALVERSION(ARBITRARYPATTERN)

SAMPLEMASTERFILE:	EXPLANATION:
subcnt=4	Thereare4subpopulations
subpop=02	Subpop0has2neighbors
neighbor=1234	gets,fromsubpop1,2randomlychosenc hrom's,
	migration_incest_reduction=3;migration_crowding_amt=4
neighbor=3234	gets,fromsubpop3,2randomlychosenchrom's,etc.
subpop=11	Subpop1has1neighbor
neighbor=0 -143	Hegets,fromsubpop0,itsonebestchrom
subpop=22	Subpop2has2neighbors
neighbor=3234	gets,fromsubpop3,2randomlychosenchrom's
neighbor=1 -234	gets,fromsubpop1,best+1randomlychosen
chrom	
subpop= 31	Subpop3has1neighbor
neighbor=0233	gets,fromsubpop0,2randomlychosenchrom's

SHORTVERSION(SYMMETRICALPATTERN)

SAMPLEMASTERFILE:	EXPLANATION:
subcnt= -4	Thereare4subpops;" -"meansallusesamepattern
subpop=02	Subpop0has2neighbors
neighbor=1234	gets,fromsubpop1,2randomlychosenchrom's,m.i.r.=3,mig_cr=4
neighbor=3233	gets,fromsubpop3,2randomlychosenchrom's, etc.

GeneralFormatforMasterFile:

subcnt=<numberofsubpops>or -<numberofsubpops>
subpop=<subpopindex><numberofneighbors>
neighbor=<subpopindex><FLAG><migration_incest_reduction><migration_crowding_amount>
etc.

wheresubcnt,ifnegative,meansthattheneighborswillbespecifiedforonlyONESubpopulation,andallotherswill useCORRESPONDINGpatterns(modulonpops,andneighborpatterns"wraparound"sothatSubpopulation "npops"isreallysubpopulation"0",etc.,

FLAGmeans:

ifpositive,thenumberofrandomlychosenindividualstoread fromthespecifiedneighbor'sfile;
ifzero,nochromosomesaretobereadfromthatneighbor;
ifnegative,theBESTchromosomeistoberead,plus|FLAG| -1additional randomlychosenindividuals(so -1 meansbestonly, -2meansbestandoneother,etc.).

Migration_incest_reduction(newin3.0)appliesonlytomigrantstobeRANDOMLYchosen(i.e.,NOTapplicable totheBESTindividualwhenitis specifically requested formigration). Migration_incest_reductionisthenumberof individualspickedatrandom(buteachisthebestofthreecandidateschosenatuniformrandom)fromthedonor subpopulation tocomparewiththecurrentbestindividualofthereceivingsubpopulation,withtheoneFARTHEST inHammingdistancewinningtherighttomigrate.Settingmigration_incest_reductionto0or1effectivelyturnsoff thisoptionalmechanism,andeachmigranttomoveeisselectedatrandom(thebestof3candidates).

Migration_crowding_amount(newin3.0)meansthenumberofindividualsfromthereceivingsubpopulationagainst whichthemigratingchromosomewillbe"crowded",withtheoneCLOSESTinHammingdistancetothemigrant selectedtobereplacedbythemigrant. ThisissimilartoDeJong -stylecrowding, butappliedinthecontextof immigrantsratherthannewlygeneratedoffspring. Settingitto1or0meansthatasingleindividualisselectedat uniformrandomforreplacement.

Forsimplicity,themasterfileisstoredinmemoryasafixed -sizetable,holdingamaximumof50subpopulations. Thisisreadilyalterablebytheuser,ifneeded. Thismakesiteasyfortheusertoimplementdynamicstrategiesfor migration,ifdesired,bychangingthetablebasedonwhatevercriteriatheuserestablishes,fromoneoftheapp callbackroutines.

RUNNINGMANYPOPSONMULTIPLEPROCESSORS

Manypops can be run on one processor or multiple processors. If it is to be run on ONE processor, its input must tell it that, given *npops* total subpopulations, *startpopnum* is 0 and *finishpopnum* is *npops*-1. That tells Manypops that this copy of the program is the ONLY one working on the problem at this time. It turns off the keeping of a global ".stt" file, since all global statistics can be kept in this one copy of Manypops. It disables file locking, since the .stt file was the only one still needing it after Release 3.0. Statistics for each subpopulation and global (best of all subpopulations) statistics are written to the output file by Manypops under user control.

If Manypops is to be run on MULTIPLE processors, then the user can EITHER manually start the multiple copies of Manypops (sharing a common file space and common .mst file), or the user may use either the PVM version or the Threaded version of GALOPPS (see those two releases for directions for their use. They must be downloaded separately from the "standard" GALOPPS 3.2 distribution). Non-Unix users have only the manual option available for using multiple processors (PC's sharing a file server, for example).

MANUALLY STARTING MANYPOPS on MULTIPLE PROCESSORS (PC's or workstations or multiprocessor systems):

1) Prepare, compile, link, and test your application on a SINGLE processor to be sure it is working correctly. It is far easier to debug problems in that environment. Put the linked Manypops code a directory from which it may be started on ALL processors, and in which ALL processors may read and write input, output, and checkpoint files.

2) Prepare separate input files for each processor (actually, every process, if you happen to want to run more than one process on any processor in a Unix system) to be used. They may be identical except for the SUBPOPULATION NUMBERS assigned. Each process must have ONE or MORE of the subpopulations (there are *npops* in total) assigned to it. For example, if running six subpopulations total on two processors, the first copy of Manypops may be given *startpopnum* = 0, *finishpopnum* = 2 and the second Manypops copy given *startpopnum* = 3 and *finishpopnum* = 5. BOTH PROCESSES MUST BE GIVEN THE SAME MASTERFILEPREFIX. That is the mechanism that allows the communication among the two processes.

3) Prepare the COMMAND LINE for starting each Manypops process. Be sure to specify the APPROPRIATE input file name and to specify a DIFFERENT NAME for the output file of each process (or they will try to overwrite each other). While no synchronization is required or maintained among the multiple processors, the process(or) running SUBPOPULATION 0 must be started first, as it is the only one which knows how to create the GLOBAL STATISTICS (.stt) FILE. In order to synchronize as much as possible, it is common practice to prepare the input line to invoke Manypops process on each processor (all except the return key), and to start them nearly together by hitting the return key on the process owning subpop 0, then the others in rapid succession. Then, to whatever extent the processors are similar, and the number of subpops/processor is balanced, they will tend to maintain similar progress in solving the problem.

(NOTE: There is no procedural problem if they run as somewhat different rates. Manypops simply takes any subpopulation's immigrant(s) from the specified neighbor's LAST SAVED checkpoint file, so if some processors are slower, the migrants they provide will simply be less "advanced." However, as migration occurs throughout a structure (at least if all nodes of the .mst file's graph are in some cycle) then slower-calculating subpopulations will

tend to “catch up” because of immigrants from faster-moving ones, although that effect is not perfect, and may allow slower populations to tend toward mimicking the searches of their neighbors if the migrants are able to “take over”. However, even in that case, the diversity of the receiving subpopulation helps to broaden the search around the new, highly-fit immigrants, at least for some time, so the slower processor is doing useful work. The theory for modeling and describing these phenomena is interesting and not yet worked out.)

