Contents

Gassy
A System For Implementing Genetic Algorithms

Terry Jones
Santa Fe Institute
terry@santafe.edu

September 1992

Chapter 1

Preface

1.1 Historical Note

The original Gassy was implemented late in 1991 at the same time as the first application. That application has since become one of the examples in the current distribution (busy_beavers). At the time, I was completely naive in my knowledge of GAs. I was not aware that there was more than one way to do crossover, or that there were many ways to do many of the things that have now become possible in Gassy. As a result, the design of the system was quite different.

Originally, Gassy was intended as a platform or starting point that would make the development of specialized GAs simpler. It is certainly true that Gassy can still be used in this fashion, but now I have the much more ambitious hope that the great majority of GAs can be implemented using Gassy *without* having to change a single line of the main Gassy distribution. Working towards this goal has been a process of identifying and removing assumptions about what an application might want to do in the course of a GA run.

Before embarking on the first Gassy, I had never heard of any other general GA package. When someone directed me towards GENESIS, it was already too late. Until then I thought that Gassy was fairly complete. After reading the manual page for GENESIS it became clear that there was rather a lot about GAs that I didn't know.

After talking to John Grefenstette (the author of GENESIS), I have stolen two lines of code from GENESIS. This came about after Robert E. Smith mentioned that GENESIS did mutation by calculating the interarrival time to the next mutation rather than generating a random number for each bit of each genome each generation. This idea was so attractive, both for its simplicity and potential speed up that I immediately knew that Gassy should do the same thing.

1.2 Acknowledgements

Thanks to the following for their comments which have helped or encouraged me at some point in the development of some version of Gassy.

David Ackley, Aviv Bergman, Stephanie Forrest, David Goldstein, John Grefenstette, John Holland, Steve Kinzler, Melanie Mitchell, Gregory Rawlins, Robert E. Smith and Stephen Wolfram

Thanks also to all the people who were brave enough to use Gassy in its earlier versions. They provided feedback on bugs and used the package in ways I had not anticipated, thereby exposing important limitations that I had unwittingly imposed. In particular I thank all the participants in the 1992 Complex Systems Summer School in Santa Fe and in Gregory Rawlins' GA seminar at Indiana University in the Fall of 1991.

This work was supported by all of: the Center for Research on Concepts and Cognition at Indiana University, the Department of Computer Science at Indiana University, the Department of Computer Science at the University of New Mexico, and especially, the Santa Fe Institute.

The development would have been *greatly* slowed, perhaps to the point of standstill, but for the fantastic software provided for free by the Free Software Foundation; primarily emacs, bash, gcc and gdb.

Chapter 2

Introduction

2.1 What is Gassy?

Gassy is a package designed to make the implementation and execution of Genetic Algorithms (GAs) as simple as possible. Gassy is written in the C programming language. This document makes certain assumptions about your knowledge — these can be found in the "Assumptions" section of this chapter.

The package provides the underlying machinations of GAs. The user fills in the gaps, writing an evaluation function and possibly several other functions, depending on the nature of the GA being implemented. Gassy can take care of details such generating the initial population, and performing mutation, crossover and selection. There are several built in functions for all the standard genetic algorithm operations – for example, uniform, single point and two point crossover can be selected for a run, from the command line, without writing a single line of C code or recompiling your application.

Gassy also makes it simple to add command line options to your applications and to use application–defined functions to replace built in ones. All such options and functions can be selected on the command line without the need for recompilation. When you come to run your application, Gassy provides a host of options for setting parameters, selecting functions, and displaying intermediate results – such as new best individuals and populations sorted according to fitness or genome, in many formats and at varying intervals. The package also provides for catching interrupts and restoring interrupted runs. Also provided are utility functions for such things as gray coding and conversion between binary numbers encoded as character strings and integers, or binary integers and floating point numbers over some range. Additionally, Gassy can provide automatic hashing of individuals with their fitnesses to avoid re–evaluation if your evaluation function is expensive.

The package comes with an application building program called **new_app** that is designed to make the construction of new applications as simple as possible. The program leads you through a series of questions in which you specify the characteristics of your application. **New_app** makes you an application directory and places a extensively commented C skeleton program in it that tells you exactly what functions your application must write, what values they will be called with and what they should return. **new_app** also produces a Makefile that will allow you to compile your application and link it with the main package by simply typing "make", as well as (optionally) a file describing all the details of the various types that are used by the main program and with which your application will expected to deal.

I have tried very hard to remove assumptions about the nature of the application that might be written to use Gassy. Support is provided for many departures from what is generally considered a standard GA. The package does not assume that the population size is constant, or that you will use only one fitness function in the course of a run, that your individuals are character strings, or that they are fixed sized, it does not assume that any of the parameters such as mutation rate or recombination rate will remain fixed throughout a single run of your application. It does not assume that the alleles on your genome occur with equal probability – either in the initial population or when mutating. It is possible for an application to use different functions for mutation and crossover and so on in the same run. The functions provided by your application have access to the population from the last generation – thus making it possible to implement selection algorithms such as CHC.

Of course, the more your application resembles a standard GA, the more Gassy can provide for you. This package has been used to write entire GAs in under 5 minutes, some of which contain as few as *ten* lines of executable C code. The simplest application, from Gassy's point of view, has fixed length, binary, character string individuals, a fixed population size, and uses standard (built in to Gassy) mutation, crossover and selection functions. In this case the application writer simply writes the evaluation function and a function that returns the length of individuals.

The generality of the package does not imply that it is also slow. I have taken pains to ensure that the package will run efficiently. The main mechanism employed is conditional compilation. The answers you provide to the **new_app** program when you build a new application result in an **application.h** file for your application which is included by the main Gassy programs that determines what support is necessary for your application. So applications that follow standard genetic algorithm practice are not hampered in any way by those that, for example, modify the size of the population and thus require extra help from Gassy. Similarly, if you are not interested in hashing, you will never pay the price of calls to hash functions or even of checks at run time to see whether you are interested in hashing – the hashing code will simply not be there when your application is compiled.

2.2 Assumptions

There are some assumptions I have made that you should be aware of before you try to read this document or use the Gassy package.

- This document does not attempt to explain what a GA is or what it might be good for. For an introduction to the subject, a good starting place is Goldberg's book [1].

- The second important assumption is that the user of Gassy is familiar with the C programming language. If you are not, you should consult Kernhigan and Ritchie [2] and you will probably need to rely heavily on that and the example Gassy applications included in this distribution when you come to writing your own applications. Gassy is written in traditional C, not in ANSI standard C. Buying the ANSI standard version of [2] is, however, strongly recommended. The visible differences between the two are not crucial to your understanding of the vast majority of existing C code, and are mainly in the declaration of arguments to functions. Gassy is written in traditional C as, despite the availability of an extremely high–quality, *free*, ANSI C compiler,[1] many people do not yet have one.

- The application building program **new_app** is actually a shell script. This is a program written in the shell programming language which is available on all versions of the UNIX[2] operating system. If you are not using UNIX, you will almost certainly not be able to use **new_app**. This is a shortcoming of the package. It *is* possible to write applications without using **new_app,** but this task is somewhat complicated and arduous. A forthcoming version may well contain a VMS DCL script or a C program replacement for **new_app.** I apologize to the non–UNIX community and will try to support them more fully in the future.

- Even if you are using UNIX, it is possible that your implementation of **/bin/sh** does not provide shell functions. If this is the case then I suggest you obtain and install the excellent and free **bash** shell from the Free Software Foundation which will be able to run the **new_app** program after you change its first line — consult your system administrator for help with all this.

2.3 Terminology

In what follows, I shall use certain words and phrases to mean some very specific things. These are:

GA GA will be used throughout this document as an abbreviation for genetic algorithm.

application An application is a set of C functions written by the user of Gassy that are compiled and then linked with the functions provided by the main package. The result is often also referred

[1] The excellent GNU C compiler, gcc.

[2] UNIX is a trademark of AT&T Bell Laboratories.

to as the application. There is a single Gassy package, but there are many applications.

individuals The population of a GA is composed of some number of **individuals**. This is a deliberate attempt to escape the predominating tendency in the GA and Artificial Life communities for people to use terms such as "he" and "guys" when talking about the things their programs deal with.

genome The genetic text of an individual.

locus A single location on the genome. The plural is **loci**.

allele The value at a locus.

command line option This is something provided on the command line of an invocation of your application that is introduced with a "–" sign. It may often be followed with an associated value. For example, "–population 100".

command line argument Something that appears on the command line following all the command line options. This may be the name of a file to read, or a file into which output should be placed. Command line arguments (in general) will never start with a "–" sign.

