Design and Implementation of Genetic Algorithms for Solving Problems in the Biomedical Sciences

Running title: "Genetic Algorithms for Biomedicine"

Michael Levin
Genetics Dept.
Harvard Medical School
200 Longwood Ave.
Boston, MA 02115

Abstract

Many problems in the biomedical sciences can be re-formulated as searches in some appropriate space. Genetic Algorithms (GAs) are a domain-independent form of search which has several characteristics enabling it to effectively search difficult spaces. Thus, the evolutionarily-inspired GA is often able to provide good answers to questions with very difficult search spaces. This paper presents a tutorial on when to use, and when not to use GA approaches, how to model problems as searches, how to define appropriate spaces, solution representations, and fitness functions, and how to implement and trouble-shoot a GA program which will efficiently locate solutions to the problem. A list of resources such as public-domain (freely-available) GA software and various internet-based GA advice groups is also provided.

Introduction

All interesting questions, including those of interest to the biomedical community, can be reformulated as searches in some appropriate space. For example, when one wants to design a molecule that binds some particular protein, one is searching for some element (or elements, or even subspace) in the space of all possible molecules. When one wants to optimally allocate hospital resources among some group of patients, one is searching for a satisfactory allocation strategy in the space of all possible strategies. When one is trying to develop a theory or a model that explains some set of phenomena, one is searching the space of all possible theories for the simplest one which fits the given data.

Any search task has several components. One needs to define a search space (the abstract form or structure of all possible entities that are being searched through), and an evaluation function (a precise way to evaluate any member of this search space and decide on its "quality" - how good or useful a solution it is). In the above examples, and in general, the space that needs to be searched is immense, and often of unknown dimentionality (for example, how many independent parameters are sufficient to enumerate all possible theories?). Several methods exist for effectively searching these kinds of spaces. Scientists commonly use heuristics and creative insight to achieve this. This process, however, is highly non-algorithmic, and thus difficult to simulate on the computer.

Because of the search space size, exhaustive searches (ones which cause every possible solution to be examined) are often not feasible, no matter what computing resources one has[1]. And, while several common heuristic methods exist, they are often ineffective on deceptive or difficult spaces. This refers to spaces with strong nonlinearities, such that elements that are close together in the space have widely differing qualities. For example, when a molecule that is pretty good at catalyzing some biochemical activity, there will often be almost identical (and

[1] While not at all obvious, this follows from certain results in the theory of computation and physics.

thus in the search space, neighboring) molecules that are very poor at this task because of single residues which cause steric hindrances etc. Such spaces are deceptive because one point in the space doesn't always give much information about its neighbors. This can cause gradient methods to become trapped in local maxima (regions of good solutions which are far away from sets of much better solutions and thus form dead-ends for the search).

Fortunately, there is another method which is often very useful at searching such spaces. Genetic Algorithms (GAs [1]) and their relatives in the wide field of Evolutionary Computation can endow a search with the power of evolution. The basic idea, analogously with biological evolution, is that an initially random set of candidate solutions to some problem is evaluated. The top few percent are kept, while the rest are discarded. These top few are recombined and randomly mutated to form the next generation's set. This process is iterated until a satisfactory solution is found. Note that this procedure is completely algorithmical (and thus easily given to computer implementations), and requires no domain-specific knowledge or user intervention - the search proceeds using only the information inherent in the search space itself. Likewise, the algorithm is domain-independent, and can be adapted to almost any task.

At this point one is naturally incredulous. How is a random set of points, driven by what looks like a randomized hill-climbing search algorithm, going to be improved in a difficult and immense search space? There are two answers to this objection. The empirical answer is that it seems to work, and very effectively indeed. References 3-8 give many examples where this type of algorithm has worked on various difficult tasks.