4) Examine the output file created by each process. Examine the .stt file for global summary information about the best values found by ANY process(or) at any point in time. NOTE: The current version of the new Tcl/Tk GUI works only with a single Manypops process, and does not know about the global .stt file. That will be improved in future releases. However, the user may USE the GUI (as a separate process on each processor) to initiate each Manypops process and to plot its results, so long as each process uses a different output file name (and hence, a different plot file name for the GUI to use).

NEWEXAMPLEPROBLEMFILES

TheSGA -Cv1.1releasefromwhichthissoftwarewasdevelopedcontainedthreeexamplefiles: **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 in developing GALOPPS). For a description of these three applications, see APPENDIX THREE, documentation for the ORIGINALSGA -C, v1.1. NOTE: these examples use unnecessarily difficult means for decoding the chromosome, and app2 unnecessarily uses utility fields, because they were part of the originalSGA -C system, and the more advanced decoding routines (getfield and chromtointarray) were not available inSGA -C.

However, in order to introduce the user to many of the new features of the GALOPPS system, this release contains many additional example files, including app0to9.c (illustrating non-binary alphabet usage), approyrd.c (Holland's Royal Road challenge problem), appbtsp.c (blind traveling salesman problem illustrating permutation problem features), 4 DeJong functions, an inversion demo application, a demo with different representations in different subpopulations, and a "generic" template for new applications, appxxxx.c. **SOME OF THESE APPLICATIONS REQUIRE THAT THE USER FIRST RUN A STANDALONE PROGRAM TO CREATE THE PARAMETER FILE DESCRIBING THE PARTICULAR PROBLEM. IF YOU WISH TO RUN THE SAMPLE INPUT FILES AS PROVIDED, YOU SHOULD LOOK IN APPENDIX ONE 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. (NOTE: the "hybrid" applications are no longer distributed with GALOPPS by default. Any users wanting them may request them from goodman@egr.msu.edu.)** Each NEW 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. Files approyrd.in and approyr8.in are sample input files for use with OnePop and Manypops, respectively, and should be understandable to all users planning to write their own applications using utility fields, user-generated output, etc.

At Release 3.0, additional examples of input files associated with the Royal Road function have been added. These files include a sequence of five, called royrdst1.in, ..., royrdst5.in, a batch file, royalrd.bat (DOS version) or royalrd.sh, Unix version), which invokes them (with some file copying interspersed), and three .mst files, 32isopop.mst, 8to4to2.mst, and 39feedbk.mst, to demonstrate a possible approach to Holland's Royal Road challenge problem. It demonstrates the flexibility of the GALOPPS framework in manipulating multiple populations, but is intended only as an example for the sophisticated user planning to use time-varying architectures for multiple population runs.

The shells script, royalrd.sh, or batch file, royalrd.bat, illustrate one way in which a student used GALOPPS to attack Holland's challenge, using a variety of configurations of subpopulations and migration patterns during a series of runs. It makes use of file sroyrdst1.in, ..., royrdst5.in.

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

File **app0to9.c** illustrates the use of an `-binaryalphabet`. 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 `alln(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 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 `projectfile` (PC systems).

APPDEMOI.C -- A Simple Demonstration Problem Using Inversion

In this "cooked -up" example, the objective function has two components: the first is a penalty for distance of the solution from being a palindrome -- i.e., best fitness is when the value is symmetrical about the center of the chromosome. A second component of fitness is added merely to gain a single global optimum, making the problem harder. It decrements `fitn ess` for distance of the left half of the chromosome from thesequence `0,1,2,3,...,numfields/2`. Thus, it is conceivable that inversion will help crossover find good solutions by rearranging the chromosomes so that the related pairs of fields (equidistant from the center) actually appear next to each other on the chromosome for some period of crossover, so that good pairs, once found, can tend to be preserved under crossover. `Numfields` should be entered as an even number.

APPDIFRP.C -- Demo Problem for Different Representations and Injection Architecture

This simple file illustrates the use of two different representations simultaneously in a parallel GA run. Four subpopulations are researching for an optimum value of the function in a coarse space, and feeding their solutions into a fifth subpopulation, which refines it using a finer `-grained` representation. The problem is simply to minimize the difference of a sum of fields from the number 100.5. Four subpops look for integer values, and only the fifth refines these search more finely. Immigrants are supplemented with 0 fractional parts when they immigrate.

APPBPP.C -- Demo Problem GGA Representation, for Bin Packing

See explanation under New Capabilities section (GGA).

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 elsewhere in the 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 positions 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.

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

GALOPPS offers the user a wider 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 appxxxx.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, dynamical 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 they are recalled, and some of their typical uses are described below.

The standard application template file, appxxxx.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.

```
/*-----*/
/* appxxxx.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"

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 raw 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
app_set_options()
/* This routine is called once when a new run or extended run is started. */
/* It must set a value for all of its global variables, although they may */
/* be modified later (in other app_ functions) during the run. */
{
    avg_mut_width = 3; /* Must set if using multmut.c. Value is expected */
/* number of adjacent fields (bits, if binary) to */
/* set randomly at once. */

    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 */

    elitism = 1; /* Must set to 0 or 1 to determine whether or not */
/* best indiv. is always preserved in next generation */
/* (1 = yes). */
    elitism &= !stochastic; /* Note that elitism is reset to 0 automatically */
/* if user specifies stochastic flag = 1, since the */
/* changing fitness makes it impossible to insure that*/
/* what was once best is still best. */

    user_supplied_initialization = 0; /* Make it TRUE (1) if you will */
/* supply 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. */

    call_app_user_does_migration = 0; /* Flag, if set, allows user full */
/* control of migration, from */
/* app_user_does_migration(), below */

    using_inversion = 0;
/* Set flag to 1 (TRUE) if will use inversion to */
/* alter placement of fields on chromosome, with */
/* remapping of fields to "standard" positions at */
/* each evaluation, before objective function is */
/* called. It is set automatically if pinversion>0, */
/* but user may set manually here if has ever used */
/* inversion, so unmapping, translation, etc., will*/
/* continue to occur, even if pinversion now set */
/* to 0. */

    different_reps = 0 || using_inversion;
/* You set to 1 (TRUE) if will use different reps */
/* in different subpops. If TRUE, and if the */
/* difference is NOT ONLY INVERSION, then YOU MUST*/
/* also supply code to transform ALL of the */
/* migrating individuals, in app_transform_migrant*/
/* in the skeleton below, and code below in */
/* app_transform_chrom_to_std to transform a */
/* chromosome into its STANDARD representation */

