Automatic molecular design using evolutionary techniques

Al Globus, MRJ Technology Solutions, Inc. at NASA Ames Research Center
John Lawton, University of California at Santa Cruz
Todd Wipke, University of California at Santa Cruz

Abstract

Molecular nanotechnology is the precise, three-dimensional control of materials and devices at the atomic scale. An important part of nanotechnology is the design of molecules for specific purposes. This paper describes early results using genetic software techniques to automatically design molecules under the control of a fitness function. The fitness function must be capable of determining which of two arbitrary molecules is better for a specific task. The software begins by generating a population of random molecules. The population is then evolved towards greater fitness by randomly combining parts of the better individuals to create new molecules. These new molecules then replace some of the worst molecules in the population. The unique aspect of our approach is that we apply genetic crossover to molecules represented by graphs, i.e., sets of atoms and the bonds that connect them. We present evidence suggesting that crossover alone, operating on graphs, can evolve any possible molecule given an appropriate fitness function and a population containing both rings and chains. Prior work evolved strings or trees that were subsequently processed to generate molecular graphs. In principle, genetic graph software should be able to evolve other graph representable systems such as circuits, transportation networks, metabolic pathways, computer networks, etc.

Introduction

Hypothesis

Crossover alone, operating on graphs, can evolve any possible molecule given an appropriate fitness function and a population containing both rings and chains. This hypothesis is supported but not proven.

Design problem

Many problems associated with the development of nanotechnology require custom designed molecules. Frequently it is possible to precisely define what a molecule must do and still have significant problems designing a molecule to do the task. Therefore, a design technique that can automatically generate candidate molecules given requirements may be useful. Genetic algorithms [Holland 75], genetic programming [Koza 92] and genetic graphs can automatically generate solutions to problems given a function that determines which of two candidate solutions is better. Genetic algorithms evolve strings. Genetic programming evolves tree-structured programs. Genetic graphs, described here for the first time (to our knowledge), evolves graphs. There are several classes of molecular nanotechnology designs that can be described as graphs; i.e., a set of vertices and a set of edges each of which connects two vertices. Molecules can be described as a set of atoms (vertices) connected by a set of bonds (edges). Analog circuits can be described as a set of vertices (nodes) connected by a set of wires or components (edges). Digital circuits and, presumably, future nanoelectronic circuits can be similarly described. An automated system for designing graphs with desirable properties should therefore be able, at least in principle, to design a variety of molecular nanotechnology systems. In this paper we focus on the design of small molecules; in particular, pharmaceutical drugs.

Although drug design is not usually considered molecular nanotechnology, this is a misconception that presumably started because the earliest nanotechnology work focused on systems analogous to macroscopic machines. Drugs are frequently small molecules that precisely fit into receptor sites to block molecular processes in the body. This must be accomplished without fitting the receptor sites of the body?s healthy molecular machinery. Furthermore, drug molecules must survive in the body long enough to be effective. Early drug discovery was accomplished without understanding these mechanisms, but modern drug design consciously creates molecules with atomic precision to bind well to receptor sites in disease organism proteins. This is precise, three-dimensional control of biological devices; i.e., molecular nanotechnology.

One approach to drug design is to find molecules similar to good drugs that have negative side effects. Ideally, a candidate replacement drug is sufficiently similar to have the same beneficial effect but is different enough to avoid the side effects. In any case, to use genetic graphs for similarity-based drug discovery we need a good similarity measure that can score any molecule. [Carhart 85] defined such a similarity measure, all-atom-pairs-shortest-path, and searched a large molecular database for molecules similar to diazepam. We use a closely related similarity measure.

Genetic software techniques

For an excellent review of genetic software techniques as of spring 1997 see [Baeck 97]. Genetic software techniques seek to mimic natural evolution's ability to produce highly functional objects. Biology produces organisms. Genetic software produces sets of parameters, programs, molecular designs, and many other structures. Genetic software solves problems by:

1. Randomly generating a population of individual potential solutions.
2. For each new generation, repeatedly selecting parent individuals at random with a bias towards better individuals and applying one of the following transmission operators:
    1. Crossover: each of two parents is divided into two parts and one part from each parent is combined into a child.
    2. Mutation: a single ?parent? is randomly modified to create a child.
    3. Reproduction: a single ?parent? is copied into the new generation.
1. Continuing until an acceptable solution is found or for a certain number of generations.