The theoretical answer is motivated by biological evolution. One is hard-pressed to come up with a more difficult search space than that of biological phenotypes. Due to genetic pleitropy, and the fantastically complex interactions that go on during embryonic development, it hardly seems plausible that random mu-

tation and differential reproduction are sufficient to account for the myriad of marvelously-adapted organisms we observe today. At all levels, from biochemicals to organs and whole organisms, each system seems so finely tuned that any random alteration will ruin its function completely (these matters are dealt with in [2]). Yet evolution obviously works, and the basic algorithm outlined above is seen to be sufficient to successfully traverse an extremely difficult search space[2]. By pursuing a population of solutions simultaneously, letting useful features propagate through selection, and recombining parts of solutions, GAs are often able to effectively search complex and many-dimensional search spaces and avoid being trapped on local maxima.

GAs however, are not a panacea. It is important to decide whether a GA approach is right for a given problem. In general, GAs should be used when:

- the space of all possible solutions is very large;
- the solution space is high-dimensional (i.e., a solution to the problem can consist of a large number of parameters, each of which can be set independently of the others, thus providing a combinatorial explosion of possibilities);
- the space is deceptive (i.e., solutions which are very similar in form and content are not guaranteed to have similar quality as solutions);
- the problem contains non-linearities and constraints;
- there is no known analytical way to solve the problem;

GAs should not be used when:

- a closed-form or analytic solution is possible (for example, when it is possible to write down a set of equations which capture the constraints of your problem, and to solve them explicitly);

[2] One of the main reasons why GAs are so important, besides their usefulness in solving various problems, is that they allow us to adjust our intuitions as to what is and is not possible for the process of evolution.

- when an exhaustive search is practical (for a small section of seemingly-difficult problems, a careful choice of hardware and fitness function may make it possible to try out every solution in the time it would take to design, code, and run a genetic algorithm;
- some other paradigm (such as an Artificial Neural Network, a classical AI expert system, etc.) is more appropriate;
- when repeatability is required (due to its stochastic nature, GAs are by definition not guaranteed to find the same solution in any two runs);
- real-time solutions are needed (GAs, like biological evolution, require time-scales which are often much greater than direct methods).

Implementation

Supposing that one has decided to use a GA approach, several issues need to be decided upon. The choice of hardware is often constrained to what is already available; however, almost all GA approaches require at least a fast workstation (Unix or VMS) to be useful. In certain extreme cases, a massively parallel architecture is needed (access to such machines can be obtained at several national supercomputing centers, such as the PSC). Memory, disk space, and graphics system requirements are generally dictated by the specific problem at hand. One generally wants the fastest CPU available, and often a separate FPU (floating point processor) can speed things up significantly. The GA itself does not require any extra disk space, nor much memory; a graphics system is beneficial (for plotting the course of the evolution) but not required. Any common language (BASIC, C, C++, Pascal, Fortran, LISP, etc.) may be used, but the nature of the specific task may make one more suitable than another.

The basic GA algorithm is diagrammed in figure 1; it faithfully represents the basic idea behind "Darwinian" evolution (except for side-stepping the question of

the origin of the first unit capable of self-reproduction). In order to implement this algorithm, several things need to be defined:

1) a representation: one needs to decide on a structure [3] which is able to represent every possible solution to the desired problem. Some examples: if one wants to fit a 4-th degree polynomial to some points of data, one can represent each candidate solution as a vector of 4 floating point numbers, where each number is a coefficient of the polynomial; if one wants to fit a curve to points of data but does not know *a priori* the form of the curve, the representation can be a regular expression (such as "x * (3 + x) - sin(x)") using a number of functions, variables, and constants; if one wants to find an optimal classification strategy, the representation can be a tree, whose nodes are questions, and whose branches represent the possible answers to those questions, which then lead to more specific questions; if one wants to be able to do visual recognition of malignant cell types, the representation can be a matrix of numbers which define an artificial neural net which can perform the recognition; etc. A single instance of such a representation is called a "member" of the population.