2.4 Changes From Versions Prior To 3.0

There are not terribly many people who used versions of Gassy prior to 2.0. Unless you are one of those people, this section will probably be of little or no interest. It can be safely skipped.

The differences between versions to 2.0 are quite minor. For that reason, the following treats all versions to 2.0 as though there were identical. The changes from version 2.0 to version 3.0 are very substantial. Version 2.0 was about 5,000 lines of C. Version 3.0 is currently 9,000 lines and is still growing. I am very happy with the design changes in version 3.0 – they have resulted in both a cleaner and more general implementation and, importantly, at the same time, the work the application writer must do has been reduced and is simpler.

Applications that were written for 2.0 or earlier will **not** work unchanged under version 3.0. I apologize for this, but I thought it necessary. Fortunately, it shouldn't be too hard to convert old applications to new. There will never be such a major change again, at least I cannot forsee such.

There were several things I was unhappy with in the design of version 2.0. The most obvious, from the application writer's point of view were:

- The number of command line options that Gassy provided to each application was becoming truly unwieldy. It was common to have to use fifteen or more one letter options to start a run. The option letters were somewhat intuitive, but it was hard to get them all straight. Additionally, if applications wanted to define their own options, which was simple enough, they could not use any of the letters that were reserved for use by the main program. I tried to leave the uppercase letters alone, but it was clear that option processing was going to need attention in the next version.

  In version 3.0, option names are words. For instance "–population" to specify the population size. Options may also be abbreviated. Abbreviations that cannot be resolved unambiguously will cause an error and all possible completions are presented. Thus one can use "–pop" or even "–p" if the application does not also define an option that starts with "p" that can be abbreviated using "–p". As a result, Gassy no longer has the feel of some horribly complicated monster that takes a million options with obscure one letter abbreviations.

  In fact, the number of options in Gassy has grown to about forty. Adding new options to applications is now even easier than it was. It is now just a matter of declaring a variable associated with the option, and adding a single (long) line to an array of option description structures. These structures contain, among other things, the address and type of the variable. The main program will assign a value to your variable, check the limits of the value given (if appropriate), call a function for you and more.

  In addition, it is now possible to use the "–options" option to specify a file that contains more options. Using this it is possible to create a file of option definitions for use with many runs. Options are processed left to right on the command line, so it is easy to change options that were set earlier via a "–options" option. Thus one can set up a file of options that are common to all runs and override some of them later on the command line. It is also legal for one file of options to include another, again via the "options" option.

In many cases I have defined abbreviations for options that are the same as those for version 2.0. Those who prefer invocations with command lines that look like "–p 100 –g 100 –x 0.7 –e –I –c 2 –h –s t –t 0.6 –v" will be happy to know that this (or some close variant) will still work.

- Another thing I was unhappy with was the method used to implement different functions for things like mutation and crossover. I felt that the mechanisms here were not general enough and that a more flexible and general method should be employed. This has now been done and functions can now be specified like any other option (they are processed by the same mechanism), and abbreviated and given longer, meaningful names. Previously there were fewer options available for each of the various customizable things and there was a switch statement to decide which function to call. This has now been changed and the code keeps a pointer to each function.

- It is no longer necessary to write function stubs for a dozen or so functions that you never use. In 2.0 it was necessary for every application to define a set of about 15 functions. Many of these were never called and the empty definitions cluttered up otherwise clean application files. Moreover, they were confusing to beginners and made explaining otherwise simple example applications a long and tedious task for me. In the new version, the application builder **new_app** creates a file of **#define**s that is used by the main package to both build the application template file and to conditionally compile the main program. As a result, the template contains stubs only for the functions that actually need to be written by the application, and the main program never calls things like, for example, **app_crossover** if the application builder said that they were not interested in writing their own crossover functions.

This has resulted in the main Gassy code becoming quite a bit more complex, but the gains in simplicity for the application writer far outweigh the costs to me.

- Applications now live in their own directories. After quite a bit of rearranging and rethinking I arrived at a solution to all the problems associated with applications having different fitness types and the resulting problem of object files compiled with FITNESS being one type getting accidentally linked with others that had a different fitness type. In the new set up, there is only one copy of the application source and each application builds its own object files in its own directory. There is no longer any need to edit the Makefile to change the FITNESS definition when compiling a different application, as each application now has its own Makefile. The Makefile is built by the application builder and there is usually nothing more to making an application than typing "make". In most cases the Makefile will not need to be touched.

- Probabilities are now all treated consistently. Instead of "–m 1000" to specify a mutation probability of one in a thousand, one now says "–m 0.001".

- The statistics that were printed were rather primitive. I enhanced this at the Complex Systems Summer School in June 1992, and have since added quite a number of useful options to control what is printed and when. It is now possible to request things such as the printing of the top 5 population members, sorted by fitness and then by genome (with duplicates removed) every 7th generation. It is also possible to sort by genome, by genome and then by fitness (in the case of a noisy fitness function) and so on.

- The random number generation in earlier versions was not well thought–out. It is very important that a package such as Gassy allow for the interruption and restarting of runs. In the course of a run, the random number generator may be called millions of times. When an interrupted run is resumed, it is a sine qua non that the generator be restored to its former state, otherwise the run will not continue as it would have had the interrupt not been received in the first instance.

For this reason, version 2.0 kept track of the number of calls to the random number generator and when a run was restarted from an interrupt file, the generator was called that many times. This resulted in slow resumption.

Even worse was the restrictions that were placed on the application writer as a result of this design. Because the generator being used was **lrand48()** it was only possible to generate random long integers, and one had to go through contortions if one wanted doubles and also that the run be repeatable. If the application called **drand48()** or some other random number generator, all bets were off.

This was unacceptable and so I obtained a random number generator designed by Knuth. I initially took this from [**?**] but was not satisfied with their code and so went to the source [**?**]. The blow by blow description of this effort is in the file **random.c** in the distribution. The result was applications that mysteriously ran three times as fast and the ability to restart interrupted applications by simply initializing an array of sixty or so integers. Also, the application writer now can use the random number generator to generate doubles and a utility macro which converts these to integers over a given range.

The package also attempts to catch people who try to call other known random number generators.

It is hard to recall all the changes that have taken place in the move to version 3.0. They have been extensive and have taken a lot of time. I am very pleased with everything so far and again I would like to thank everyone who used an earlier version of Gassy and reported problems, suggested ideas and so on. Each of you contributed to my producing the new and improved version.

Chapter 3

An Example Of Gassy Use

3.1 Maximizing Ones

This is one of the examples that are distributed with Gassy. Here a GA is built that searches for genomes that are composed completely of "1" characters, where each locus on the genome may be either "0" or "1". The genome has a fixed length of 64 characters. For a more general example, in which genome length is given on the command line, see the example directory **max_ones**.

The source for this example can be found in the **simple_example** directory of the main Gassy distribution. The application directory was created by the **new_app** program, as were all the files in that directory. The main program can be found in **simple_example.c** and is reproduced below. Writing the application consisted of editing that file and adding the ten or so lines of code, of which only seven were actually executable.

```
/*
 * This program tries to find binary character strings of length 64
 * that consist of all ones. The fitness is the number of ones in the string.
 */

#include "types.h"
```

```
#define GENOME_LENGTH 64

INT
app_individual_size()
{
    return GENOME_LENGTH;
}

FITNESS
app_evaluate_individual(who, context)
INT who;
CONTEXT *context;
{
    INT i;
    FITNESS fitness = (FITNESS) 0;
    INDIVIDUAL_TYPE genome = context->population[who].genome;

    for (i = 0; i < GENOME_LENGTH; i++)
        if (genome[i] == '1')
            fitness++;

    return fitness;
}
```

The above code is probably largely self–explanatory. The file **types.h** must be included by all applications and contains the definitions of all the types that Gassy uses and with which it communicates with the application's functions. For details about these types, consult the Types section on page 41.

When **simple_example.c** was originally created by **new_app**, it contained everything you see, except for the comment, the **#define** and the contents of the two functions.

The function **app_individual_size()** returns the length of the genomes in this application. This is called once at the beginning of the run when the main program creates the initial population.

The second function, **app_evaluate_individual()** is the fitness function. The type it returns is **FITNESS**, which in this case is an integer. This was all arranged during the run of **new_app** and the type definition is in the **types.h** file. The fitness function is passed two arguments. The details of the type **CONTEXT** can also be found in **types.h**. It suffices to say for now that the **context** argument contains, among a large number of other things, the current population, and that **who** is an index into that population, denoting the individual to be evaluated.