```

```

        /* (used in the objective function objfunc(). */
        /* Always 1 if inversion is being used. */

if (different_reps && !using_inversion) {
    /* User is using different representations in various subpopulations, */
    /* so must be supplying app_transform_chrom_to_std() and */
    /* app_transform_migrant(). User must also set the variables */
    /* max_chromsize_among_all_subpops and max_numfields_among_all_subpops */
    /* in the lines directly below, and delete the print and the call to */
    /* exit() which are inserted as a reminder to the user. */
    fprintf(stderr, "\n*** You have selected different_reps and are not ");
    fprintf(stderr, "using inversion, \nso you MUST specify here in app_init");
    fprintf(stderr, " the maximum sizes of chromsize and numfields.***\n");
    exit(-9);
}
else {
    max_chromsize_among_all_subpops = -1; /* set to illegal value, as flag */
    max_numfields_among_all_subpops = -1; /* set to illegal value, as flag */
}
}

void
app_set_field_sizes()
/* Routine called ONLY IF user inputs alpha_size as < 2, meaning fields have */
/* different lengths. In that case, user MUST provide code here to set the */
/* array field_sizes[i], 0<=i<numfields, to the number of different codes */
/* possible in the ith field (codes will vary from 0 to field_sizes[i] - 1). */
/* Data may be set in executable statements, or via initializers. If the */
/* problem uses different representations in different subpopulations, and */
/* if that affects the field sizes, then this code must set the proper sizes */
/* for each subpopulation, using popno to tell which subpop is being */
/* initialized. Elsewhere, the program will also automatically use the */
/* field_sizes to set array field_ends, to the final bit number (in */
/* [1, lchrom]) of each field. */
{
    int i;

    /*      field_sizes[0] = 24;  Replace with code to set them to the sizes */
        /* you actually want. */
        /* ... */

    /*      field_sizes[9] = 24;  Replace with code to set them to the sizes */
        /* you actually want. */
}

void
app_read_prob_params()
/* Application dependent data input, called ONCE at start of run (whether */
/* started from input parameters or EXTENDED from restart files) by main */
/* program BEFORE it reads the "standard" fields like numfields, etc. */
/* (Note: Onepop may do many RUNS, and this is called at start of each). */
/* Values read here are usually stored in static variables */
/* declared at the top of the app file, so they may be used in any of the */
/* user callback routines, but do NOT have to be read/written to the */
/* checkpoint files. Thus, user should read these values ALSO when a run is */
/* restarted from saved checkpoint files. If values MUST be preserved ACROSS */
/* runs (i.e., saved when checkpointing and reread when restarting) then */
/* that must be done in app_write_ckp_hdr() and app_read_ckp_hdr(), below, */
/* and the variables will be saved with EACH subpopulation. Note: values */
/* like numfields, alpha_size, etc., are NOT yet known when this is called. */
{
}

```

```

void
app_data(REVISING)
BOOL REVISING;
/* Application dependent data input, called by initdata() and revisedata(), */
/* just after user answers questions about crossover rate, scaling, */
/* convinterval, etc. Ask input questions here which might be different for */
/* each subpopulation, and write (and read) the values to the checkpoint */
/* header file by putting code into app_write_ckp_hdr() and app_read_ckp_hdr()*/
/* below. Values like numfields, alpha_size, etc., are already known when */
/* this routine is called. */
/* NOTE: app_data is called whenever the user is initializing (from user */
/* input, flag REVISING == FALSE) or revising (flag REVISING == TRUE) input */
/* data (but only once if flag all_subpops_use_same_parameters is TRUE). */
/* HOWEVER, IF user is RESTARTING from saved checkpoint files, and does NOT */
/* ask to revise any parameters, this routing IS NOT CALLED. Therefore, for */
/* user data (tables, etc.) which you want to read in on a RESTART, also, */
/* use app_init to read those data, as it is called whether restarting or */
/* starting from the beginning. (Put it inside the "firstcall" logic.) */
{
    if (REVISING)
        return;
}

void
application()
/* this routine should contain any application-dependent computations */
/* that should be performed before each generation. called by main() */
{
}

void
app_init()
/* Application dependent initialization routine called by initialize() for */
/* EACH subpopulation each cycle, whether started from input or restarted */
/* from a restartfile, and whether all_subpops_use_same_parameters is TRUE */
/* or FALSE. Called after individuals are malloc'd and after individuals */
/* are read from .ind file (on a restart), but before random initialization*/
/* of all individuals (if initial start) or of individuals NOT read (if */
/* restarting). */
{
    static int firstcall = 1;

    /* Any code you want to add to app_init goes below. */

    if (firstcall) {
        firstcall = 0;
        /* Put here any code you want to do just once in a run, regardless of */
        /* how run was started. */

        return;
    }

    /* Put here any code you want run for each subpopulation at the beginning */
    /* of its execution in each cycle. */

    return;
}

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. */
{
}

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

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 */
/* 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.
 * User can do 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");

    /* Logic below starts printing 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). It is called */
/* after the LAST GENERATION of the cycle of EACH subpopulation. That means */
/* that some things MAY have changed (inversion pattern, etc.) in some */
/* generations BETWEEN when this best individual was found and now, when it */
/* is to be printed, so user must be aware of that in writing outputs. Also */
/* called in checkhdr.c to print the same values (local_bestfit.xxx) when */
/* that local best turns out also to be the best among all pops, all */
/* processes, so is EQUIVALENT at that time to printing all_pops_bestfit.xxx) */
/* It should print out any app-dependent fields user will want to see. */
/* (This call is not made at all from Onepop -- single-population 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() and initreport() */
/* if flag printstrings == 1, just after it has finished printing the */
/* chromosomes in "standard" binary format. (Using this is OPTIONAL.) */
{
}

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

BOOL
app_user_does_migration(IndArray, filename, number_to_migrate, want_best
                        , migration_crowding_amount, ChromSize
                        , migration_incest_reduction, BestInd
                        , migrant_holder)
/* In this routine, if the user has set the global variable (in app_init) */
/* call_app_user_does_migration, and user is NOT using inversion, then the */
/* user HERE is allowed to perform migration among subpopulations in any */
/* manner the user chooses. ChromSize is the number of unsigneds in the */
/* chromosome of the DONOR population, and global chromsize is the number of */
/* unsigneds in the chromosome of the RECEIVING population (in IndArray). */
/* IndArray is the receiving population (the global variable is oldpop), */
/* and transfer occurs at the start of a cycle); the index of the best */
/* individual in oldpop (in internal form, without one added) is in BestInd. */
/* For guidance in performing the required operations, see routine */
/* GetMigrants() in file checkrd.c, where the system-provided migration */
/* capabilities (migration_incest_reduction, migration_crowding, inversion, */
/* etc., are all done. */
struct individual *IndArray;
char *filename;
int number_to_migrate;
BOOL want_best;
int migration_crowding_amount, ChromSize, migration_incest_reduction;
int BestInd;
struct individual * migrant_holder;
{
    return TRUE;
}

```

```

BOOL
app_transform_chrom_to_std(mapped_saved_chromosome, chrom_for_objfunc)
/* If the user is using different representations for different subpops, */
/* but is NOT using the system-provided inversion code, here is where the */
/* user must supply the code to transform a migrant individual's chromosome */
/* (in mapped_saved_chromosome) from its "mapped" representation in the */
/* subpopulation to the STANDARD representation, which is the one the user */
/* uses in defining the objective (or fitness) function, objfunc. The */
/* "standard rep" chromosome must be returned in chrom_for_objfunc, which is */
/* a local name for critter->chrom, the chromosome passed to objfunc(). */
/* The user can use any of the global variables, or local variables defined */
/* at the top of this file, to determine which supopulation this is (popno), */
/* and what characteristics THIS representation may have (local variables, */
/* utility fields, etc.) */

unsigned *mapped_saved_chromosome;
unsigned *chrom_for_objfunc;
{
    int ii;

    /* The commented-out code below does the "identity" transformation on the */
    /* chromosome -- i.e., copies one to the other. If the user is doing some */
    /* transformation OTHER than the system-provided inversion, user */
    /* should change the first loop below to perform the actual */
    /* transformation process. Finally, user must make the function return */
    /* TRUE if successful. */

    /* NOTE: Next loop must be modified if subpops have different chromosome */
    /* representations -- user must do a transformation, not a copy. */
    /* for (ii=0;ii<chromsize;ii++) {
        chrom_for_objfunc[ii] = mapped_saved_chromosome[ii];
    }
    return TRUE;
*/
    return FALSE;
}

BOOL
app_transform_migrant(migrant_holder, DonorChromSize, newguy, filename)
/* If the user is using different representations for different subpops, */
/* but is NOT using the system-provided inversion code, here is where the */
/* user must supply the code to transform a migrant individual (in a struct */
/* individual pointed to by migrant_holder) from its donor pop rep to its */
/* popno (a global, the current "receiving" population) representation, */
/* returning the transformed guy in newguy, which has a chromosome length of */
/* RecipientChromSize unsigneds. If user is using inversion, not user's */
/* "own" transformation, then this routine is not called, so may be left in */
/* its "commented out" form, which is the the IDENTITY transformation (i.e., */
/* copy the individual unchanged from migrant_holder to newguy. */

/* A pointer to the individual to migrate, as read from donorpop, and its */
/* DonorChromSize, are provided, as is the filename of the recipient pop. */
/* After transforming the individual, the user should place it into new_guy */
/* and return TRUE if successful. */
struct individual *migrant_holder, *newguy;
int DonorChromSize;
char *filename;
{
    int ii;
    /* The commented-out code below does the "identity" transformation. If user */
    /* is doing some transformation OTHER than the system-provided inversion, user */
    /* should change the first loop below to perform the actual transformation */
    /* process. Finally, make the function return TRUE if successful. */

    /* NOTE: limit chromsize in next loop below must be revised if subpops have */
    /* different chromosome sizes... user must do a transformation, not a copy. */
    /* for (ii=0;ii<chromsize;ii++) {
        newguy->chrom[ii] = migrant_holder->chrom[ii];
    }
}