The choice of representation is important; it must not be too narrow (or good solutions will be missed), nor too wide (or else time will be wasted in searching inappropriate parts of the state space). The landscape should be well correlated: as much as possible, representations which are close in content must represent solutions whose quality or fitness is also close. For example, if one was searching for an animal which matched certain characteristics, it would be a poor choice to arrange the data in order of alphabetical name. This is because animals whose names are alphabetically close (worms and wombats) will not tend to have similar characteristics. A much better choice in this case would be a tree-like arrangement which preserves phylogenetic relationships, because then close points on this tree would represent animals which are likely to be similar. This makes for a non-

[3] Some common choices include artificial neural networks, finite state automata, decision trees, regular expressions, logic truth tables, graphs, etc.

deceptive landscape, and maximizes the effectiveness of a GA approach. Finally, the choice of representation may suggest one computer language over another. For example, LISP is usually preferred when one needs to deal with lists and variable length trees, while C is simpler for fixed-shape arrays etc.

2) a fitness function: the next crucial ingredient is to define a fitness function. This is a function which takes a single member as input, and returns a real number between 0.0 and 1.0. This function must evaluate the member it is given, decide how good a solution it is for the problem at hand, and return a single number expressing this result; numbers closer to 1.0 represent better solutions. Thus, when a fitness function is defined, such that it is possible to perform an unambiguous ordering of members with respect to it, the GA search performs an optimization of that function.

Some examples: if one is searching for an equation to fit some data, the fitness function should return the quality of that fit (perhaps measured by the least-squares method); if one is searching for a classification strategy, the fitness function should run a bunch of test cases through the strategy it is given, and arrive at a score expressing how good a job this strategy did; if one is searching for a pattern recognizer, the fitness function should submit some number of positive and negative instances to the recognizer determined by the genotype, and score its performance; etc.

Clearly, this function is application-specific. Usually, this function (which is called repeatedly to evaluate every member of the population, in each generation) is where most of the compute-time is spent, so most optimization effort should be directed at it. The fitness function should be scaled, so that 1.0 represents a perfect solution, and so that it is as linear as possible (i.e., avoid functions which give all-or-nothing grades; partial credit works best).

3) mutation and recombination operators: for evolution to take place, one must have a means for producing new genomes. There are two types of operators: mu-

tation and crossover (although sometimes, more exotic operators such as inversion, duplication, deletion, etc. can be used). Mutation randomly alters some aspect of a member, while cross-over performs an exchange of genetic material (of the member's information) between two members. For example: if a representation is a vector of numbers, mutation can simply change one or more of those numbers by some small offset, while crossover swaps one or more numbers between two individuals; if a representation is a decision or parse tree, mutation can replace nodes or cut or add branches, while crossover can swap branches at defined points; etc.

An important issue in designing operators is what to do about the fact that members can be produced which are illegal, and cannot be evaluated. For example, the expression "3 ÷ x" can be mutated to "3 ÷ 0", which cannot be evaluated. Four strategies are possible for dealing with this:

- protected operators - define special functions (in this case, a protected divide function) which watch for illegal inputs, and return pre-defined values (in this case, 0 or 3 are both reasonable choices);
- "hand of God" - one can have a special routine that scans for and removes illegal individuals prior to evaluation;
- "dog-eat-dog" - one can simply assign a fitness of 0.0 to illegal members, without evaluating them at all - they will then tend to die out;
- "Lamark's revenge" - one can define smart mutation and crossover operators that are context-sensitive[4] and never produce illegal members;

Which of these strategies is chosen depends on the representation. If there is a natural way to exclude illegal members when mutating, the last method is preferable. If not, the first is probably best.

[4] i.e., they are not really random because they take the current genotype into account when changing it.

4) GA parameters: besides all of the parameters of the specific task, a GA has a set of its own parameters, all of which can be tuned to achieve better performance. The particular values are application specific (and thus are to be determined by experiment), but some hints can be given. These parameters include:

- population size (P) - the bigger the population, the more chance of finding a good solution. Population sizes are generally on the order of between 100 and 1000. Populations that are too small are likely to miss good solutions through premature convergence on sub-optimal solutions, while ones which are too big can waste time.
- survival size (S) - this determines what percent of the population survives at each generation. Thirty percent is a decent number. If this value is made too small, the population will quickly lose solutions which are not so great now, but have a chance of being greatly improved, in favor of things which look good now but may turn out to be dead ends. If the value is too large, the GA will waste time on poor candidates.
- mutation rate (M) - this number determines the likely-hood of mutation in each generation. A good general scheme is as follows: at each generation, after evaluating all individuals, the next generation's population is made up of: unaltered copies of the best S individuals [5], and as many mutated (M times each) individuals as there are room for in P.
- crossover (C) - this number determines how many individuals out of the rest described in the point above are not straight mutants but represent cross-overs between two randomly-chosen individuals in the top S of the population. Using crossover generally leads to faster convergence. This may be a good thing or a bad thing, depending on the task and how deceptive the landscape is.