This may seem unnecessary. Surely an evaluation function just needs a pointer to the individual to be evaluated? In a standard GA, this is true. But it is conceivable that the fitness of an individual is dependent on other individuals in the population, and so Gassy, attempting to be as general as possible, passes the entire context of the run. The **context** argument contains other things that might be used or changed in evaluating this individual.

For convenience, the function declares a variable **genome** that points to the character string that defines the individual to be evaluated. Once this is done, the fitness calculation is a simple loop that counts the number of ones in the genome.

To see how this example program could be compiled and then run, consult the chapter on running applications (page 43).

Chapter 4

GAs And Gassy

4.1 The Gassy View Of GAs

When Gassy performs a run of your application, there are several distinct steps that are taken. With the exception of the evaluation function, which the application must provide, all of these steps can be done using the default settings and functions built in to Gassy. On the other hand, it is possible for an application to determine how virtually everything is to be performed. An application can even change these settings during a run.

For example, one step that must be performed is crossover. The application writer can choose to use one of the crossover methods that Gassy provides, or it may do its own crossover. It may also do both, or modify the crossover probability during a run.

Another example is in the creation of the initial population. If the individuals are character strings and each allele at each locus on the genome is equally likely to occur initially, Gassy will happily generate the initial population. But perhaps this is not the case. The application writer can arrange to give different alleles different probabilities and let Gassy create the population. Or else it can specify part or all of the population explicitly and have the remainder generated via one of the above methods.

The following describes the way that Gassy performs the steps involved in running a GA. It does not describe the choices of built in functions you have to perform these operations. These are described in the section on what Gassy can do for you (page 28).

The steps taken by Gassy in a run are:

1. Create the initial population.
2. Evaluate the population.
3. Scale the fitness values.
4. Collect statistics.
5. Select survivors if the generation gap is not one.
6. Select individuals that will be used to fill the next generation.
7. Perform crossover on these individuals.
8. Perform mutation on the newly created individuals.

The first step is of course only done at the beginning of a run. Thereafter, Gassy loops through the remaining steps until some termination condition is met. Termination is most often caused by the target number of generations being reached.

The above steps require a reasonably detailed explanation. Even those who feel they understand all these ideas well should read the following as it describes details of the Gassy point of view that should be understood by the application writer.

Before we begin, it is important to understand that not all of what follows will be relevant to all applications. The sort of functions that Gassy will be expecting from you depends on the answers you give to the **new_app** program when you build your application.

4.1.1 The Initial Population

Let's treat the most complicated case first. If you tell **new_app** that your application's individuals are not constant size or that they are not strings, then Gassy cannot possibly help you to create an initial population. Gassy will have absolutely no idea of the size or data type (or both) of your individuals and so could hardly create a population of them. In this case, the template C program that **new_app** creates for you will contain a function stub called **app_create_random_individuals()** which will be called when Gassy wants some random individuals for the initial population. Note that this function will not be called if you specify an initial population file that contains enough individuals to fill the population.

Otherwise, as mentioned above, Gassy will be happy to help build the initial population. In this case, your individuals are character strings of fixed length. If your strings are binary and the probability of occurrence of "0" and "1" is the same at each allele, then you have nothing to do.

If your strings are not binary, then you will have to tell Gassy what the possible alleles are at each locus. An application that does this is the **my_name** example. **New_app** will arrange for a function stub called **app_allele_possibilities()** to appear in the C template it creates for you. If the probability of occurrence of each allele is the same, you have nothing further to do.

If your allele occurrence probabilities are not equal, then you must tell Gassy what they are. This is done by filling in the stub function **app_allele_probabilities()**.

Applications that have both non–binary alleles and non–equal allele probabilities will just write a single function, **app_initial_population_alleles()** which will do the work of both of the above.

From this information, it is easy to see how Gassy can generate an initial population for you. This information can also be used to perform mutation automatically too. One complication is that your application might wish the initial population to be generated with allele values that obey one probability distribution, but might want mutation to follow another. It is simple for an application to arrange to change the allele probabilities after the generation of the initial population. This will be described in the section on mutation below.

4.1.2 Population Evaluation

In the simplest view of a GA, each individual gets evaluated each generation. In practice, it is possible to improve on this by keeping track of which individual's fitnesses are already known. If the fitness function is noisy then this cannot be done, but often this is not the case. The **new_app** program will allow you to choose to take more or less of the responsibility for population evaluation in this case. From the application writer's point of view, the simplest thing to do is to just write a function that evaluates a single individual and returns the fitness. If you would like to keep things this simple, Gassy will keep track of the individuals whose fitnesses are already known and call your function on the remainder.

It is also possible to have Gassy pass you the entire population and ask you to evaluate it as a whole. In this case you can (and should) check whether the fitness of an individual is already known and you will have to store the calculated fitnesses somewhere. The details of all this are explained in the template file. In fact, code to do all this sort of housekeeping is provided.

There is one definite advantage to choosing this more involved route. When the population is evaluated, there is only one call to application code. Your function may contain the evaluation code in a loop and never do a function call. The simpler method will result in a function call to evaluate each population member.

A second advantage to this approach is for those whose evaluation function is not so simple that a single member of the population can be evaluated without considering the others. In real worlds individuals are not evaluated in isolation, but as a part of their environment – which includes the other individuals in the population. If your application requires this kind of evaluation, then Gassy will give you access to the entire population and you may do evaluation in whatever fashion you please.

4.1.3 Fitness Scaling

Fitness scaling can be used to amplify the differences in the values returned by the fitness function. From a numerical point of view, as is adopted by most selection methods, there may be little difference between an individual with fitness six and one with fitness eight. Semantically however, the difference may be great. One way to add emphasis to the small differences is to scale them in some way.

Currently Gassy has support for fitness scaling, but there are no built in scaling functions. In fact, the **new_app** program gives you the option of having the scaling housekeeping code omitted from the compilation entirely – ensuring that you don't pay for something you don't even use. In the future Gassy will support some scaling functions. In the meantime you can always write your own. If scaling is done, all of the following steps operate on the scaled fitness values.

4.1.4 Statistics Collection

If left to itself, Gassy will not collect any statistics. It is simply concerned with getting the job done. It is easy however to request that Gassy do some analysis each generation and print the results to the screen or to a file or files. It is also easy for an application to be given the chance to do its own statistics collection and printing. If the application writer tells **new_app** that they want a chance to collect statistics after each generation, the template file will contain a function stub called **app_generation()**. This function will be called after the evaluation of each population. The application may do anything it likes at this point. This includes actions that might be more devious than simply collecting statistics – for instance, the application may increase the population size, or change the selection function, but more of that later.

It is not necessary that the application print its results at this point. They can be stored and then printed at the end of the run if the application writer indicates to **new_app** that they would like a chance to do or say something at the end of a run. A complication can arise in this case if the application keeps an array of data with information for each generation recorded – how can the application know in advance how many generations will be run? A partial solution to this problem is available and is discussed in detail in the section on writing your application (page **??**).

4.1.5 Survivor Selection

It is common for an application to wish to copy some set of the current generation unchanged. The *generation gap* is the proportion of the population that gets replaced each generation. In a standard GA the whole population is replaced and thus the generation gap is 1.0. If the generation gap is less than 1.0 then some members of the current population must be selected for survival into the next. How are these individuals to be chosen?

Gassy will do the survivor selection for you using a built in survivor selection method (chosen from the command line), or you can write your own. You can write more than one for the same application and choose between them just as though they were built in to Gassy. Now recompilation is necessary. The functions that Gassy has for survivor selection are described in the section on what Gassy can do for you (page 28), and the details involved in writing your own functions are described in the section on writing the rest of your application on page 18.

4.1.6 Selection

Selection is the process of choosing individuals that are to be used to produce the individuals for the next generation. The individuals selected in this step are operated on by crossover and mutation to form offspring. Gassy takes a view of the selection process that is somewhat more general than the normal one. The traditional view of selection is that the selection algorithm chooses exactly as many individuals as are needed to make up the next generation after the survivors have been selected. These individuals are then randomly paired for crossover. Each crossover operation produces two offspring which are put into the next generation. Thus the population size is preserved. If the number selected is not even, an extra crossover can be performed and one of the children discarded.

This approach works, but it makes some forms of selection difficult to perform. A selection method such as sigma scaling might indicate that an individual should have 1.3 offspring. The implementor then has to decide whether to give it one or two offspring.