```

```

    newguy->parent[0] = migrant_holder->parent[0];
    newguy->parent[1] = migrant_holder->parent[1];
    newguy->xsite[0] = migrant_holder->xsite[0];
    newguy->xsite[1] = migrant_holder->xsite[1];
    newguy->fitness = migrant_holder->fitness;
    newguy->init_fitness = migrant_holder->init_fitness;
    newguy->neval = migrant_holder->neval;
    sys_util_copy_utility(newguy->utility, migrant_holder->utility);

    return TRUE;
*/
    return FALSE;
}

void
app_generate()
/* Opportunity to do any desired (application-dependent) operations at the */
/* end of each generation of each subpopulation. */
{
}

void
app_stats(pop)
/* Application-dependent statistics calculations called by statistics() at */
/* the end of each generation. */
struct individual *pop;
{
    /* See approyrd.c for an example. */
}

void
app_before_inversion()
/* Allow user to perform any desired application-specific actions before */
/* calling the inversion routine. Only called when an inversion WILL be */
/* performed on the current subpopulation at the current generation. */
{
}

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_ckp_hdr(fp)
FILE *fp;

/* Application-dependent callback routine user may use for reading back */
/* from the checkpoint header file any fields added to the checkpoint */
/* header by the user. */
{
    /* 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 to malloc 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;
}

int
GetMaxUtilitySize()
/* User must here return 0 (utility fields not used) or GetUtilitySize */
/* (if different_reps are NOT used), or the size (bytes) of the LARGEST */
/* utility field used by ANY subpopulation, if different_reps includes use */
/* of utility fields of different sizes. */
{
    int nbytes;

    nbytes = GetUtilitySize();
    if (nbytes) {
        if (different_reps)
            /* Here, return size of largest utility field used by any rep. */
            /* nbytes = xxxxxx; */
            return nbytes;
        else
            return nbytes; /* same as GetUtilitySize if !different_reps */
    }
    return nbytes;
}

```

CODEDISTRIBUTIONFORMAT

The GALOPP System Release 3.2 is being released in two formats -- Unix and DOS/Windows (with identical 'C' code, but different compilation files, text file formats, etc.) (the PVM distribution and the multi-THREADED distribution have different code, and are described in their own guides -- see the WWW page):

DOS Format

GALOPPS 3.2 is bundled with the djgpp 'C' compiler for DOS: Disk or directory contains djgpp, and below it, a subdirectory containing galopps3.2. Galopps3.2 has a subdirectory structure as follows:

```

c:\djgpp\galopps3.2
  \src      .c files comprising "core" of galopps3.2
  \include .h files included in all files in galopps system
  \docs     documentation for galopps system
  \work     directory for developing new applications
  \GUI      (Not yet available for DOS, but forthcoming; available now for Unix only)
  \examples
    \app           Goldberg's first example problem
    \app1          Goldberg's second example problem
    \app2          Goldberg's third example problem
    \btsp          Blind Traveling Salesperson Problem
    \royalrd       John Holland's Royal Road Challenge Problem
    \demoalph      Demo of a problem with non-binary representation
    \demoinv       Demo of a problem using inversion
    \diffreps      Demo of a problem using different reps (injection architecture)
    \dejongfn      Four of Ken DeJong's five benchmark functions
    \gga           Bin Packing example and code for Falkenauer's Grouping Genetic Algorithm
                  representation and operators.

```

\src contains the "core" .c files for galopps3.2; \include contains the .h header files. \work and the subdirectories under \examples contain example application ".c", input, master, etc., files. All of the text files in this directory are in "DOS standard file format." Djgpp users must use the makefile in /src first to compile the "core" routines, then switch to an example directory and use the makefile there to create a Manypops or Onepop; then they must run "coff2exe Manypops", for example, to create the executable file Manypops.exe. Users may need to perform minor editing on the makefiles, to supply the correct compiler name (cc, gcc, CC, etc.), for example.

Unix Format:

```

/galopps3.2
  /src      .c files comprising "core" of galopps3.2
  /include  .h files included in all files in galopps system
  /docs     documentation for galopps system
  /work     directory for developing new applications
  /GUI      (Not yet available for DOS, but forthcoming; available now for Unix only)
  /examples
    /app          Goldberg's first example problem
    /app1         Goldberg's second example problem
    /app2         Goldberg's third example problem
    /btsp        Blind Traveling Salesperson Problem
    /royalrd     John Holland's Royal Road Challenge Problem
    /demoalph    Demo of a problem with non-binary representation
    /demoinv     Demo of a problem using inversion
    /diffreps    Demo of a problem using different reps (injection architecture)
    /dejongfn    Four of Ken DeJong's five benchmark functions
    /gga         Bin Packing example and code for Falkenauer's Grouping Genetic Algorithm
                  representation and operators.

```

A main directory galopps3.2 contains 6 subdirectories: src, include, docs, GUI, work, and examples. The src subdirectory contains all of the GALOPPS3.2 files which are NOT application-specific. It must be compiled before any application can be run. To compile, enter the src subdirectory and type "make". The makefile contains directives to generate all of the .o modules. You should first edit makefile to select the compiler options you want (for example, -g for debugger-compatible code, -xO2 for a high level of optimization on SPARCworks compiler, etc.) This make produces no executable (no linking is done), but generates all but the application-specific ".o" files. Subdirectory include contains the header (.h) files common to all applications. Subdirectory work contains a "blank" application template, appxxxx.c, which users "customize" to program their own applications. Subdirectory GUI contains the Tcl/Tk code for the new (optional) Graphical User Interface, which has been tested to date only on Sun Solaris systems. Subdirectory examples contains many other subdirectories, of example applications. In each subdirectory is one or more .c files (which are derived from appxxxx.c), and example input files, examples of any datasets needed by the application (or programs to generate them), .mst files describing migration patterns for the parallel example, etc. To run any application, the user should enter its directory, type "make" or "make Manypops" to prepare for a parallel GA run, or "make Onepop" to prepare for a single-population run. Then typing "Onepop" or "Manypops" will invoke the GA in interactive mode. Typing "Onepop -i abcdef.in" or "Manypops -i abcdef.in" will run the GA with inputs taken from the file abcdef.in, and typing "Onepop -i abcdef.in -o abcdef.out" or "Manypops -i abcdef.in -o abcdef.out" will run the GA with inputs from the first file and all but error output going to file abcdef.out. See earlier sections for full description of command line parameter overrides, etc.