[5] This is called elitist selection, and ensures that the best individuals are never lost.

In general, when one implements such an algorithm according to the flowchart in figure 1, one should use a graphical package to plot (or at least record to a data file for later study) several important characteristics of the evolution. These include the fitness of the top individual in the population, the average fitness, and the population convergence (some measure of how similar the members of the population are). These should usually be plotted as a function of generation number, though sometimes (for study as a time-series) these values are plotted against their own values at the previous generation.

These kinds of plots should be studied because they carry information showing how fast better solutions are (or are not) being found, and when the evolution can be stopped. The shape of the fitness curve is usually linear at first, and then exhibits gradual flattening, in a manner reminiscent of punctuated equilibrium (a sample plot appears as figure 2). It is also a good idea to implement some kind of event handler which will save the current population to a disk file. This is because it can be very frustrating to lose the results of a 12-hour evolution when some bug in the program causes a crash.

Several tricks exist for improving the performance of GAs. The details are outside of the scope of this introductory paper, but they can be found in [3-8]. One standard approach is to reward parsimony along with performance. That is, when calculating fitness, most of the value depends on how the member performs as a solution to the problem, but some percentage of it is a function of how simple the member is. For example, when evolving variable-length decision trees or expressions, it is often useful to reward parsimony, thus selecting for members which are functionally the same but simpler in form (for example, preferring "3" to "3 + 2*0"). This is beneficial both in terms of later use of the result discovered, and time saved by not evaluating useless pieces of the members. This should be used with caution, however, since often such seemingly useless sections can have useful intron functions, by protecting critical pieces of code against cross-over, and by serving as place-holders which are later replaced by useful code.

Once a GA implementation is up and running, it is important to determine how well it is working, and to adjust things if it is not working well. One of the simplest analyses involves observing the top fitness vs. generation time plot. It should rise rapidly at first. If it begins to level out (such that no significant progress in fitness is being made) while the quality of the best solution is really low, then there is a problem. Often it is a bug in the code, although the reverse is not true: fitness can often rise nicely even with significant bugs in the code, because the opportunistic evolution will take advantage of mistakes in the fitness function, and evolve things that were not intended to be selected for.

If a bug in the code is suspected, one way to narrow down its location is to replace the fitness function (and nothing else) with one which is known to be easily optimizable using GAs (such as a simple polynomial minimization problem or something similar). If this works, it then follows that the GA engine itself is working, and the problem is either a bug in the fitness function itself, that the problem is simply too difficult, or that a poor representation has been chosen. If it does not work, one should strongly suspect the GA code itself, and to use some debugger to make sure that mutations are being done properly, and that the top S% of the population is really being carried over to the next generation. Other kinds of statistics (such as an averaged measure of how much mutated offspring differed from their parents) etc. can be useful in narrowing down the bug.

Before deciding that a problem is too difficult, it often pays to reconsider the representation, and to think about other ways of representing the problem and possible solutions. Perhaps there is a better way to produce a correlated landscape, thus increasing the chances that a GA approach will work. One other reason for premature flattening of the fitness curve is premature convergence on sub-optimal solutions. This can be detected by computing and plotting a convergence measure[6], and can sometimes be ameliorated by getting rid of cross-over (using muta-

[6] This is a scalar number which is determined by how similar all of the genotypes in the population are.

tion only), and by using larger population sizes. Likewise, if the fitness is rising too slowly, it may be worthwhile to experiment with GA parameters such as population size and mutation rate. Finally, the problem may really be difficult, and the only remedy may be patience (and a faster computer).