Gassy tries to be a bit more flexible. A selection method is passed the population and is expected to return an array whose elements index into the population. The important difference is that Gassy does not impose a size limit on the returned array. This means, for instance, that the array returned by the selection method might be ten times the population size. The array will be used to produce offspring by the crossover function. The crossover functions that are built in to Gassy select pairs randomly from the array and build the new population as described above. The array can represent a many to one mapping onto the population (i.e. many array values may index the same population member). A selection function such as sigma scaling could gain more accuracy by allocating a selection array ten times the size of the population and instead of being forced to give either one or two slots to an individual that should receive 1.3, it can give it 13. Of course this help is limited, if the scaling method indicated that an individual should receive 1.35 offspring, then what should be done?

The important thing to understand is the way Gassy expects selection functions to behave.

4.1.7 Crossover

Once again, the type of your individuals determines how Gassy performs some step. If you do not have constant length character strings as individuals, Gassy cannot know how to do crossover for you. In this case the only built in crossover functions will be **no_crossover_random_copy()** and **no_crossover()** which are explained later, but neither do any crossover. The application should write its own crossover function(s) if it wants to do crossover.

On the other hand, if your individuals are constant length character strings, Gassy can do a lot to automate the crossover process. There will be at least 3 additional forms of crossover available (described under what Gassy can do for you (page 28).

In this case, Gassy does crossover using the array of selections that are returned by the selection function. Each element of the array indexes a member of the population. All of the built in crossover functions in Gassy choose pairs at random from this array and produce two offspring that are copied into the next generation. If the number of places in the next generation that need to be filled by crossover is even, the final crossover operation discards one of the children at random. Gassy provides a generalized crossover function for use by applications that does most of this work for any crossover function that produces two offspring from two parents.

4.1.8 Mutation

Like crossover, the way mutation is done in Gassy depends on the type of the individual. If your individuals are fixed length character strings, Gassy can do all the work for you. In this case Gassy knows the possible alleles at each locus and also knows the probabilities associated with each. When mutation occurs, it is simple to choose another legal allele value for the locus. The application writer has nothing to do.

As mentioned above, it is possible that the allele probabilities for mutation will not be the same as those used to generate the initial population. You can tell **new_app** this and it will arrange for a function stub called **app_mutation_allele_probabilities()** that you complete, filling in the allele mutation probabilities for each locus on the genome. This function will be called by Gassy after the initial population is generated.

I appropriated a nice idea from the GENESIS GA package. In it, the time until the next mutation is calculated and then the package just counts alleles until it is time to mutate again. This is far more efficient than generating a random number for each locus on each genome in each generation. Thanks to John Grefenstette for his donation of two lines of code and the idea – which he says he took from Knuth.

If your individuals do not have constant size or are not strings, then Gassy will have no idea how to mutate them. In this case the only built in mutation function will be **no_mutation**. The application should write its own mutation function(s) and choose them from the command line. How this should be done is described in the chapter on writing applications.

4.2 Bits Versus Bytes

One question that is often asked is whether one should use bits or bytes to represent loci on genomes. The following discussion assumes that bytes each have n bits. In practice, n is usually eight, but it is becoming increasingly more common to find machines on which this is not true. Currently, Gassy uses a byte based approach. Other GA packages such as GENESIS and GAucsd (based on GENESIS) use a bit based approach.

One thing is obvious. If you are using bytes to represent loci, then unless you are using the full capacity of each byte, i.e. unless each locus has 2^n possible alleles, you are going to be wasting space. In the worst case, your GA will use about n times as much space as it would have with a bit–based encoding.

This seems like a big waste, but how important is it? This will depend on the population sizes you will be running your GA with and the type of computing environment in which your program will run. To my mind it is not clear that the space consideration is a critical one for most applications. It is very rare to find GA research in which populations have even a thousand individuals. A more common figure is something between fifty and a few hundred. On the other hand, it is very likely that most applications will find themselves running on a computer with many megabytes of main memory and also tens of megabytes of virtual memory. (Apologies to the majority of the PC world – although they too have systems with virtual memory).

Gassy allocates enough space for two populations and thereafter does not make space requests. Not for standard GAs anyway. If we suppose a population size of one thousand and a genome length of one hundred bytes, then this is a total of only 200K. This is a fairly modest amount of memory. Even with a memory allocator that uses a buddy–like system (as is fairly common) the total memory request for the populations will not exceed twice this figure.

Clearly space is being thrown away, but it seems that the amount will generally not be enough to cause trouble.

What else can we say about the byte based approach? The programming is certainly simpler. Null terminated strings are very well understood by most C programmers, whereas the average C programmer probably could not tell you how to find out the number of bits per byte in a portable fashion. In addition, writing portable code that uses a bit representation is not a simple task. Machines vary widely in the number of bits in their different types. Of course, all this can (and is) done, and that is the point of having nice packages to do the work for you. But what happens when the application writer wants to add their own crossover function?

Writing a crossover function that manipulates bytes is very simple. Writing one that does so with bits is definitely not so easy.

One solution to this problem is to let the application writer deal with the genomes on a character level whilst doing the underlying GA manipulations at the bit level. This is the approach currently adopted by GENESIS. This requires the translation from one form to another, from a stream of bits to characters. It is usual that the person running the GA will want statistics generated during the GA run. If these involve printing genomes in a readable form to a file, then translation must be done here too.

Do the translation times cost you more than what you gain by using a bit–based representation? I don't know the answer to this and there is no general one. It depends on how much translation must be done. I do know that there is a simple solution to all these problems and that it is to use a single byte–based view of the genome.

Another advantage that the byte–based view has is in debugging. This is no small boon either. How many C programmers are comfortable taking apart and examining the contents of unsigned integers with a debugger? How many debuggers allow you to do this in an easy fashion? Not only are the debugging benefits great for the person writing the GA package, but the benefits are also there for the application writer when they decide to write their own crossover or mutation function and wish to debug it.

Of course it will be countered that the bit based approach is so much faster than the character approach that there is no question which should be used. In general this is true, a single bit operation on an unsigned integer will be *much* faster than looping through a character array. But is it true that using bits as opposed to bytes in a GA setting is going to be equally beneficial? I tend to think that a reduced speed advantage will still be there but that the complications involved in getting the code as fast as it could be will great.

On the down side for the character based approach, there is a theoretical complaint that can be levelled at applications which use non–binary alleles in a character oriented setting. All the current theory about GA behavior and convergence assumes a genome that is binary. If you have an indivisible unit on the genome that can assume one of, say, sixteen values, then it is not possible to perform crossover or mutation *within* that section of the genetic text. Perhaps if the sixteen values were represented as four bits, the GA would find building blocks within the four. This is a serious objection.

Despite the lack of theory in support of loci whose alleles can take on more than two values, such applications do tend to work. Perhaps the current GA theory will be generalized to account for this case. At least three of the example applications provided with Gassy do not use a binary genome.

The wary can always encode their individuals in any way they wish. If you do have sixteen possible values for some parameter in your GA, there is nothing to stop you using four bytes in Gassy and then decoding them in the appropriate way in your evaluation function.

In summary, I do not know which approach is better. It is not clear what better is in this case, as is virtually always true in analysis of algorithms. I am not advocating one approach over another, but wish to emphasize that the bit based approach is not necessarily faster than the character based one, and that the space wasted by the character based approach is probably not going to be an issue for most GA runs.

I have not performed any time comparisons between Gassy and any other GA packages that use bit based encodings. These would probably be interesting, but also probably unfair to one application or another.

Best would probably be to allow Gassy use the bit based approach as well as the character based approach. Then direct comparisons could be done in which all other things would be equal. I have always had plans to add this functionality to Gassy, but it is currently very low on a very long list. In the meantime I may perform some timing tests that will provide crude estimates.

4.3 What Gassy Can Do For You

4.3.1 Scaling

4.3.2 Survivor Selection

4.3.3 Selection

4.3.4 Crossover

4.3.5 Mutation

Chapter 5

Writing Applications

5.1 Introduction

As you may have gathered and will soon be told explicitly, Gassy relies heavily on conditional compilation for maintaining simplicity and speed whilst remaining general. There are currently over thirty possible symbols that may be **#define**'d or not defined which will alter just what you get when you compile the Gassy package. For this reason there are perhaps 2^{30} possible compiled forms of Gassy. Thus there is no standard Gassy – typing **make** in the main distribution directory makes each of the examples that are distributed but does not result in any executable object being placed in the main directory.