Genetic software techniques differ in the representation of solutions. Genetic algorithms use strings of symbols for the representation. The crossover operator breaks strings in half, usually at a random point. Bit strings are a common representation, but arrays of floating point numbers, special symbols that generate circuits [Lohn 98], robot commands [Xiao 97], and many other symbols may be found in the literature. Strings may be of fixed or variable length.

Genetic programming [Koza 92] uses trees to represent individuals. This is particularly useful for representing computer programs. For example, a tree node representing assignment has two child-nodes, one representing a variable and the other representing a value. The crossover operator exchanges randomly selected sub-trees between two parent-trees. Trees may be viewed as graphs without cycles.

Many molecules contain cycles, which chemists call rings. Therefore, any attempt to use genetic programming to design molecules must have a mechanism to evolve cycles. This is non-trivial when crossover can replace any sub-tree with some other random sub-tree. After much thought we were unable to devise a crossover-friendly tree representation of arbitrary cyclic graphs. Crossover-friendly means that any sub-tree is a potential crossover point without restriction.

[Nachbar 98] used genetic programming to evolve molecules for drug design by sidestepping the crossover/cycles problem. Each tree node represents an atom with a bond to the parent-node atom and each child-node atom. Hydrogen atoms are explicitly represented and are always leaf nodes. Rings are represented by numbering certain atoms and allowing a reference to that number to be a leaf node. **Crossover is constrained not to break or form rings**. Ring evolution is theoretically permitted by specific ring opening and closing mutation operators, but the only example problem solved in [Nachbar 98] did not involve any molecules with rings.

Method

Genetic Graphs

Genetic graphs uses cyclic graphs to represent molecules. Vertices are typed by atomic elements. Edges can be single, double, or triple bonds. Valence is enforced. Heavy atoms are explicitly represented by vertices but hydrogen atoms are implicit; i.e., any heavy atom with unfilled valence is assumed to be bonded to hydrogen atoms but these are not represented in the data structure. Our genetic graph software evolves the population using crossover only; i.e.., mutation and reproduction are not implemented.

The initial population is generated by choosing a random number of atoms between half and twice the size of the target molecule. Atomic elements are randomly chosen from the elements present in the target. Bonds are then added at random to construct a spanning tree; i.e., at this point all atoms are connected into a single molecule. Then a random number of additional bonds are added to create cycles. This number is chosen to be between half and twice the number of cycles in the target molecule. The number of cycles is always bonds - atoms + 1.

For this work, tournament selection was used to choose parents in a steady state genetic system. Tournament selection means that parents are chosen by comparing two randomly chosen individuals and taking the best. Steady state means that new individuals (children) replace poor individuals in the population rather than creating a new generation. The poor individuals are also chosen by tournament, but the worst individual is selected for replacement. By convention, after population-size individuals have been replaced, we say that one generation is complete. The implementation follows this procedure:

1. Generate a random population of molecules
2. Repeat many times, gathering data periodically:

    1. Select two molecules from the population at random. Call the better molecule father.
    2. Select two molecules from the population at random. Call the better molecule mother.
    3. Make a copy of father and rip it into two parts at random.
    4. Make a copy of mother and rip it into two parts at random.
    5. Combine one part of the copy-of-father and one part of the copy-of-mother into a molecule called son.

6. Combine the other part of the copy-of-father and the other part of the copy-of-mother into a molecule called daughter.
7. Choose two molecules from the population at random. Replace the worst one with son.
8. Choose two molecules from the population at random. Replace the worst one with daughter.