An Example: Antisense Therapy Advice Tool

With those general principles in mind, it is now possible to turn to a sample GA, applied to a real problem: the problem of designing antisense oligonucleotides in antisense therapy[7].

Much research has shown ([9-10]) that introducing into cells oligonucleotides whose sequence is complementary to the sequence of an expressed gene can often greatly inhibit the presence of the protein. This can be (and is being) used as antisense therapy, to knock out the function of certain genes of interest to the biomedical community. The first step in this kind of approach is to design an oligonucleotide which matches some part of the sequence of the gene of interest. This choice is subject to a number of constraints:

- The oligo should be long, for maximal specificity and binding avidity, but if it gets too long, cell uptake will not be good;
- The oligo should span a strategic point in the gene (the translation initiator region, a splice site, etc.);
- The oligo should have certain composition characteristics (for example, > 50% GC-rich);
- The oligo should not have more than triplet repeats;
- The oligo should avoid other sequences which would result in unusual secondary structure; etc.

[7] This is a good example because it lends itself well to a GA approach; it is not meant to suggest that it is the only, or even the best, way of dealing with this specific problem.

A GA approach is a good idea for this kind of task[8] because:

- time is not critical because the time to design an oligo is negligible compared to the time that will be spent investigating its effects; also this is a procedure that will only be done a few times at the most, not repeatedly;
- the problem has complex, interacting constraints, which are difficult for a human designer to take into account by inspection;
- the space of all possible oligos matching a given gene is very large (though for smaller genes and fast CPUs, an exhaustive search just may be practical). For example, considering 14-mer oligos somewhere along a gene 2 kb in size, there are on the order of 10^{11} possible oligos.
- the fitness landscape is difficult (because even single base-pair substitutions can introduce secondary structure and make a good oligo choice be unsuitable), but still correlated, because in general, single base-pair differences do produce oligos that are similar in predicted quality.

Thus, the first task it to choose an appropriate representation. Since any oligo can be represented by an array of characters (A,C,G,T) of some length, the representation will consist of a structure containing two elements: a variable-length character array, and an integer expressing the starting position of the binding region on the target DNA. Since the representation involves simple linear arrays of characters, this algorithm can be easily coded in C or C++.

The fitness function will measure the predicted quality of any given oligo. In this application, the function will rate the oligo on a number of criteria (such as match to the target DNA, closeness of match region to some desired spot such as a translational-initiation site, GC-richness, lack of base-pair repeats longer than doublets, etc.). The oligo gets a score S on each of these characteristics, and the fitness function returns a number $0.0 < F < 1.0$, with $F = S_1 \cdot w_1 + S_2 \cdot w_2$ ... where

[8] Though this is of course a "toy" problem, and is presented mainly for illustration.

the constants w_1, w_2, etc. represent the weights given to each criterion (that is, match to target DNA sequence is probably more important than most of the other characteristics, so that its value will be weighted more heavily in the definition of the fitness function than the other criteria). Since the fitness function will involve simple string matching and counting, and since the strings are short, this algorithm can probably be efficiently run on a regular workstation, and supercomputer resources will not be necessary.

The algorithm would be coded exactly as shown in figure 1, and linked with a simple graphical or data-plotting library; calls to this library will be used to display (as a function of generation number) the top fitness and average fitness of the population, resulting in a plot like that of figure 2. The initialization step of the program will involve reading the target DNA sequence from a file on disk.

If one already has an oligo in mind, that oligo can be hand-inserted into the initial population, in order to save time, and to optimize it further. This should only be done, however, along with another purely random run, because having a good oligo present from the beginning may cause it to dominate the evolution, thus pushing out other solutions which may in time evolve to be even better.

Conclusion

Genetic Algorithms have been shown to be effective on a variety of problems resistant to analytical solutions and other search methods. Their straightforward design, which is based on the evolutionary paradigm so familiar to biologists, and their domain-independent nature, makes them a good choice for many types of problems of interest to the biomedical community. A vigorous field of research is currently growing around GAs, and it is expected that their utility to real-world problems will increase greatly in the near future. However, I think the primary interest of GAs, and other members of the field of Artificial Life, lies in their implications for the study of biological evolution and development. I encourage all in-

terested readers to scan the proceedings of the four Artificial Life conferences, and some issues of the journal *Evolutionary Computation*. Your time will be amply rewarded by new ideas and approaches to problem solving based on the biological realm.