Each application resides in its own directory and when it is made the Gassy sources are compiled and the object files are put into the application's directory. This way every application has its own object files (which may be removed once the application is built) but there is only one copy of the actual Gassy source. Makefiles can make the above arrangement simple in practice.

Because there are such a large number of possible symbols that determine what parts of the main Gassy sources are compiled, it is very hard to know what you should define in order to get the code you need and, also important, exclude the code you don't need. Of course this can be done, but not without quite detailed knowledge of the main sources. One of the aims of Gassy is that no–one should have to consult or understand or modify the main sources unless they particularly wish to.

For this reason, a program, **new_app** is provided to ease the pain of building a new application.

5.2 The new_app Program

The **new_app** program is a UNIX shell script that greatly aids in building new applications. If you are working on a UNIX machine but you don't know what a shell script is, don't worry, it'll most likely work just fine.

The program asks a long series of questions about the new application. There are at least twenty questions at the moment, and that is likely to grow a little in the future. But you only have to go through the process once and it does make things far easier. In fact, I am not even sure how successful I would be at creating a new application without running **new_app**. It would certainly take a lot longer. All the examples in the Gassy distribution were built with it.

The program should be run in the main Gassy directory. If it is invoked with an argument (anything will do), it will use "expert" mode and not present you with a lengthy explanation before each question.

Once the program has run, several things will have been done for you:

1. A directory for your application will have been created. This directory will have a name that is the same as the name you choose to give your application.

2. In this directory you will find several files. The most important to you will be the skeleton file containing stubs for the functions Gassy will be expecting from your application. The **new_app** program will offer to put comments in this file for you. The first times you run Gassy, this is an offer you should definitely not refuse! There are a lot of comments and they describe exactly what you are expected to do. The file name will be the same as your application's name, with a ".c" appended.

3. Also in the application directory you will find a file called **application.h**. The contents of this file are the direct result of your answers to **new_app**. There will probably never be any need for you to look at or edit or include this file in any of your programs. You may feel free to do so of course, but you usually shouldn't need to. The only exception to this is when your individuals are some type other than string and you have not yet decided on the name of that type. If you can't tell **new_app** what the type is yet, that's fine, but you will have to make a small change to the **application.h** file. The location and nature of the change will be obvious to you when you look in the file.

4. You will also find a file called **Makefile**. If you are unfamiliar with the **make** utility in UNIX then this is a good time to get acquainted. That doesn't mean that I'm going to explain it to you. Suffice to say that make is an extremely useful program that will build your application's executable file for you. The **Makefile** knows where the main Gassy sources are located and exactly how to compile them. When you have filled in the stub functions in the skeleton file, you can build the entire application by just typing "make" in the application's directory.

   As with the **application.h** file, there will normally be no need for you to examine or edit the **Makefile**. If your application is very large however, you may want to split it into several source files. If this is the case, then you'll need to do some minor editing on the file. If you feel out of your depth because I have not said what sort of editing may need to be done, ask a system administrator, or, simpler, just leave your application in the single skeleton file and all will be well.

5. The last thing that can be generated is optional. It is a file called **TYPES** which will contain a description of all the types that will be applicable to your application. This file is not used by any C program, it is merely for your information. It is generated from the file **types.h** in the main Gassy directory. You cannot simply read that file when you are wondering what sorts of things a certain type contains as that file is riddled with **#define**'s –

> to the point of being unreadable. The **new_app** program uses the generated **application.h** file to produce a **TYPES** file that contains just the types you need, each with only the fields that are relevant to your application. The skeleton C program that is generated for you is made in almost exactly the same way from the file **main_template.c** in the Gassy directory. As before, unless you are already very comfortable with Gassy, it is a good idea to let **new_app** generate this file for you.

Almost every question that the **new_app** program asks will have an associated default answer that you will get by simply hitting RETURN. In every case, the default answer is designed to give you the most simple, standard and helpful application configuration. Having named your new application, if you hit return for every question you'll end up with an application that uses fixed length binary character strings, that uses built in crossover, mutation and selection functions, that expects no command line arguments or options, all in a nicely commented C skeleton file. In fact the file will be extremely simple and will contain just two function stubs.

Yes/No questions are interpreted as being a "Yes" if the first character of the response is a "y" or a "Y". Thus you may use "Yeah, yep, yee–ha" for "Yes" and "Nope, Nah, no–way" for "No". You get the idea.

You can quit the **new_app** program at any time by interrupting it from the keyboard. On most UNIX system this is done with control–C. If the application directory has been created, it will be removed.

If you find the question that the program asks is something you don't understand or you are not sure about what answer to give, it is probably best to take the default. It is easy to add functionality to your application later on if you decide, for example, that you want to have a command line option. Here are the questions that the program asks. The description here of the particular feature that Gassy is asking you about are very similar to those you will see when you run the program. There is sometimes slightly more detail here.

One thing to be aware of when reading through the following is that in any one run of **new_app**, you are not likely to see *all* the following questions. In fact, there are some you may never see. It all depends on what type of applications you create. If you tell the program that your individuals will not be character strings, it obviously isn't going to ask you questions about the possible alleles and their probabilities at each locus, since Gassy will not know anything about your individual type or what it may consider an allele (if anything).

- The first thing you need to supply is the name of the new application. As mentioned, this will be the name of the directory that is created to hold your application and also the base name of the skeleton C program that will be constructed for you.

- Often, the particulars of a new application are exactly the same as those of an old one. If your application will be the same as some other application you created using this program, you will probably just want to copy that application's setup files.

  By "exactly the same", I mean that the individuals are the same type, that the fitness type is the same, and so on.

  Usually you will want to create a totally new application as there may be no previous application that was identical to this one. But if you use the package a lot, you may want to start producing applications that are just variants of each other to test certain things. Often this can be done by adding a command line option to an existing application.

- You will next be asked (assuming you didn't tell the program that the new application was the same as an old one) if your individuals are fixed length character strings. This is fairly self–explanatory. This does not imply that your strings are binary however.

- If you answer "no" to the preceeding question, the program will ask you if your individuals are strings. This does not imply that they contain printable characters. There are some things that Gassy can help you with even if it doesn't know the length of your strings.

- If you answered "no" to the question about whether you have fixed length strings for individuals, you will now be asked if your individuals have a fixed size. For instance an individual might be a C struct, in which case you will answer yes. If Gassy knows that your individuals are all the same size it can do a lot of things to help – allocating space for the populations, copying individuals and so on. The skeleton file you will be asked to complete will contain a stub for a function that you will use to return the size of your individuals.

- Next, if you are using strings for individuals, the program will want to know if you allele values will be binary. If you say "yes", then your strings will contain printable 0 and 1 characters, *not* the integers 0 and 1 (which are not printable characters in any character set I know of). This will not make your program run more slowly. It will make it easier to print your strings and to debug your program.

- If you answer "no" to this question, the program will ask if your strings are printable via **printf()**. If so, Gassy can at least take care of printing your individuals. Gassy will arrange for your strings to be NULL terminated if they are of fixed length. Otherwise you should make sure of this.

  It is also possible that your strings are printable by **printf()**, but that you don't want them printed in such a simplistic fashion. If so, just answer "no" and the skeleton file will contain a stub that will ask that you print an individual. Of course if your individuals are not strings in the first place then the skeleton file is going to contain this function stub and you will not be asked this question by the program.

- If your strings are not printable, you might wish to control the way they are written to the file that is generated when an interrupt is received by Gassy. Gassy can do this for you no matter what your individuals contain, but if you want to look at the dump files, then perhaps you'd like them to be in a readable form. You will be asked whether you want to leave this job to Gassy or if you would like to do it yourself. If you decide to do it yourself, then you will also be responsible for reading the individuals back into memory from a dump file.

- Similarly, Gassy may have to read individuals from an initial population file. If the individuals are not printable or are not strings, you may have trouble editing these files. In this case it might be simpler to you to have some printable form of individuals stored in the initial population file and then to read it in yourself and convert it to the actual form of an individual. The program will ask you if you want to do this sort of thing if it knows that your individuals are not printable.

- If your individuals are strings of constant length, the program will now ask if all the alleles at each locus in each individual in the initial population have equal probability of occurrence. If this is true, Gassy can generate the initial population without assistance.

- The probabilities that are used to create the initial population could also be used by Gassy when it is doing mutation. However you may want to have a different probability distribution for mutation. You will be asked whether this is the case. If so, and the probabilities do not become equal, you will be given a chance to have them adjusted after the initial population is generated.

- If you indicate that the allele probabilities should change you will now be asked if they become equal. If so, it's Gassy to the rescue again. If not, you'll get to change them, as mentioned above.