The most difficult portion of implementing genetic graphs is the crossover operator described above as "ripping" molecules into two parts and combining parts from each parent-molecule. Crossover is accomplished by the following procedure:

1. Create copies of each parent.
2. Randomly cut each copy into two parts by selecting a random edge-cut-set and removing the edges in the cut set from each copy. An edge-cut-set is a set of edges that, when removed, causes a graph to break into two disconnected subgraphs. The cut set is generated by the following procedure:
   1. Choose an edge at random.
   2. Find the shortest trail between the vertices of the edge. A trail is an ordered list of edges, each of which shares a vertex with each neighboring edge. The first and last edges connect the two vertices of the original randomly chosen edge.
   3. Select a random edge from the trail, remove it from the molecule, and place it in the cut set.
   4. Go to 2 until a cut set is found.
3. Combine one part of the father's copy with one part of the mother's copy at random by the following procedure:
   1. Select a random cut edge (an edge in either cut set). Call this edge?s vertex in the part to be mated v1.
   2. If any compatible (same bond type) cut edge in the other part-parent-copy exists, choose one at random. Call this edge?s vertex in the other part-parent-copy v2. Connect v1 and v2 with a compatible edge.
   3. If no comparable cut edge was found, select a random cut edge in the other part-parent-copy and connect the appropriate vertices with a random edge that satisfies valence.
   4. If no cut edge is left in the other part-parent-copy, flip a virtual coin. If heads, connect v1 to a random vertex in the other part-parent-copy.
   5. Go to 1 until all cut edges have been processed exactly once.

This approach can open and close rings using crossover alone and can even generate cages and higher dimensional graph structures as long as there are rings in the population. Unfortunately, if there are no rings in the population none can be generated. Also, once the population consists entirely of two-vertex-graphs, no graphs with more than two vertices can be generated. Nonetheless, this approach is by far the most general of those we examined or found in the literature. In particular, unlike [Nachbar 98] no special-purpose ring opening and closing operators are necessary.

The resources required for genetic software to find a solution is a function of the size of the search space, among other factors. The space of all possible graphs is combinatorial and enormous. For molecular design this space can be radically reduced by enforcing valence limits for each atom. Thus, a carbon atom with one double and two single bonds will not be allowed to add another bond. Also, avoiding explicit representation of hydrogen atoms substantially reduces the size of the graph and therefore the search space.

Fitness function: all-pairs-shortest-path similarity

The key to any genetic software solution is a good fitness function -- in this case a function that can determine if one molecule is better than another. This function must be very robust since the randomly generated initial molecules rarely make much chemical sense. Also, for our initial studies we wanted a fitness function that only required the graph of a molecule, not the xyz coordinates of each atom. This simplifies initial studies and avoids the necessity of minimizing candidate molecules, a CPU intensive step. The all-atoms-pairs-shortest-path similarity test chosen [Carhart 85] is a robust graph-only fitness function. Each atom is given an extended type consisting of the atomic number and the number of single, double, and triple bonds the atom participates in. Then the shortest trail between each pair of atoms is found. A set is constructed with one element for each atom pair at each path length plus duplicates when there is more than one shortest path (this happens going around rings). The fitness of each candidate molecule is the distance between its set and the set of a target molecule.

The distance measure used is the Tanimoto Coefficient. This is the size of a-intersection-b divided by the size of a-union-b; where a is the candidate?s set and b is the target?s set. Two elements are considered identical for the purpose of the intersection and union operations if the atoms have the same extended type and the distance between them is identical. This measure always returns a number between 0 and 1. Historically, we have preferred fitness functions that return lower numbers for fitter individuals, so we subtract the Tanimoto Coefficient from one.

The targets for our initial study were butane, benzene, cubane, purine, diazepam, morphine and cholesterol. The fitness function can not only find similar molecules, which is useful in drug design, but also can lead evolution to the exact molecule used as a target. This proves that the algorithm can reach particular kinds of molecules and the number of generations to find the target provides a quantitative measure of performance. Unfortunately, all-pairs-shortest-path is an $O(n^3)$ algorithm.

Implementation

Genetic graphs is implemented in Java. Java was chosen since it is similar to C++ (a language known to the authors), is becoming quite popular so many useful libraries are available, its garbage collection vastly simplifies memory management, and Java?s bug-reducing features substantially reduce debugging time and produce more robust code. A significant run-time penalty is paid for these advantages. With luck, future improvements in Java development and run-time environments will reduce the performance penalty. All production runs were executed on SGI workstations at the NAS Division of NASA Ames Research Center.