When using your OWN application file (derived from appxxxx.c in /galopps3.2/work), you need to edit the Makefile so the "APP=" line lists your .c files for this application.

The nature and use of all files is explained in Appendix One: Auxiliary Files.

GALOPPS/Onepopand/Manypopshare 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 Manypop), plus filemaster.c only for Manypop. The two programs use exactly the same application files (app.c, app1.c, app2.c, approyrd.c, appbtsp.c, app0to9.c, appxxxx.c, apprally.c, app1perm.c, app1posn.c, app1both.c, appmatch.c, etc.). They use identical formats for checkpoint and restart files.

When editing any of the files distributed with the GALOPPS system, your TAB characters 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' coder runs unmodified on Unix systems (tested on Sun4, HP9000/7XX, DEC station, and NeXT systems), on PC's (tested under MSDOS 5.0 and 6.0 and under Windows 3.1, with djgpp; 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 ga.h and external.h, they will do full ANSI-style prototype checking, using prototypes in file sgafunc.h. If your compiler object to these prototypes, comment out the line #define PROTOTYPES_ACCEPTED at the top of the ga.h and external.h files, and then the older-style declarations in sgapure.h will be used instead. In that case, use extra caution that the types of all arguments in any functions you create for calling from the appxxxx.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 (other than with djgpp, which can use the Unix makefiles), you may simply follow the directions given below to compile and link the necessary modules with whatever tools you are familiar.

How to Prepare the GALOPPS 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 inside a copy of the appxxxx.c file. 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 future releases of GALOPPS.
- 2) add new operators, selection methods, etc., 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. Upgrading 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 application (in the `examples` subdirectory on Unix release and for `djgpp` users under DOS. 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.

You should first copy all files (with their directory structure) from the distribution diskette or archive file downloaded from a web or ftp site to a directory `GALOPPS3.2` (or `djgpp`, for the DOS distribution) 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. If you are using `djgpp`, you will be able to make use of all RAM on your system. If you are using Borland C, Microsoft C, etc., 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 `appxxxx.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 `appxxxx.c`, called `appmine.c` (for example), and then edit the makefile in the `/work` directory to set `APP=appmine.c`, for example.

Similar to 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 un-comment the appropriate line in the makefile. Files `select.cis` roulette wheel selection; `srselect.cis` stochastic remainder selection, `suselect.cis` stochastic uniform sampling (usually recommended over the first two), `tselect.cis` tournament selection, and `rnkslect.cis` rank-based selection. Similarly, un-commenting the appropriate line chooses among the various crossover operators (`oneptx.c`, `twoptx.c`, `unifx.c`, `uobx.c`, `pmx.c`, `cx.c`, and `odox.c`) and the various mutation operators (`bitmutat.c`, `multimut.c`, `swap.c`, and `scramble.c`). You must use care to be sure that if your problem is a permutation problem, you use only the legal permutation operators (`uobx`, `cx`, `ox`, `pmx` for crossover; `swap` or `scramble` for mutation), and if not, that you use the other operators (`oneptx`, `twoptx`, `unifx` for crossover; `bitmutat` or `multimut` for mutation).

Modules to Compile

If an ANSI-compatible 'C' compiler is available, the code can be compiled "asis". 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.

The files to compile and link for ALL types of problems are the following (they are in the subdirectory `galopps3.2/src`):

<code>checkhdr.c</code>	<code>checkrd.c</code>	<code>checkwt.c</code>	<code>ffscanf.c</code>
<code>filestat.c</code>	<code>generate.c</code>	<code>memory.c</code>	<code>random.c</code>
<code>report.c</code>	<code>statisti.c</code>	<code>utility.c</code>	<code>user_in.c</code>

Ingalopps3.2/includemustbethethe#includefiles:

sga.h(#includedinmainone.cormainmany.c)
external.h(#includedinmostother.cfiles)
sgafunc.h(#includedinsga.handexternal.h)
sgapure.h(#includedinsga.handexternal.h)

OnemustalsochooseEXACTLYONEAPPLICATIONFILE --inasubdirectoryUNDER galopps3.2/examples:

app.c	app1.c	app2.c	appbpp.c
approyrd.c	app0to9.c	appbtsp.c	
appdemoi.c	appxxxx.c	appdjf1.c(etc.)	
appdifrp.c	(oryourproblem,madebymodifyingtheappxxxx.cformsupplied indirectorygalopps3.2/work).		

plusachoiceofEXACTLYONESELECTIONMETHOD(inthefileintheproblem'ssubdirectoryFROM:

rselect.c srselect.crnkslect.c
suselect.c tselect.c

plusachoiceofEXACTLYONECROSSOVEROPERATOR(inthefileintheproblem'ssubdirectoryFROM:

oneptx.c (for non -permutationproblems)
twoptx.c(for non -permutationproblems)
unifx.c(for non -permutationproblems)
uobx.c(for permutationproblems)
ox.c(for permutationproblems)
cx.c(for permutationproblems)
pmx.c(for permutationproblems)

plusachoiceofEXACTLYONEMUTATIONOPERATOR(inthefileintheproblem'ssubdirectoryFROM:

bitmutat.c(for non -permutationproblems)
multimut.c(for non -permutationproblems)
scramble.c(forpermuta tionproblems)
swap.c(for permutationproblems)

plusachoiceofEXACTLYONEINVERSIONOPERATOR(inthefileintheproblem'ssubdirectoryFROM:

invercla.c
invercir.c

plus,forSINGLEPOPULATIONOPERATION,BOTHOF:

mainone.c
startup.c

ORforPARALLELSUBPOPULATIONOPERATION,ALLTHREEOF:

mainmany.c
initsubp.c
master.c

AUXILIARYMAINPROGRAM --COMPILATION

To compile the only **"auxiliary"** main programs provided with the GALOPPS system, for making a sampled data files for the blind traveling salesperson problem, the following directive (or its equivalent) may be used (the file cre8btsp.mak may be sourced by Unix users to do this, instead):

For **cre8btsp.c** (Creates distance table for randomly placed cities in a square of user -specified size, for blind traveling salesperson problem):
cc -gcre8btsp.cutility.c -lm -omakecity

NOTE: (At some point (probably not now), you may want to know this:)

The GALOPPS/Onepop and Manypops programs can read and write each other's checkpoint/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 files 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 (see appendices) 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 SHIBAEVA, Olga L., GARAGE, 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: 095/263 -65-26 (department office).

(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.

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.

APPENDIX ONE -- AUXILIARY FILES

List of the Auxiliary Files Provided with the GALOPPS System, Release 3.2, 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 GALOPPS 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 be alert in mind when running these examples or your own problems.