- If your individuals are fixed length strings, Gassy will be able to do mutation for you automatically. Occasionally when mutating, an allele may mutate to the same value. If you do not want this to happen, you can tell **new_app** not to allow it and to always force a change in allele value. This is probably more biologically realistic, if that matters to you.

- Usually Gassy considers it an error if you specify a locus on the genome that only has one possible allele value. It is conceivable however that you one day might want to do this on purpose. For example, you might have intron-like regions of your genome that act as separators and do not contribute to the fitness of an individual. These loci may as well have just a single possible allele.

If you do want to do this, Gassy will allow it, and also will never attempt to mutate that locus.

Note that this is not the way *real* introns work!

- If you do say that you want to be allowed loci with only one possible allele, **new_app** will ask you if the number of these locations will be the same for every generation. If it is, Gassy will be able to do mutation for you faster than it could if it needed to recalculate the number of such possible locations (these locations do not get mutated). It is possible for an application to change the possible allele values during a run. Not many applications will need this functionality, but it is supported.

- You will now have a chance to request that Gassy allow for the possibility that you will be writing your own scaling function.

- Next you will be asked if you will be wanting to write your own survivor selector functions. The purpose of survivor selection functions is described on page 18.

- Next you will be asked if you want to write your own selection function(s).

- Then if you want to write your own crossover function(s).

- Lastly, at least in terms of writing your own functions, you will be asked if you wish to write your own mutation function(s).

- Next you will be asked if your application will have any command line arguments. Make sure you remember the difference between command line arguments and command line options (page 6).

- Next, will you be wanting to use command line options in your program. It is much more likely that you will want options than it is that you will want command line arguments.

- If you will be using command line options, Gassy can do quite a bit of checking on the values that are provided when the program is run. For instance, it can check whether mandatory options were actually specified and can compare the given values against legal lower and upper bounds. You get all of this automatically when you define your options.

  Gassy doesn't know about any inter–option dependencies that your program might have. If there are such and you would like to check that these conditions have been satisfied, answer "Yes" to this question and **new_app** will arrange for a function stub called **app_check_options()** to appear in the skeleton file. This will be called after Gassy has processed all the options.

- Gassy doesn't make any assumption about the type that is returned by your fitness function, except that it is numeric. I struggled for quite a while trying to even avoid committing Gassy to this seemingly harmless assumption. It is possible to imagine fitness functions that return, for example, a string. Eventually I gave up when it dawned on me that it was going to be very hard to write selection functions that expected to be able to add and divide etc., fitness values.

  So I bit the bullet and decided that all fitness functions must return something numeric. **new_app** now asks what type type is going to be. Then it knows how to print fitness values.

- Again on the subject of fitness values, Gassy needs to know what a very low fitness value is. When trying to find the best fitness whilst evaluating a population, it makes things easier and faster if you can initialize the best–fitness–so–far variable before you enter the loop. This avoids the two other solutions – duplicating code for the first case before the loop, or checking the loop index inside the loop. These are just details. In any case, **new_app** will now ask you for an extremely low fitness value. If you give a value that can never occur, then so much the better.

- The program will now ask if your fitness function is noisy. In most GAs this is not the case. Noisy means that the fitness assigned to an individual on one occasion may differ from that assigned to the same individual at some other point in time during the same run.

- Next you will be asked if you would like to evaluate the entire population in one function call. The simplest method of evaluating a population is to let Gassy call your evaluation function on each individual. If you do things this way, Gassy can take care of storing the fitness values, watching for and recording new record high fitnesses, computing the total fitness of the generation and so on. On the other hand, this will mean a function call is done for each individual in each generation. Over a long run, this could be millions of function calls that might otherwise be avoided.

  It is simple to arrange for Gassy to pass you the entire population. In fact, most of the code to do all the housekeeping described above is written for you, you just need to compute the fitness of all the individuals. This can be done without the extra function call and is really no more difficult for you. It will make your application seem more complex and your side of the code will be larger, but things will run somewhat faster.

  You will definitely want to choose this way of doing things if your fitness function examines the whole population before deciding what fitnesses to assign, or if your application wants to change the fitness function it uses in the middle of a run.

- Gassy has a hashing package built in that you can use fairly easily. Hashing is a way of storing keys and their associated values in such a way as to facilitate very fast expected lookup time. This can be used to great advantage in a genetic algorithm as one can store an individual and its fitness when it is evaluated the first time, and thereafter, the fitness value can be looked up instead of being recomputed. This can result in enormous savings - especially if your fitness function is expensive.

  Why hashing isn't used more commonly in GA applications, I don't know. It is very useful in the Busy Beaver application whose evaluation function simulates Turing Machines that may not halt. It is common for 60% of the machines evaluated during a run to be found in the hash table.

You'll now get to decide if the hashing package should be compiled into your application. If you choose to include it, you will be able to enable hashing from the command line.

- The next question concerns what happens when Gassy receives an interrupt or hangup signal. The main program knows how to save its state to a file that can be read in later to restart a run. But what of the poor application. If you have been faithfully compiling statistics each generation, you too will want a chance to dump your state when an interrupt is received. If you tell the program that you want a chance to do this, it will arrange for the skeleton program to contain two function stubs that will be called at the appropriate times.

- Now you will be asked if there are things you would like to do each generation. If so, a function stub called **app_generation** will be generated and that function will be called after each generation has been evaluated. It will have access to all the details of the population and the fitnesses etc.

- If you are keeping track of some statistics, you'll probably want a chance to print them at the end of the run. **New_app** will ask you about this now.

- At the other end of things, your application may need to do some initialization before the run is started. For instance, you might want to grab some memory to store statistics. If you have needs such as this, a function stub called **app_initialize** will be generated for you.

- All Gassy options have an option named −**usage** which produces a message that describes the various options that are available, their abbreviations, their legal limits, their default values (if any) and so on. If you like, the usage message can contain longer explanations of the effects of the various options. If you already know what the options do and you don't anticipate others who may not using your application, you can have a terse version of the text appear instead. The difference in program size is about 8K.

- Next you will be asked if you want the optional **TYPES** file to be generated for you. This was described at the start of this section.

- The final question asks you if you want the skeleton file to contain comments that will help you in filling in the stub functions. This was also described above and, in case you have forgotten, it's a *very* good idea to say yes. Even if you are an experienced Gassy user, you will probably want to answer yes if you are creating a new application with some features enabled that you have not used before. Comments can always be deleted later if you don't need them.

5.3 The Rest Of Your Application

5.3.1 Types

5.3.2 Conditional Compilation

5.3.3 Adding Command Line Options

5.3.4 Adding New Functions

Introduction

Adding Scaling Functions

Adding Survivor Selection Functions

Adding Selection Functions

Adding Crossover Functions

Adding Mutation Functions

5.3.5 Hashing

5.3.6 Utility Functions and Macros You May Use

Random Numbers

Normally Distributed Numbers

Gray Coding

Integers As Binary Character Strings

5.4 Modifying An Application

Chapter 6

Running An Application

6.1 Command Line Options

All Gassy applications can use any of the forty or so options that are built in to the main package. Applications are free to add their own, as was discussed on page 41.

This section describes all the options that are built in to the main package. The information contained here is the same as you'll see if you use the –**usage** option when invoking any application that was compiled with the verbose usage messages (this is a choice made by the application writer when running **new_app**. See page 39).

–**crossover_method** This option can be abbreviated to –**cm**. Specify, by name, the crossover method. There are several built in crossover methods, including **no_crossover**, **one_point**, **two_point** and **uniform**. If you write your own crossover function you will have to give it a name in your **app_crossover_functions** variable and you will ask Gassy to use it with this option, just like any built in method. Crossover function names may be abbreviated. To see a full list of the crossover methods available for any given application, invoke the application with the –**functions** option. The default value for this option is **two_point**.

–**crossover_probability** This option can be abbreviated to –**cp**. This option has a lower limit of 0.0 and an upper limit of 1.0. Specify, with a 0.0 to 1.0 argument, the probability of crossover occurring. Each time crossover is to be performed, a randomly chosen value from 0.0 to 1.0 is chosen and the crossover is done if that value is less than the one you specify with this option. The default value for this option is **0.75**.

–**data_only** This option can be abbreviated to –**do**. The output produced by the **show_*** options, contains words that make the file's contents understandable. If you use this option, the words will be omitted from these files where it makes sense to do so. It "makes sense" if the contents of the file, without the accompanying text, would be suitable as input to a plotting program. Thus the file generated by **show_best** will just have 2 numbers per line (generation number and best fitness) if the **data_only** option is used. In other cases, for example in a file that contains sorted populations for each generation, where it does not make much sense to leave out the text, this option is ignored. The default value for this option is **FALSE**.