Test environment

By hypothesis, our genetic graphs algorithm can find any possible molecule. To partially test this hypothesis, we ran the program using several target molecules:

1. Butane -- for a simple linear molecule.
2. Benzene -- for a simple molecule with a ring.
3. Cubane -- for a cage molecule.
4. Purine -- for fused rings and heteroatoms.
5. Diazepam -- since this was used in [Carhart 85].
6. Morphine -- for a complex drug molecule and because Dr. Wipke's group has worked on morphine analog design for many years.
7. Cholesterol -- for a non-drug molecule.

Since the algorithm is stochastic, twenty runs were conducted for each target molecule. The number of generations and population size was varied in an attempt to have enough successful runs (at least 11) to calculate the median time to find the target. Once the target is found the run stops. Runs also stop after a fixed, maximum number of generations. A few of the best individuals were saved to see if the software produced molecules similar to the target. These may be useful for drug design.

Results

20 runs for each molecule	Population size	Median generations to find target	Minimum generations to find target	Number of runs that failed to find target	Maximum generations
Benzene (H_6C_6)	200	39.5	2	8	1000
Cubane (H_8C_8)	100	46.5	13	0	1000
Purine ($H_4C_5N_4$)	100	245	19	4	1000

Median, rather than mean, generations to find the target was chosen because the data varied widely and many runs did not complete. Butane was usually found in the initial random population so data were not taken. At the time of writing, there was not sufficient data for diazepam, morphine, or cholesterol to calculate the median for twenty runs. With a population of 100, the benzene runs did not find the target often enough to calculate the median so the larger population size was used. Diazepam has been found once in generation 256 with a population of 200 out of nine runs that terminate in at most 1000 generations. Also, during testing leading up to the production runs morphine was found in one run at generation 208 with a population of 1000.

A number of other measures were taken. These will be presented in a future paper when data for all the target molecules is available.

Discussion

The genetic graphs algorithm can clearly find small molecules given an appropriate fitness function, and can find more complex molecules although significant time is required. The variability of runs is remarkable. Note that eight runs failed to find benzene in 1000 generations, but one run found benzene in only two generations. It?s also interesting to note that cubane was always found although the median time to discovery was somewhat greater than for benzene; and the cubane run used a smaller population. Finally, note the much larger median time to success for purine. Apparently the addition of nitrogen and the fused ring made finding the target significantly more difficult.

Performance analysis demonstrates what might be expected - the $O(n^3)$ fitness function takes most of the CPU time. A faster fitness function would substantially speed calculations. Genetic software lore suggests that the fitness function is exceptionally important [Kinnear 94]. Our results bear this out.

Fortunately, the algorithm is embarrassingly parallel since many runs are required. Also, there is significant potential for parallelism within runs since fitness function execution on each individual is completely independent. Furthermore, the algorithm can be easily restarted and can afford to lose some runs. Thus, genetic graphs is a good candidate for cycle-scavenging batch systems such as Condor [Litzkow 88]. Large genetic graph production runs can therefor use wasted workstation and PC cycles at facilities with large numbers of these machines.

Future work

The performance of the software clearly needs improvement. Not surprisingly, most of the time appears to be spent in the $O(n^3)$ fitness function. We have identified coding changes that should improve performance substantially and modifications to the fitness function that should provide much larger benefits by degrading the quality of the similarity test for most fitness function evaluations.

Although finding target molecules is a useful measure of the algorithm, we already know the target molecule. The real purpose of the similarity fitness function is to find molecules similar but not identical to the target. In particular, the ideal result is a wide variety of molecules dissimilar to each other but relatively similar to the target molecule. This provides a diverse set of candidate molecules for drug development, a process that takes millions of dollars and years to complete. In the ideal case, one or more candidates will have the beneficial properties of the target without negative side effects. Preliminary analysis of collections of the best individuals from each generation suggests that these collections are quite diverse and share some of the properties of the target molecule. These data are not yet ready for publication.