Information For Users (most in subdirectory "docs"):

README.FI.RST (ASCII text file describing this version of the GALOPPS system)

changes.txt (ASCII text file describing differences of this version from previous ones)

nowarran (ASCII text file disclaimer, indicating that there is **NOW WARRANTY** of any kind associated with the GALOPPS system --user uses at own risk)

guide32.ps (User's Guide for GALOPPS system, Release 3.2, in Microsoft Word 6.0 (.doc) and PostScript (.ps) versions)

license (GNU General Public License, under which GALOPPS is distributed)

Templates Defining the Structure of Input Files: (For users NOT using the new Graphical User Interface)

NOTE: GUI users: The GUI will automatically write correctly formatted input files for you, which you may then use with or without the GUI for further runs.

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 been looking for and what it has found, so it is very easy to develop a file with all inputs present in the correct order. (Template files are in subdirectory "docs"):

intempla.one (templatedefiningtheoptionalinputfileforOne pop)
intempla.man (templatedefiningtheoptionalinputfileforManypops)

Subdirectoryexamples/app

ApplicationFile **app.c**:(FromGoldberg'sbook)

makefile (samplemakefileforapp.cforOnepop
appone.in (sampleinputfileforapp.cforOnepop(i.e.,
mainone.c))
2runsapp.in (sampleinputfileforOnepop,2runs(diff.params))
appop4.in (sampleinputfile,app.cforManypops(4
subpopulations))
4popbest.mst (sample.mstfileusedbyseveralManypopsinputfiles)
contin1.in (sampleinputfileforOnepop,forEXTENDINGtherundoneby
appone.in)

Subdirectoryexamples/app1

ApplicationFile **app1.c**:(FromGoldberg'sbook)

makefile (samplemakefileforapp1.c)
app1one.in (sampleinputforOnepop(app1one)run)
app1pop4.in (sampleinputfileforManypop(app1many)multi -subpoprun)
4popbest.mst (sample.mstfileusedbyseveralManypopsinputfiles)
1st2of4.in (sampleinputfileforManypopsrunfromtwodifferentprocessors(or
processes --thisfilerunssubpops0and1)
2nd2of4.in (sampleinputfileforManypopsrunfromtwodifferentprocessors(or
processes --thisfilerunssubpops2and3)
readme (describesrunningofManypopson2machines,using1st2of4.in,
2nd2of4.in)

Subdirectoryexamples/app2

ApplicationFile **app2.c**:(FromGoldberg'sbook)

makefile (samplemakefileforapp2.c)
app2one.in (sampleinputforOnepop(app2one)run)
app2pop8.in (sampleinputforManypops(app2many)multi -subpoprun)
8pop2nbr.mst (sample.mstfileusedbyseveralManypopsinputfiles)

Subdirectoryexamples/royalrd

ApplicationFile **approyrd.c**:(JohnHolland'sRoyalRoadChallengeProblem)

makefile (samplemakefileforapproyrd.c)
approyrd.in (sampleinputforOnepop(approyrd)run)
approy8.in (sampleinputforManypops(approyr)multi -subpoprun,
butallrunningfromoneprocess)
approy8.in1 (oneofFOURsampleinputfilesforManypops(approyr)
foratrulyparallelrunon4processors(orfaked
with4processesononeprocessor,inUnix)
approy8.in2 (seeabove)
approy8.in3 (seeabove)
approy8.in4 (seeabove)

rr24to8.in (sampleinputforManypops(approyrp)multi -subpoprun)
 royalrd.sh (exampleshellscripttorunamulti -phase,multi -poprun)
 royalrd.bat (same,butforDOS)
 rrnoinit.sh (restartscriptforrunningroyalrd.sh,butwithoutthefirstroundof
 initializations)

 rrnoinit.bat (restartscriptforrunningroyalrd.bat,butwithoutthefirstroundof
 initializations)
 (plusvariousother.mstand.infilesusedforamulti -phaseManypopsrunbyroyalrd.shorroyalrd.bat)

Subdirectoryexamples/btsp
 ApplicationFile **appbtsp.c**:

 makefile (samplemakefileforappbtsp.c)
 appbtsp.in (sampleinputforOnepop(appbtsp)run)
 appbtsp8.in (Manypops(mainmany.c)run)
 8pop2nbr.mst (sample.mstfileusedbyseveralManypopsinputfiles)
 cre8btsp.c (programtomakeintercitydistanceinputfile.Torun
 thesampleinputfiles,youshouldruncre8btsp(or
 makecity,asexecutableiscalledunderUnix)to
 createfiles10cities.dst(forappbtsp.in)or
 20cities.dst(forappbtsp8.in.)
 cre8btsp.mak (samplecompilationcommand(canuse“sourcecre8btsp.mak”)for
 makingthestandaloneprogramwhichwrites the
 tableofdistances
 betweenrandomlyselectedcities,tousewithappbtsp.c.)
 cre8btsp.bat (samplecompilationcommand(cantypecre8btsp.bat)for
 makingthestandaloneprogramwhichwritesthe
 tableofdistances
 betweenrandomlyselectedcities,tousewithappbtsp.c.)

Subdirectoryexamples/demoinv
 ApplicationFile **appdemoi.c**:

 makefile (samplemakefileforappdemoi.c)
 appdemoi.in (sampleinputforOnepop(appdemoi)run)
 apdemoi4.in (Manypops(mainmany.c)run)
 apdemoi8.in (Manypops(mainmany.c)run)
 4popbest.mst (sample.mstfileusedbyseveralManypopsinputfiles)
 8pop2nbr.mst (sample.mstfileusedbyseveralManypopsinputfiles)

Subdirectoryexamples/demoalph
 ApplicationFile **app0to9.c** --Demoapplicationwithnon -binaryfields

 makefile (samplemakefileforapp0to9.c)
 ap0to9on.in (sampleinputforOnepop(ap0to9on)run)
 ap0to9p8.in (Manypops(mainmany.c)run)
 8pop2nbr.mst (sample.mstfileusedbyseveralManypopsinputfiles)

Subdirectoryexamples/dejongfn
 ApplicationFile **appdjf1.c -appdjf4.c** --First4ofDeJong’sTestFunctionSuite

makefile	(samplemakefileforappdjf1.c) --modifyfordjf2 -djf4.
apdjf1s.in	(sampleinputforOnepoprun)
apdjf1p8.in	(sampleinputforManypops(mainmany.c)run)(similarly,apdjf2p8.in, apdjf3p8.in,apdjf4p8.in)
8pop2nbr.mst	(sample.mstfileusedbyseveralManypopsinputfiles)
readme	(filedescribingexamplesinthisdirectory)

Subdirectoryexamples/diffreps

ApplicationFile **appdifrp.c** --Demoapplicationwithdifferentrepresentations,injectionarchitecture
(NoOnepopversionisavailablewiththisapplication)

makefile	(samplemakefileforappdifrp.c)
appdifr5.in	(Manypops(mainmany.c)run)
4to1inj.mst	(sample.mstfilefor“injectionarchitecture”toinjectlow -resolution solutionsfromfoursubpopsintoonehigher -resolutionsubpop)
readme	(descriptionofexampleprovidedhere)