–**dump_final_population** This option can be abbreviated to –**df**. Dump the final population and other relevant information to a file that Gassy can be restarted from at a later date. The name of the file will be the name of your main program with **.final.XXXXXX** appended, where **XXXXXX** is some six digit number (the next one available). The file format is the same as that for dumping when an interrupt is received. This is very useful when you want to run an experiment for more generations than you originally thought. Of course it helps to know this in advance :-).

–**elitist_selection** This option can be abbreviated to –**e**. Use elitist selection (save the best individual of each generation). This option may not be available with all forms of selection. Of the builtin selection functions, it is available with tournament selection but not roulette wheel fitness proportionate selection.

–**functions** This option can be abbreviated to –**fun**. This option will result in a help message describing all the functions that are selectable from the command line to control things like selection, generation gap survival, crossover, mutation and so on.

–**generations** This option can be abbreviated to –**g**. This option must be specified for every run. This option has a lower limit of 1. Specify, with the next argument, the number of generations to run. This is more like the maximum number of generations to run if you have asked the program to stop at some maximum fitness value.

–**generation_gap** This option can be abbreviated to –**gap**. This option has a lower limit of 0.0 and an upper limit of 1.0. Specify, with the next argument, generation gap. This determines what proportion of the population will be replaced from generation to generation. A value of 1.0 means that the whole population is to be replaced. 0.1 would mean that 10% of the population is to be replaced etc. The default value for this option is **1.0**.

–**hashing** This option can be abbreviated to –**h**. Turn off hashing. Hashing can only be turned off when the hashing code was compiled into your program. You may object that this option should even exist – if you wanted hashing compiled, you probably want to use it always. But sometimes you can run out of memory if the hash table contains too much and so it is nice to be able to turn hashing off.

–**help** This option can be abbreviated to –**he**. Print a help message describing all options and their effects.

–**initial_population_file** This option can be abbreviated to –**i**. Specify, with the next argument, a file containing an initial population. If this file does not contain enough individuals to fill the population, random ones will be created to make up the difference. If it contains too many, you will be warned but the run will continue. If your individuals do not have constant size, then each should be preceded by its size on a line by itself.

–**interrupts** This option can be abbreviated to –**in**. Enables interrupt catching and state dumping. If used, Gassy will, on receiving SIGINT or SIGHUP, write out the state of the run to a file whose name is the name of your main program with **.interrupt.XXXXXX** appended, where **XXXXXX** is some six digit number (the next one available).

–**maximum_fitness** This option can be abbreviated to –**mf**. Specify, with the next argument, the maximum value of your fitness function. There will be many occasions when you will not know the maximum possible fitness value in advance, but if you do, this option can be used to specify that value. If the program finds an individual with that fitness (or higher!), it will cut short the run.

–**mutation_method** This option can be abbreviated to –**mm**. Specify, by name, the mutation method. There is currently only one built in mutation method: **simple_mutation**. If you write your own mutation function you will have to give it a name in your **app_mutation_functions** variable and you will ask Gassy to use it with this option, just like any built in method. To see a full list of the mutation methods available for any given application, invoke the application with the –**functions** option. Mutation method names may be abbreviated. The default value for this option is **simple_mutation**.

–**mutation_probability** This option can be abbreviated to –**mp**. This option has a lower limit of 0.0 and an upper limit of 1.0. Specify, with a 0.0 to 1.0 argument, the per bit mutation probability. Mutation on constant length character string individuals will be done automatically by Gassy – using the allele value and frequencies you can specify. The default value for this option is **0.001**.

–**options** This option can be abbreviated to –**op**. Specify, with the next argument, a file containing more options. Options in files should be specified one per line in the format **option–name : option–value** (there need not be a value if the option does not take an argument, see below). The option names and abbreviations are the same as the command line option names. Files of options can specify other files of options, which is very useful if you are doing a number of experiments in which many options remain the same – they can each have an experiment–specific options file that includes a file of options common to all the experiments.

Options are processed left to right as they are found on the command line, and the given value for an option will reset the former value. This can be used to override options set in a default common option file.

Boolean options may be given with no argument (and the colon may be omitted too), in which case their value will be toggled from TRUE to FALSE, or vice–versa, each time they are encountered. Alternately, they may be given arguments such as "true" or "false", or "yes" or "no", or "on" or "off", or even "1" or "0" – take your pick.

Blank lines or lines starting with # in options files will be ignored, so you can comment your option files.

–**overwrite** This option can be abbreviated to –**ov**. If this is set to TRUE, then files that are given as arguments to the **show_*** options will be overwritten instead of appended to. The default value for this option is **FALSE**.

–**output_file** This option can be abbreviated to –**o**. Specify, with the next argument, a file into which all output should go. This option is actually not needed if you are using some version of UNIX as all output goes to standard output and can be redirected from the shell (if you using some version of UNIX).

–**population_size** This option can be abbreviated to –**p**. This option must be specified for every run. This option has a lower limit of 1. Specify, with the next argument, the population size for the run.

–**report_interval** This option can be abbreviated to –**ri**. The number of generations between the printing of the current generation. This only takes effect if you are using one of the **show_population*** options to have the populations printed. The default value for this option is **1**.

–**restore_file** This option can be abbreviated to –**re**. Specify, with the next argument, a file to restore a run from. The program will read in the state of the world and start it running again. If output was previously being redirected to some files, then the program will attempt to append to those files.

–**seed** This option can be abbreviated to –**r**. This option has a lower limit of 0. Specify, with the next argument, a seed for the random number generator. This can be used to repeat a run or to repeat and extend a run for a longer time if you specify more generations.

–**selection_method** This option can be abbreviated to –**sm**. Specify, by name, the crossover method. There are currently two built in selection methods: tournament and roulette. If you write your own selection function you will have to give it a name in your **app_selection** variable and you will ask Gassy to use it with this option, just like any built in method. To see a full list of the selection methods available for any given application, invoke the application with the –**functions** option. Selection function names may be abbreviated. The default value for this option is **tournament**.

–**survivor_selection_method** This option can be abbreviated to –**ssm**. Specify, by name, the survivor method. There is currently only one built in survivor method: **random_survivors**. If you write your own survivor function you will have to give it a name in your **app_survivor_functions** variable and you will ask Gassy to use it with this option, just like any built in method. To see a full list of the survival selection methods available for any given application, invoke the application with the –**functions** option. Survivor selection function names may be abbreviated. The default value for this option is **random_survivors**.

–**show_averages** This option can be abbreviated to –**sa**. Have the average population fitness for each generation printed. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_best** This option can be abbreviated to –**sb**. Have the fitness of the best individual of each generation printed. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_final** This option can be abbreviated to –**sf**. Show the final population. The population will be sorted and duplicate genome individuals will be reduced to a single line of output followed by a count of their multiplicity. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_hashing** This option can be abbreviated to –**sh**. Have statistics on the final contents of the hash table printed once the entire run has completed. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_hashing_verbose** This option can be abbreviated to –**shv**. Have statistics on the final contents of the hash table printed once the entire run has completed. This option also shows the contents of every non–empty bucket. This can generate a LOT of out put (thousands of lines). If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_intermediates** This option can be abbreviated to –**si**. Have the program output the new highest fitness so far individual each time it finds one. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_populations** This option can be abbreviated to –**sp**. Print the population at the end of each generation. The population will not be sorted in any way, for this use one of the next options. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_populations_sorted_by_g** This option can be abbreviated to –**spsg**. Print the sorted population at the end of each generation. The population will be sorted by genome. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_populations_sorted_by_f** This option can be abbreviated to –**spsf**. Print the sorted population at the end of each generation. The population will be sorted by fitness. If this option is given an argument, it will be taken as the name of a file to append the output to. If the file does not exist, it will be created. If the –**overwrite** option has been specified and the file exists, it's contents will be overwritten. If no filename argument is given, the output will go to **stdout**.

–**show_start** This option can be abbreviated to –**start**. This option has a lower limit of 1. Set the generation at which functions that show population statistics should begin to be called. The default value for this option is **1**.

–**show_begin** This option can be abbreviated to –**begin**. This option is a synonym for **show_start**.

–**show_stop** This option can be abbreviated to –**stop**. This option has a lower limit of 0. Set the generation after which functions that show population statistics should not be called. A value of zero means that the functions should be called through the final generation. The default value for this option is **0**.