Many fitness functions of interest require molecular conformation; i.e., xyz coordinates for each atom. For example, designing a molecule to fit in a protein receptor to inhibit the activity of a disease organism. The new and fairly effective AIDS drugs are an example of this approach. To design a fitness function evolving molecular fit, the very bizarre molecules often created by crossover must be minimized quickly. Most minimizers available today will not work with without a "reasonable" start point. We are searching the literature for algorithms to minimize very "bad" molecules.

Circuit design is another field for which genetic graphs should, in principle, be well suited. Genetic algorithms (using variable length strings) [Lohn 98] and genetic programming [Koza 97] have been used to design analog circuits. In the genetic programming case, a tree language to generate analog circuits compatible with the SPICE (Simulation Program with Integrated Circuit Emphasis) [Quarles 94] simulator was constructed and a 64 node (80MHz per node) parallel supercomputer was used to design the circuits. The system designed a lowpass filter, a crossover filter, a four-way source identification circuit, a cube root circuit, a time-optimal controller circuit, a 100 dB amplifier, a temperature-sensing circuit, and a voltage reference source circuit. Thus, genetic programming can design graph-structured systems. However, we have found it extremely difficult to create a tree language that can generate any possible graph and support crossover cleanly. Therefore, it may be advantageous to directly evolve graphs rather than evolved trees that generate graphs.

Summary

Algorithms and software to evolve graphs using genetic techniques were developed and applied to drug design using a molecular similarity based fitness function. Early data suggest that the software can indeed discover a variety of small molecules. Significant additional work will be required to demonstrate that representing molecules as graphs and using genetic software techniques is of major benefit in molecular design.

Acknowledgments

Thanks to Rich McClellan for providing the mol file reading and atomic element code. Thanks to Creon Levit, Subash Saini, and Meyya Meyyapan of NASA Ames for their support. Thanks to Creon Levit and Bonnie Klein for reviewing this paper. This work was funded by NASA Ames contract NAS 2-14303.

References

[Baeck 97] Thomas Baeck, Ulrich Hammel, and Hans-Paul Schwefel, "Evolutionary computation: comments on the history and current state," *IEEE Transactions on Evolutionary Computation*, volume 1, number 1, pages 3-17, April 1997.

[Carhart 85] Raymond Carhart, Dennis H. Smith, and R. Venkataraghavan, "Atom pairs as molecular features in structure-activity studies: definition and application," *Journal of Chemical Information and Computer Science*, 23, pages 64-73, 1985.

[Holland 75] John H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press, 1975.

[Kinnear 94] Kenneth E. Kinnear, Jr., "A perspective on the work in this book," *Advances in Genetic Programming*, edited by Kenneth E. Kinnear, Jr., MIT Press, Cambridge, Massachusetts, pages 3-20, 1994.

[Litzkow 88] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations,'' *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June1988. See http://www.cs.wisc.edu/condor/.

[Lohn 98] Jason D. Lohn and Silvano P. Colombano, " Automated analog circuit synthesis using a linear representation,'' *Second International Conference on Evolvable Systems: From Biology to Hardware*, Springer-Verlag, Sept.23-25, 1998. (to appear)

[Quarles 94] T. Quarles, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE 3 Version 3F5 User's Manual*, Department of Electrical Engineering and Computer Science, University of California at Berkeley, CA, March 1994.

[Koza 92] John R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Massachusetts, 1992.

[Koza 97] John R. Koza, Forrest H. Bennett III, David Andre, Martin A. Keane and Frank Dunlap, "Automated synthesis of analog electrical circuits by means of genetic programming," *IEEE Transactions on Evolutionary Computation*, volume 1, number 2, pages 109-128, July 1997.

[Xiao 97] Jiang Xiao, Zbigniew Michalewicz, Lixin Zhang, and Krzysztof Trojanowski, "Adaptive evolutionary planner/navigator or mobile robots," *IEEE Transactions on Evolutionary Computation*,

volume 1, number 1, pages 18-28, April 1997.