Subdirectoryexamples/gga

ApplicationFile **appbpp.c** --DemoapplicationofuseofGroupingGeneticAlgorithm(GGA)
representationtosolveabin -packingproblem

makefile	(samplemakefileforappbpp.c)
gga.c	(additionalproblem -independent‘C’codeforproblemsusingGGA representation)
gga.h	(includefileforproblemsusingGGArepresentation)
ggax.c	(crossoveroperatorforusewithGGArepresentation)
ggamut.c	(mutationoperatorforusewithGGArepresentation)
apbp1002.in	(sampleinputfileforusewithdatasett1002.datforbinpacking (Onepop))
apbpp501.in	(sampleinputfileforusewithdatasett501_00.datforbinpacking (Onepop))
bpp501p4.in	(sampleinputfileforusewithdatasett501_00.datforbinpacking (Manypops))
bpp81002.in	(sampleinputfileforusewithdatasett1002.datforbinpacking (Manypops))
4popbest.mst	(sample.mstfileusedbyseveralManypopsinputfiles)
8popring.mst	(sample.mstfileusedbyseveralManypopsinputfiles)
readme	(textfiledescribingGGAexample)
t501_00.dat	(sampledatafileof501itemstobepackedintobins)
t1002.dat	(sampledatafileof1002itemstobepackedintobins)

APPENDIX TWO:

CONTENTS OF THE FILE TYPES WRITTEN BY GALOPPS

I. Filexxxxxx99.stt:

Runs of Many pops which involve more than one PROCESS (i.e., not all subpops are recalculated by a single process) create or read and write a file called xxxxxx99.stt, where xxxxxx is the checkpoint file prefix 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 Many pops run. (One pop run does not need or use this file at all, and neither does Many pops when all subpopulations are recalculated within a single process.)

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 EXTENDS 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 extended, so it will be used when the user specifies the restart file prefix 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 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 users should RECOGNIZE that the global statistics have been affected by adding in off-fitnesses, evaluations, etc., even though the populations which did them have been THROWN OUT. This does NOT occur if the Many pops processes are terminated normally at the end of a cycle, or if only a single process is being used to run all subpopulations.

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 (chromsizeunsigneds)
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 redefined (pointed to by utility on the chromosome), the utility fields' contents are written hereby app_write_utility_fields, which the user must define if using utility fields.

II. Files with extension.ind

Files with extension.ind are written in filecheckwt.c, by function writecheckpoint().

The filename to be written is contained in the string checkpointindfilename. 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 One pops, this name is assembled from the user input (file or keyboard), up to 8 characters, using the field "checkpointfileprefix", and appending ".ind". If the user does not specify a checkpointfileprefix, then the restartfileprefix is used, unless none is specified; in that case, the default "sgackp" is used. For Many pops 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 filenames as "runone00.ind", "runone01.ind", etc. Each subpop has its own checkpoint individual file.

At the end of a complete cycle of all subpopulations in a Many pops run, all of the checkpoint individual files written during the cycle are updated to exchange their read and write buffers. This enables a restart to be performed "fairly", regardless of when a run may have been interrupted.

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 chromsizeunsigneds and the utility fields
lchrom	Length of chromosome in bits

```

bestnow      Indexinoldpopofcurrentbestindividual[0,popsize -1].

(Then,foreachindividualfrom0topopsize -1:)
{
chrom        chromsizeunsignedscontainingtheactualchromosome

fitness      scaledfitnessforthechromosome

init_fitness raw(unscaled)fitnessforthechromosome

neval        numberofevaluationsperformedinTHISsubpopulationthisindividualwasfound.

xsite[0]     site[0,lchrom -1]onparentchromosomeswhererecrossoverwasdone(oneptx)orsubstringstarted
              (twoptx)whenthisindividualwascreated,or0ifcreatedsomeotherway.

xsite[1]     sitewheresubstringendedonparentchromosomesifthisindividualwascreatedbytwoptx;0
              otherwise.

parent[0]    positioninoldpop[1,popsize]]ofparent[0]ofthisindividual(onemorethansubscriptinarray).

parent[1]    asabove,secondparent.

utilityfieldsifutilityfieldsaredefined(pointedtobyutilityonthechromosome),theutilityfields'contents
              arewrittenherebyapp_write_utility_fields,whichtheusermustdefineifusingutilityfields.
}

```

III. Files with extension .ckp

Files with extension .ckp 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 Onepopsruns, this name is assembled from the user input (file or keyboard), up to 8 characters, using the field "ckptfileprefix", and appending ".ckp". If the user does not specify a checkpoint file prefix, then the restart file prefix is used, unless none is specified; in that case, the default "sgackp" is used. For Manypopsruns, 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 filenames like "runone00.ckp", "runone01.ckp", etc. Each subpopulation in a run has its own checkpoint header file.

At the end of a complete cycle of all subpopulations in a Manypopsrun, all of the checkpoint header files written are updated so that their read and write buffers are exchanged. This enables a restart to be performed "fairly", regardless of when a run may have been interrupted.

File contents are:

VariableName	Explanation
popsiz	numberofchromosomesinpopulationbeingcheckpointed
indcnt	numberofindividualsbeingrecordedintheaccompanying.indfile(alwayspopsiz,if writtenbythisfunction,butrecordedasecond timeforfutureflexibilitywithout changingfileformat).
chromsiz	numberofunsignedintsinasinglechromosome
numfields	thetotalnumberoffieldstoberepresentedonthechromosome.
fieldlength	(calculated)lengthofeachfield,forpermutation -typeproblems.
this_pop_map_to_std[i]	arrayof <i>numfields</i> intindices,tellingwhichfieldinthestandardrepresentationisnow representedatfieldIofthecurrentpopulation'schromosome(neededwhenusing inversion)
max_chromsiz_among_all_subpops	whenuserdefinesdifferentrepsfordifferentsubpops,specifiesherethelongestonein useinanysubpop(innumberofunsigneds)
max_numfields_among_all_subpops	whenuserdefinesdifferentrepsfordifferentsubpops,specifiesherethemaximum numberoffieldsusedinanysubpop
field_ends[i]	arrayof <i>numfields</i> intindices,tellingtheendingbitposition(numbered1..lchrom)of eachfield <i>ione</i> eachchromosomeofthissubpop.
field_siz[i]	arrayof <i>numfields</i> intindices,tellingthenumberofdistinctintegervaluesfield <i>ion</i> eachchromosomeofthissubpopcanrangeover(i.e.,rangeis[0, <i>field_siz[i] - 1</i>]).
bestfit.chrom	chromsizunsigneds,containingthechromosomeofthemostfitindividualfoundto dateINTHISSUBPOPULATION.
bestfit.fitness	scaledfitness(ifscalingused;otherwise,rawfitness)ofmostfitindividualtodateIN THISSUBPOPULATION.
bestfit.init_fitness	unscaled(raw)fitness(fromuser -definedroutineobj func())ofmostfitindividualto dateINTHISSUBPOPULATION.
bestfit.neval	NumberofevaluationsINTHISSUBPOPULATIONONLYofindividualsupthrough firstfindingthismostfitindividualofTHISSUBPOPULATION.
bestfit.generation	Generationnumber(forTHISSUBPOPULATION)atwhichthismostfitindividualOF THISSUBPOPULATIONwasfirstfound.
bestfit.map_to_std[i]	arrayof <i>numfields</i> valuesstoringthethis_pop_map_to_stdattimethisindividualwas saved,allowinglaterdecodingofit
bestfit.field_ends[i]	arrayof <i>numfields</i> valuesstoringthefield_endsarrayforthesubpopatthetimethis individualwassaved,toenablelaterdecodingoftheindividual,evenifarraychanges
*(bestfit.utility)	Iftheuserdefinesutilityfields,theircontents(forthebestindividualINTHIS SUBPOPULATIONtodate)iswrittennext,byacalltotheuser -definedutility writer,app_write_utility(bestfit.utility,fp).Fieldbestfit.utility(notwritten)isthe pointerthesecontents.
std_field_ends[i]	arrayof <i>numfields</i> valuesstoringthefield_endsarrayfortheSTANDARD representation(i.e.,usedbythefitnessfunctionobjfunc()),butNOTnecessarilyin thissubpop
one_pop_cum_sumfitness	

