Greedy Closure Evolutionary Algorithms

Dan Ashlock  
Mathematics Department  
Iowa State University, Iowa 50011  
danwell@iastate.edu

Ling Guo  
Bioinformatics Program  
Iowa State University, Iowa 50011  
guol@iastate.edu

Fang Qiu  
Agronomy Department  
Iowa State University, Iowa 50011  
fqiu@iastate.edu

Abstract

We present a method of using a genetic algorithm to evolve controls for a greedy algorithm. We demonstrate the technique on the location of embeddable DNA markers for genetic libraries. The technique has the potential for broad application and other applications are discussed.

1 Introduction

Greedy algorithms are familiar to most software developers. A few, like the algorithm for finding a minimal-weight spanning tree, can be proven to yield optimal results. Problems such as graph coloring or the traveling salesman problem admit a plethora of greedy algorithms which yield sub-optimal results. The control of greedy algorithms seems a natural target for evolutionary computation but relatively few papers have appeared. There are several possible approaches. The approach explored here consists of making small modifications in the order of presentation of potential parts of a growing structure as a means of deflecting the greedy algorithm’s behavior. The role of evolutionary computation is to locate the modifications of the order of presentation. The resulting chromosome structure is not a standard one for evolutionary computation and the binary variation operator (crossover operator) used shows empirical evidence of macromutational behavior[2].

1.1 Other Approaches

An approach to evolutionary control of greedy algorithms appears in [4]. A classical binary-gene genetic algorithm was used to optimize the behavior of a greedy algorithm for finding vertex covers of graphs. The basic greedy algorithm used was to place the neighbor or neighbors of the lowest degree vertex remaining in the graph into the vertex cover and then update the graph by deleting the vertex in question along with its neighbors. A binary gene, termed a binary decision diagram, within the genetic algorithm was used as a controller for the greedy algorithm. Each bit of the gene controlled one choice made by the greedy algorithm. When the relevant loci was 0 the greedy algorithm behaved in its usual fashion. If the loci was 1 then the algorithm instead placed the low degree vertex into the cover and update the graph by deleting only that vertex. In abstract, the idea is to permit a gene to function as a string of Boolean variables that override the behavior of the greedy algorithm. This idea has a very large number of possible applications. A potential problem in application of this technique is that if the number of objects to be examined by a greedy algorithm is large then a very large binary gene is required. The hybrid greedy-evolutionary algorithm described here avoids this potential problem.

A technique that may be able to incorporate either the binary decision diagram approach or the approach outlined here is described in [6]. In this technique an evolutionary algorithm is used to decide which of a group of heuristics are used and in what order to solve a problem. The authors experiment with direct solution of a complex scheduling problem with an evolutionary algorithm. When the direct approach failed, they instead gathered a group of heuristics and subdivided the problem, using an evolutionary algorithm to perform the assignment of heuristics.

In [3] the authors use greedy algorithms as variation operators, and in fact local search operators that yield faster hill climbing within the fitness landscape are a logical place to incorporate greedy algorithms into evolutionary computation.

1.2 Conway’s Lexicode Algorithm

We will focus on Conway’s lexicode algorithm, Algorithm 1, for the example of our technique presented here. An extensive work on coding theory is [9] and we assume at least some familiarity with the theory of error correcting codes, though we will be working with edit metric based rather than Hamming metric
based codes. An \((n, d)\)–code is a collection of words of length \(n\) over some alphabet that have minimum pairwise distance \(d\) for some metric. Our alphabet is the \(\{C, G, A, T\}\) of DNA and our metric is the edit distance \([8, 5]\). The edit distance between two strings is the least number of single-character insertions, deletions, and substitutions that will transform one string into another.

**Algorithm 1**

**Input:** A minimum distance \(d\) and a word length \(n\).  
**Output:** An \((n, d)\)–code.

**Algorithm:**

Place the words of length \(n\) in lexicographical order. Initialize an empty set \(C\) of words. Scanning the ordered collection of words, select a word and place it in \(C\) if it is at edit distance \(d\) or more from each word placed in \(C\) so far.

The motive for focusing our efforts at evolutionary control of greedy algorithms on this problem was a need, within another project, for an error correcting code for use in DNA constructs. In the process of building genetic libraries tissues from many parts of the target organism are pooled. Short stretches of DNA, called DNA barcodes, that identify the source tissue can be incorporated into the libraries and used to retrieve the identity of the source tissue in subsequent experiments. Since DNA sequencers sometime miscall, ignore, or duplicate a base the edit metric is the appropriate metric for error correction in DNA barcodes. Computation of the edit distance between two strings is a quadratic time algorithm, taking time proportional to the product of the lengths of the strings involved \([5]\).