–**show_totals** This option can be abbreviated to –**st**. Show the total fitness of the population at each generation once the entire run has completed.

–**top_ranking** This option can be abbreviated to –**tr**. This option has a lower limit of 0. Specify how many of a population to print. The top this many ranks will be shown. This only takes effect if you are using one of the **show_population*** options to have the populations printed. A value of zero means to show the whole population. The default value for this option is **0**.

–**tournament_probability** This option can be abbreviated to –**tp**. This option has a lower limit of 0.0 and an upper limit of 1.0. Specify, with the next argument, the tournament selection probability. When deciding whether to select the more or less highly fit individual in a tournament, the more highly fit one will be chosen if a randomly generated number between 0.0 and 1.0 is less than the value you specify with this option. The default value for this option is **0.75**.

–**unique** This option can be abbreviated to –**uni**. Controls whether population printing will be followed by a count of the number of unique individuals.

–**usage** This option can be abbreviated to –**us**. Print a help message describing all options and their effects.

–**uniform_probability** This option can be abbreviated to –**up**. This option has a lower limit of 0.0 and an upper limit of 1.0. Specify, with the next argument, the uniform crossover probability. When uniform crossover is being used, this probability will determine how likely it is that bits will be swapped in the offspring. The bits will be exchanged if a randomly generated number between 0.0 and 1.0 is less than the value you specify with this option. The default value for this option is **0.5**.

–**verbose** This option can be abbreviated to –**v**. Turn on all of the **show_*** options, except the **show_population*** ones. That is, **show_averages**, **show_best**, **show_hashing**, **show_intermediates**, **show_final** and **show_totals**. Note that, unless already specified with an argument, the output of each of these will go to **stdout**.

–**version** Show the Gassy version number that was used to make this program.

–**?** Print a help message describing all options and their effects.

6.2 Command Line Arguments

Gassy does not use any command line arguments. Recall from the **Terminology** section (page 6) the difference between options and arguments. Normally anything found on the command line following the options will generate an error message. An application may make use of command line arguments if it chooses. This is most easily accomplished by telling **new_app** that this is desired.

6.3 Displaying Results

6.3.1 Intermediate Results

6.3.2 Final Results

6.4 Dealing With Interrupts

6.5 Restarting Interrupted Runs

Chapter 7

Getting Help

7.1 Getting Help At Runtime

7.2 Reporting Bugs

Chapter 8

Examples of Gassy Use

8.1 A Simple Example

8.2 Maximizing Ones

8.3 Faster Ones Maximization

8.4 Spelling My Name

8.5 An Arithmetic Problem

8.6 Busy Beavers

Chapter 9

Obtaining Gassy

9.1 Obtaining Gassy Via Ftp

The entire Gassy package can be obtained via ftp from moocow.cogsci.indiana.edu in the file **pub/gassy-3.0.tar.Z**. To retrieve this file and unpack it, you should do the following[1]:

$ **ftp moocow.cogsci.indiana.edu**
Connected to moocow.cogsci.indiana.edu.
220 moocow FTP server (SunOS 4.1) ready.
Name (moocow.cogsci.indiana.edu:terry): **anonymous**
331 Guest login ok, send ident as password.
Password: **¡Enter your e–mail address¿**
230 Guest login ok, access restrictions apply.
ftp¿ **cd pub**
250 CWD command successful.
ftp¿ **binary**
200 Type set to I.
ftp¿ **get gassy-3.0.tar.Z**
200 PORT command successful.
150 Binary data connection for gassy-3.0.tar.Z (192.12.12.11,1220)
(60937 bytes).
226 Binary Transfer complete.

[1]The text in **bold** is what you enter.

local: gassy-3.0.tar.Z remote: gassy-3.0.tar.Z
60937 bytes received in 9.9 seconds (6 Kbytes/s)
ftp¿ **quit**
221 Goodbye.
$
$ **uncompress gassy-3.0.tar.Z**
$ **tar xfv gassy-3.0.tar**
$ **rm gassy-3.0.tar**

This will result in a directory called **gassy-3.0** being created in the directory you did the **tar** command in. In that directory you will find all the source code for the Gassy system, several example applications and this document in both LaTeXand postscript formats. The **README** file is probably a good place to look first if you are unsure about where to find things.

9.2 Obtaining Gassy Via Electronic Mail

Chapter 10

Planned Additions To Gassy

I have many plans for the future of Gassy. In no particular order, these are:

- **BEFORE RELEASE**
  Document GASSY_DIR and change everything to use it.

- **BEFORE RELEASE**
  Make an env variable to point to the main gassy source location. This will allow the construction of applications that do not have to reside below the source dir. Move the examples to an examples directory. Fix **new_app** to look fro the variable and ask if it is not defined. And fix it to make the makefile right.

- **BEFORE RELEASE**
  Make the file all-possible-defines a part of the distribution. Or put it into the tex document. That way people can see what all the defines do and they will have somewhere to go for help when all else fails or when they want to find out more details about what they can define in their application.

- **BEFORE RELEASE**
  What if allele probs are equal for the initial population and different for mutation? Is this change supported? I don't remember. If not, then make it so.

- Adding various forms of scaling.

- Devise and implement a way to make some variables dependent on some functions. For instance, the tournament selection probability should not be around unless tournament selection is being done. a function should have a list of variables associated with it. When the function is seen on the command line, the variables should be added to the possible variables list. Something like that.

- Add new selection methods, such as rank selection.

- Make it possible for users to provide their own hashing function. Some applications will not have string genomes and so they will not be able to use the standard hashing stuff. Also in the case they will ne to provide a function which returns TRUE if two passed individuals are the same. So that we can look in a hash bucket and decide if something in there is what we seek. Also, what about the type of struct that is placed in the hash table? That must be alterable.

- Add some provision for implementing steady state GAs.

- **BEFORE RELEASE**
  Provide more miscellaneous utility functions.

- **BEFORE RELEASE**
  Add a method for simple construction of hill climbing programs. These can then be used to compare the GA's performance with hill climbing. It shouldn't be too hard to do this, it is mainly a matter of just calling the application's evaluation function and providing all the normal bit (if they are bits) flipping.

- **BEFORE RELEASE**
  Add the possibility of doing some hill climbing to the GA package itself. For instance, an option might allow each member of the population to hillclimb at the end of the generation. Or just the best n. This needs to be well thought out. Could or would the package then become less GA specific and more a mix – you could turn off GA processing and just hillclimb. Or vice–versa, or else do a hybrid, etc. etc.

- Add new stopping criteria, such as stop after a certain time, stop after a certain number of trials, and even stop after looking at a certain percentage of the space. That would be interesting. Time and trial stopping are simple. Allow for the use of more than one of these at a time and stop for the one that becomes true first. Space percentage can be done without too much trouble if the genome is a string. In this case you simply calculate the size of the space by multiplying all the allele multiplicities. What if the space is too big to be calculated?

- **BEFORE RELEASE**
  Make Gassy handle multiple runs with the same parameters. In fact add a new function that will give an application a chance to change parameters if it so desires between runs. Should this be done this way or should we let people write shell scripts? Probably the former.

- **BEFORE RELEASE**
  Add some provision for negative fitness being good. Watch for logs of negative fitnesses.

- **BEFORE RELEASE**
  Write **app_change** (or similar) to allow for easy changing of an applications characteristics. This is **not** easy.

- **BEFORE RELEASE**
  Get rid of terminator in all the lists of things and use sizeof to determine the number of items they have.

- **BEFORE RELEASE**
  Fix dumping / restoring.

- **BEFORE RELEASE**
  Add schema tracing.

- **BEFORE RELEASE**
  Make the TYPES file produced by **new_app** look better. Write an awk script to take the blank lines out of struct definitions.

- **BEFORE RELEASE**
  Find the Aviv mutation bug.

- **BEFORE RELEASE**
  Make hashing happen automatically if the individuals are strings. Move calculation of hash hits and misses into the main program in this case. Add a function to alert the application if the fitness of an individual has been retrieved from the hash table (if the application wants). Add this question to **new_app**.

- Perhaps rewrite **new_app** as a C program to remove the need for a shell with functions. This will increase portability to the VMS and PC worlds.

- Finish this documentation...

Bibliography

[1] Goldberg, D. [1989], "Genetic Algorithms in Search, Optimization and Machine Learning" Addison Wesley, 1989.

[2] Kernighan, B. W. and Ritchie, D. M. "The C Programming Language", Second Edition.