	Sumoftherawfitnessesofallindividualsevaluatedinthissubpopulationtodate.
one_pop_sum_best_fitness	Sumoftherawfitnessofthebestindividualofeachgenerationofthissubpopulationevaluatedtodate.
one_pop_best_fitness_count	Countofthenumberofindividualsevaluatedinthissubpopulation'sone_pop_sum_best_fitness.Thus,offlineperformanceforthissubpopulationcanalwaysbecalculatedasone_pop_sum_best_fitness/one_pop_best_fitness_count.
one_pop_online_denominator	Numberofindividualsevaluatedinthissubpopulationwhoserawfitnessesareincludedinone_pop_cum_sumfitness.Thus,onlineperformanceforthissubpopulationcanalwaysbecalculatedas: $\text{one_pop_cum_sumfitness} / \text{one_pop_online_denominator}$
minraw	Lowestrawfitnessinthecurrent(sub)population.
avgraw	Averagerawfitnessinthecurrent(sub)population.
fit_max	Highestrawfitnessinthecurrent(sub)population.
pcross	Probabilityofcrossover(perchromosome),in[0.,1.].
pmutation	Probabilityofmutation(perBIT,forordinarybinaryrepresentations,butperCHROMOSOME,forpermutation -typerepresentations).
p_multimut	Probabilityofmulti -fieldmutation(perCHROMOSOME)
avg_mut_width	Expectedvalueofnumberoffields(Pois son-distrib.)whichwillbesubjectedtosetting atrandomwhenamultimutisdone(takenfromapp_set_options()).
pinversion	Probabilityofinversion,pergeneration.
autocontrol_selection	Flagtellingwhethertocontrolselectionautomatically(Shapiro -likescheme)ornot
beta	BoltzmannscalingparameterspecifiedbyuserwhenNOTautocontrollingselection,or setautomaticallyifAREautocontrollingselection.
bets_s_tilde	Valueusedtodeterminebetawhenautocontrollingselection
scalemult	Multipleofaveragefitnesstowhichmostfitindividualistobescaled.Alsousedwith samemeaningwhenrank -basedselectionisused.Value -1meansdon'tdolinear scaling.
conv_measure	Resultofaconvergence measure;seebodyofmanual.
conv_sigma_coeff	Numberofstandarddeviationsabove/belowmeantobeusedforconvergence measure above.
sigma_trunc	Numberofstandarddeviationsfrommeanatwhichrawfitnessesaretobetuncatedin calculatingscaledfitness(0.0 meansdon'tdon'tdosigmatruncation).
scaling_window	NumberofgenerationsbackfromwhichLEASTfitindividualwillbeusedto determinescalingscalingofcurrentpopulation(-1meansdon'tuse;0meanscurrent generationonly,1meanscurrentpluspreviousone,etc.;maxvaluecurrently19).
windowstart	Pointerfortrackinggenerationsforwindowscaling.
windowend	(sameasabove.)
savemin[20]	Arrayofminimumsusedforwindowscaling.
sigma	Standarddeviationofcurrentrawfitnesses.
neval	Numberofchromosomesevaluatedincurrent(sub)population(todate).
lchrom	Lengthofthechromosome(inbits).

genspercycle (Many popsonly) 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, runbeg ins with highest value gen attained in previous run, so maxgen must be larger than that or unatall.

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).

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 restart scan pickup 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 it is (== 1) a permutation problem or not (== 0)

tourneysize Int telling size of tournament if tournament selection is used.

ANYUSER -DEFINED VARIABLES WHICH MUST BE RECORDED (except utility fields)
 Any values written to the file by CallBackFun(), which are added to end of "standard" file.
 User provides the code to write these variables (if any) in a function called app_write_ckp_hdr() in file appxxxx.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()).

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

NOTE: This appendix describes the SGA-C release on which the GALOPPS 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 textbook) to the present system.

SGA-C: AC -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 assumption 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.

Files Distributed with SGA -C (EDGNOTE: 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 external 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 the GA 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 subtractive method specified by Knuth:81.

[rnd(low,high)]returnsanuniformly -distributedintegerbetween lowandhigh.
[rndreal(low,high)]returnsanuniformly -distributedfloating pointnumberbetweenlowandhigh.
[flip(p)]flipsa biasedcoin,returning 1withprobabilityp, and0withprobability1 -p.
[advance_random()]generatesanewbatchof55randomnumbers.
[randomize()]askstheuserforarandomnumberseed.
[warmup_random()]primestherandomnumbergenerator.
[noise(mu,sigma)]generatesanormalrandomvariable withmeanmuandstandarddeviationsigma.ThisroutineisnotcurrentlyusedinSGA -C, andisonlyincludedasageneralutility.
[randomnormaldeviate()]isautilityroutineusedbynoise. Itcomputesastandardnormalrandomvariable.
[initrandomnormaldeviate()]initializationroutinefor randomnormaldeviate().

[report.c]containsroutinesusedtoprintareportfromeachcycleof SGA C's operation.

[report()]controlsoverallreporting.
[writepop()]writesoutthepopulationateverygeneration.
[writechrom()]writesoutthechromosomeasastringofonesand zeroes.Inthecurrent implementation,themostsignificantbitistherightmostbitprinted.

Threeselectionr outinesareincludedwiththeSGA -Cdistribution:

[rselect.c]containsroutinesforroulette -wheelselection.

[srselect.c]containstheroutinesforstochastic -remainderselection (Booker:82).

[tselect.c]containstheroutinesfortournamentsselection (Brindle:81a).
Tournamentsofanysizeuptothepopulationsizecanbeheldwiththisimplementation.[Thetournament selectionroutineincludedwiththedistributionwaswrittenbyHilolKargupta,oftheUniversityof Alabama.]

Formodularity,eachselectionmethodismadeavailableasacompiletimeoption.EdittheMakefiletochoosea selectionmethod.Eachofthreeselectionfilescontainstheroutinesselect_memoryandselect_free(calledby initmallocandfreeall,respectively),whichperformanynecessaryauxiliarymemoryhandling,andtheroutines preselect()andselect(),whichimplementtheparticularselectionmethod.

[stats.c]containstheroutinestatistics(),whichcalculates populationstatisticsforeachgenera tion.

[utility.c]containsvariousutilityroutines.
Ofparticularinterest istheroutineithruj2int(),whichreturnsbits\$i\$through\$j\$ofa chromosomeinterpretedasanint.

[app.c]containsapplicationdependentroutines.
UnlessyouneedtochangethebasicoperationoftheGAitself,youshouldonlyhavetoalterthisfile.
FurtherinstructionsforalteringtheSGAApplicationareincludedinthedescriptionofroutineapp.
[application()]shouldcontainanyapplication -specific computationsneededbeforeeachGAcycle.Itiscalledbymain().

[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 call to malloc() to dynamically allocate memory. This routine is called by init_malloc().

[app_free()] should perform any application -specific call 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., sgamy.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).
 The population size (int).
 The chromosome length (int).
 Print the chromosome string 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 runtime. 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.handexternal.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`.

This 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.handexternal.h`. However, take care that you do not redeclare existing variables.

Three example application 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 included in `app2.c`. This application illustrates two features that have been added to SGA-C. The first of these is the `thruj2int` function, which converts bits 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 `thruj2int` 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.