2 Greedy Closure Evolutionary Algorithms

Examining Algorithm 1, the reader will see that the algorithm extends a partial code as it goes along. Since the algorithm considers the potential code words in a fixed order and starts with an empty set of code words the algorithm is deterministic. Two potential evolutionary approaches would be to evolve the order in which the words are considered or to have a binary choice to accept or reject a given word that would normally have been acceptable. This latter approach is exactly a binary decision diagram (BDD) as used in \([4]\). If we use such a BDD it is intuitive that the fitness of bits used later with a given chromosome will be very strongly dependent on the early bits. This is because those early bits mark large metric spheres of the space as used and leave other parts of it available. The hardness, as an evolutionary search problem, is also high because the problem will decompose poorly. Since choices made by the greedy algorithm early on during the processing of a given chromosome deflect the algorithm into entirely different parts of the search space, there is likely to be a high level of epistasis. The recognition of this potential for epistasis is what led us to develop our new approach.

A greedy closure evolutionary algorithm is, in a sense, a restricted version of permuting the order in which the words are considered. We will initially present greedy closure evolutionary algorithms as a modification of Conway’s lexicode algorithm. The chromosome is a short list of code words, termed a seed, that are placed in the code before applying the greedy algorithm. We then use Conway’s algorithm to finish (close) the partial code. The fitness of a particular seed is the size of the error correcting code that results when the seed is closed. Note that a greedy closure evolutionary algorithm could be used to modify any greedy algorithm that operated by extending partial structures. The chromosomes are seeds (partial structures) and the fitness is a measure of the quality of the completed structure found by applying the greedy algorithm to the seed. In forcing a few words into the code, we are slightly modifying the order in which the Lexicode algorithm considers the words. Because of the dynamics of the algorithm this small change can have a huge effect. In essence we are recognizing and exploiting potential epistasis by evolving only a few of the bits and then using the greedy algorithm to find the rest of the structure quickly, instead of using evolution to hack our way through an epistatic mire.

2.1 Implementation Details

An evolutionary algorithm comes with a laundry list of parameters and choices. These choices include representation (which choice is central here), selection technique, and choice of variation operators. The parameters include population size, termination conditions, and rates of application for variation operators. Our representation (chromosome) will consist of a seed code of three code words. Initial populations are created by generating seeds repeatedly, rejecting seeds that have members closer than the given minimum distance, until enough seeds are available to fill the population.

Our selection method will be steady state single tournament selection with tournament size four. In each tournament, four chromosomes are chosen uniformly at random. The two best undergo reproduction and the resulting new chromosomes displace the two worst. Reproduction consists of two steps. The first step copies the parents and then independently swaps the code word at each locus within the seeds 50% of the time. This is a form of uniform crossover
that treats the code words making up the seeds as atomic objects. The second step modifies each of the new structures produced by crossover by picking one position within the seed and placing a new code word there. This latter variation operator is a type of mutation. Both the variation operators listed here have the potential to create seeds which violate the minimum distance for the code. Such invalid seeds are awarded a fitness of zero and so eventually removed by selection. For valid seeds, the size of the code resulting from application of the lexicode algorithm to the seed is the fitness of that seed. Larger codes are better given fixed length and minimum distance parameters.

3 Results

<table>
<thead>
<tr>
<th>Code Sizes</th>
<th>Minimum Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
</tr>
<tr>
<td>7</td>
<td>311</td>
</tr>
<tr>
<td>8</td>
<td>1025</td>
</tr>
<tr>
<td>9</td>
<td>3451</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
</tr>
</tbody>
</table>

*Big

Table 1: Size of DNA lexicodes found with the plain lexicode algorithm.

As a baseline for the evolutionary algorithm, Table 1 shows the result of running the plain lexicode algorithm, i.e. the fitness of an empty seed. Since the underlying space grows as \(4^n\), the lexicodes of a given minimum distance can grow no faster. The code sizes in Table 1 grow at a lower empirical rate. As a second baseline, sampling of the fitnesses of random seeds for one set of parameter values was performed. The result for length 6, edit distance 3 DNA codes is shown in Figure 1. Random sampling provided a substantial improvement upon the lexicode algorithm on empty seeds. The code size, 96, located by the algorithm running with an empty seed was one above the mode value of 95 obtained by sampling. The best code size located by random sampling was 103, which appeared three times in 20,000 samples.

The evolutionary algorithm was run 100 times with 10,000 mating events in each run for each of ten different choices of word lengths and minimum distances. Results from these runs are given in Table 2. For the benchmark parameter set for comparison with random sampling, length 6 distance 3, the evolutionary algorithm produced codes better than the best produce by random sampling in the majority of cases. Each of the evolutionary algorithm run performed 20,050 fitness evaluation, 50 more than in the random sampling. The greedy fitness evolutionary algorithm outperformed the plain lexicode algorithm for all parameter sets we tested.

![Histogram showing distribution of length 6, distance 3 edit-metric codes on the DNA alphabet located in 20,000 random samples. The largest code located by sampling has 103 code words. The empty seed yields a 96 word code.](image1.png)

![Histogram showing distribution of length 6, distance 3 edit-metric codes on the DNA alphabet located in 20,000 random samples. The largest code located by sampling has 103 code words. The empty seed yields a 96 word code.](image2.png)

Table 2: Comparison of code sizes for the plain lexicode algorithm and the greedy fitness evolutionary algorithm. The figures in parenthesis are the fraction of times the best result was located in 100 trials.