Appendix

Several important resources are available free to those who are implementing GAs. These resources are internet-based[9], and consist of repositories of ready-made GA software and places where GA experts are available for advice.

1) Bibliographies: large bibliographies of GA and related literature can be obtained by anonymous ftp at:

- host garbo.uwasa.fi, in directory /pc/research/2500GArefs.ps.gz
- host magenta.me.fau.edu, in directory /pub/ep-list/bib-EC-ref.ps.Z

to obtain these lists, one needs to FTP to the host mentioned (using "anonymous" as the username, and an email address as the password), cd to the appropriate directory, get the file (in binary mode), and uncompress it before printing. Both are in PostScript format.

2) Digests: mailing lists are maintained by people knowledgeable in GAs. By sending email to the following addresses, including your question and a reply address, it is possible to obtain answers to almost any GA-related problem.

- ga-list@aic.nrl.navy.mil
- alife@cognet.ucla.edu
- ep-list@magenta.me.fau.edu
- genetic-programming@cs.stanford.edu
- ga-molecule@tammy.harvard.edu

There is also the USENET group "comp.ai.genetic" which may be accessed by various newsreader programs (such as "rn"). Posting to this newsgroup is another way to get help with GAs.

9 And thus require access to the internet; in general, a book on basic internet use is also helpful.

3) Packages: a wide variety of packages (ready-made GA software which can be supplemented with problem-specific routines) are available at no cost on various FTP sites. A complete list (along with much other useful information) can be obtained as the files in /pub/usenet/comp.ai.genetic/, on the host rtfm.mit.edu. A small subset of these is:

- ESCaPaDe, for Unix systems, from hoffmeister@ls11.informatik.uni-dortmund.de
- Genie, for MAC systems, from p_stampoul@fennel.cc.uwa.oz.au
- LibGA, for Unix, DOS, NeXT, and Amiga systems, from corcoran@penguin.mcs.utulsa.edu

Of course, a variety of commercial packages are available as well. In addition, a good and freely-available graphical package, "pgplot", can be obtained by anonymous FTP from deimos.caltech.edu. This package runs on a wide variety of computers and allows easy plotting of population statistics on many kinds of output devices.

4) Supercomputing resources: information about access to supercomputers (on a competitive grant basis) can be obtained by sending email to remarks.psc.edu.

References

[1] Holland, J. H., Adaptation in Natural and Artificial Systems, Ann Arbor MI: Univ of Michigan Press, 1975

[2] Kauffman, Stuart A., The Origins of Order, New York: Oxford Univ. Press, 1993

[3] Goldberg, D. E., Genetic Algorithms in Search, Optimization, and Machine Learning, MA: Addison-Wesley, 1989

[4] Davis, L., ed., Handbook of Genetic Algorithms, NY: Van Nostrand Reinhold, 1991

[5] Koza, J. R., Genetic Programming, MA: MIT Press, 1992

[6] Michalewicz, Z., Genetic Algorithms + Data Structures = Evolution Programs, NY: Springer-Verlag, 1992

[7] Fogel, L. J., A. J. Owens, and M. J. Walsh, Artificial Intelligence through Simulated Evolution, NY: Wiley, 1966

[8] Mitchell, M., and S. Forrest, (1993), Genetic Algorithms and Artificial Life, *Artificial Life*, **1**(1): 267-289

[9] Erickson, Robert P., (1993), Focus on antisense approaches, *Developmental Genetics*, **14**(4): 251-257

[10] Crooke, Stanley T. ed., Antisense Research and Applications, Boca Raton: CRC, 1993

Figure Legends

1) "Flowchart of a bare-bones genetic algorithm"
2) "A sample plot of fitness vs. generation in a GA"

Figure 1:

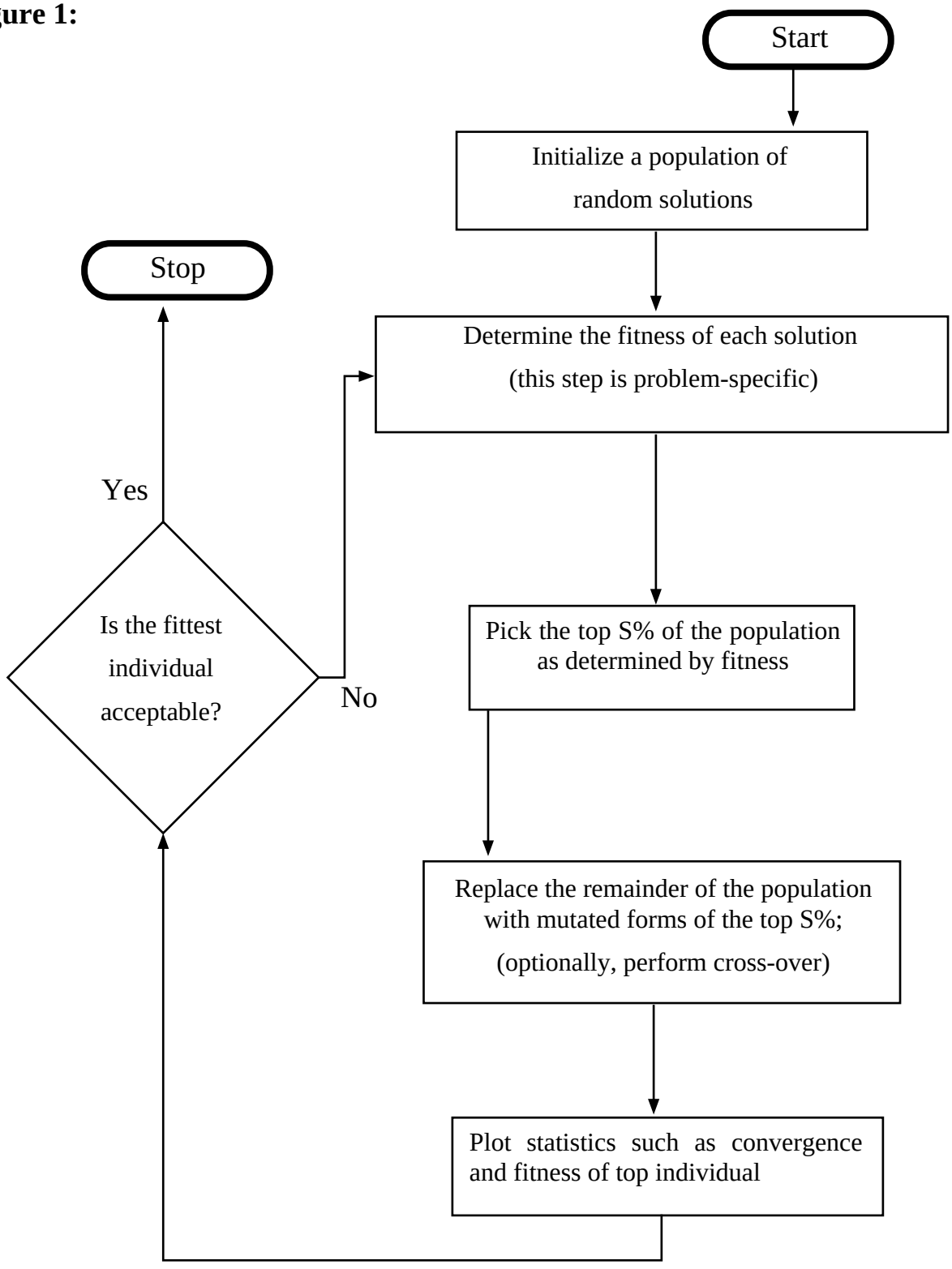
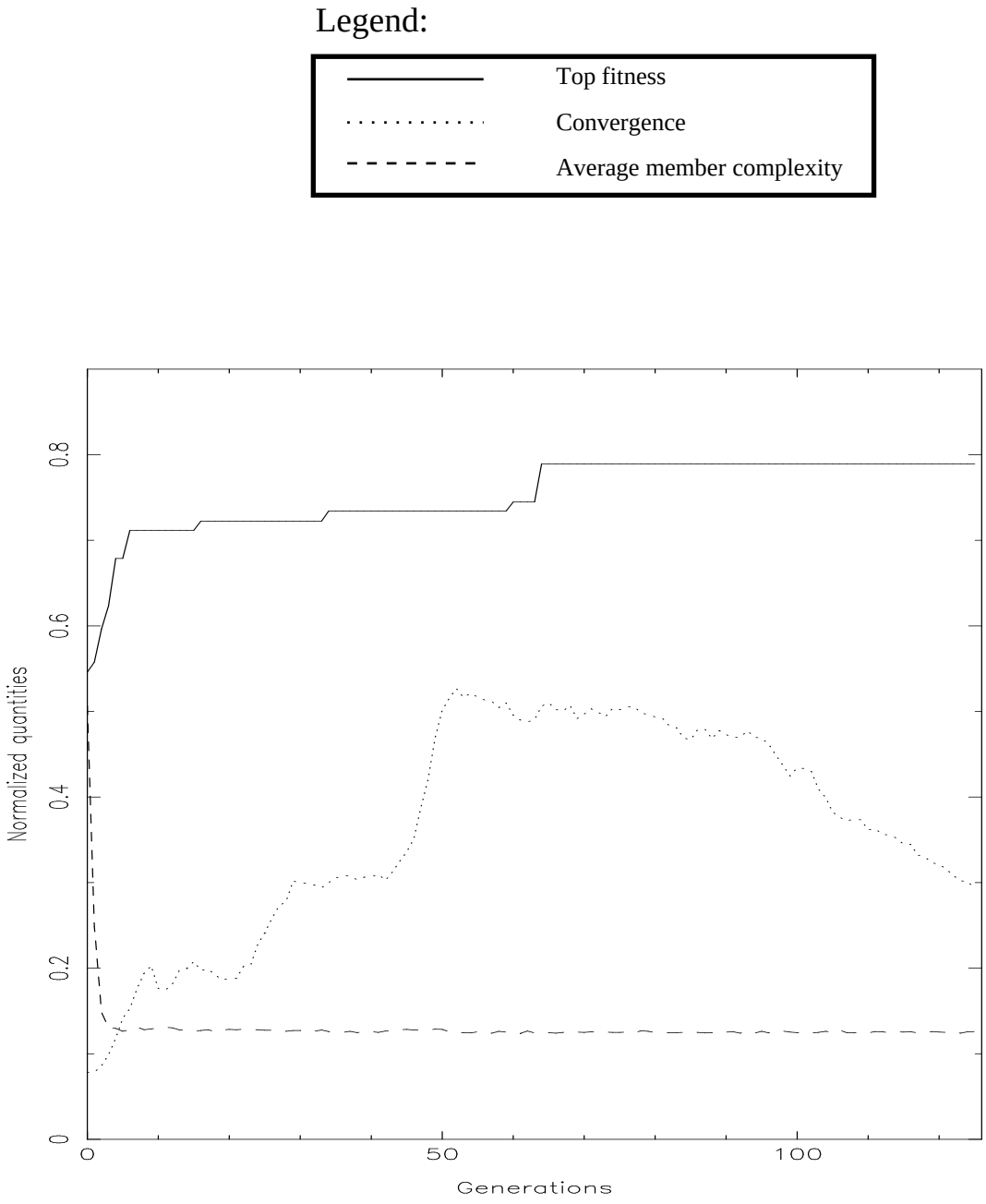

Start
Initialize a population of random solutions
Determine the fitness of each solution (this step is problem-specific)
Pick the top S% of the population as determined by fitness
Replace the remainder of the population with mutated forms of the top S%; (optionally, perform cross-over)
Plot statistics such as convergence and fitness of top individual
Is the fittest individual acceptable?
No
Yes
Stop

Figure 2:


Legend:
Top fitness
Convergence
Average member complexity
Normalized quantities
0
0.2
0.4
0.6
0.8
0
50
100
Generations