<table>
<thead>
<tr>
<th>Length</th>
<th>Minimum Distance</th>
<th>Plain Lexicode</th>
<th>Evolutionary Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>12</td>
<td>16 (18/100)</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>36</td>
<td>41 (2/100)</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>8</td>
<td>11 (1/100)</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>96</td>
<td>106 (2/100)</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>20</td>
<td>25 (11/100)</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>9 (9/100)</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>311</td>
<td>329 (2/100)</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>57</td>
<td>63 (1/100)</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>14</td>
<td>18 (12/100)</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>4</td>
<td>7 (92/100)</td>
</tr>
</tbody>
</table>

3.1 Modifying the Algorithm for Applications

The application for the edit metric lexicodes is to provide embeddable markers for cDNA libraries. Because these DNA barcodes are to be embedded in constructs there are a number of constraints on the
sequence that may be used that are driven by biology beyond the need for error correction. In making the constructs various restriction enzymes are used which cut a DNA strand at a particular pattern of DNA bases. We must avoid creating additional instances of this pattern either within our markers or as a side effect of embedding our markers into the construct. Substrings of the form TT or AAA will interfere with use of the construct because of a long string of T's near the point were the marker is embedded. It turns out that modifying the evolutionary algorithm to deal with such constraints is not difficult. In the seed generator, mutation operators, and the greedy algorithm we simply call code that checks the acceptability of each string. For the application that motivated this research the restrictions were as follows:

- The code word must not end in T.
- The strings TT and AAA must not appear.
- When the code word replaces the NNNNNN in the nucleotide string: AACTGGAAGAATTCGCGGCCGCNNNNNTTTTTTTTTTTTTTT no new instances of the following restriction enzyme sites must be created: GCGGCCGC, GAATTC, or CAGCTG.

Running the unmodified greedy algorithm on six character words with minimum edit distance 3 produced a 63 word code once these restrictions were imposed. The greedy fitness genetic algorithm improved substantially on this, finding several 74 word codes. The code sizes found in 100 runs with the constraints given are shown in Table 3. One of the four resulting largest codes is shown in Table 4.

<table>
<thead>
<tr>
<th>Words per Code</th>
<th>Instances in 100 trials</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>1</td>
</tr>
<tr>
<td>71</td>
<td>25</td>
</tr>
<tr>
<td>72</td>
<td>51</td>
</tr>
<tr>
<td>73</td>
<td>19</td>
</tr>
<tr>
<td>74</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3: Results of the biologically constrained greedy fitness evolutionary algorithm for length 6 distance 3 codes. The genetically controlled greedy algorithm finds from 7-11 additional code words as compared to the unmodified greedy algorithm.

4 Discussion and Conclusions

Edit metric codes of the DNA alphabet were proposed by Tom Cassavant of the department of Electrical Engineering at the University of Iowa. Dr. Cassavant’s suggested that these be used as embedded markers in cDNA libraries for assessment of tissue type in a pooled library. Edit metric codes are harder to locate by classical theorem-and-proof techniques because of the intrinsic ugliness of the spaces associated with the edit metric in comparison to those associated with the Hamming metric. The greedy fitness evolutionary algorithm bested the unmodified lexicode algorithm for all parameter values it was test on. When applied to the problem of locating a code with biological constraints, the evolutionary version of the algorithm produced improvements in code size of the same magnitude (a little better) than in the unconstrained case. The unconstrained search saw the plain lexicode’s 96 word code improved to 106 by use of the evolutionary algorithm. The constrained version had corresponding values 63 and 76. The suggests that the algorithm may well handle constraints of this type transparently.

The behavior of the evolutionary algorithm suggests that the crossover operator used is not mixing and matching building blocks, but rather acting as a macromutation in the sense described in [2]. Recall that to produce two “child” seeds from two parent seeds we perform uniform crossover on the list of code words forming the seeds. If two words in one of the new seeds are closer than the minimum distance permitted, then that seed is awarded a fitness of zero. Examine Figure 2 displaying the maximum and average population fitness for two runs of the evolutionary algorithm. It appears that succession events, in which one type replaces another within the population, are accompanied by large downward
spikes in the average fitness. The minimum fitness for a seed (established by sampling in one of the two baseline studies) that is not awarded a fitness of zero is above the bottom of some of the spikes shown in Figure 2. This suggests many invalid seeds are being created.

The apparent macromutational character of our binary variation operator is not intrinsically good or bad, but delimits the effective management strategies. One potentially valuable modification to the current evolutionary algorithm would be to reduce the rate of application of the binary variation operator below its current value of 100%. Another would be to use soft brood recombination as suggested in [1]. Since the brood size is $2^{\text{Seed Size}}$ (8 for the current algorithm) we could perform an exhaustive search on all eight possible results of the binary variation operator.

4.1 Issues not treated

We did not perform a study of the effect of varying the rate of application of the variation operators we are using. Given that we have some theoretical grounds for thinking our binary variation operator was applied at too high a rate this is a fertile area for potential incremental improvement in the future. We also neglected to look for other potential variation operators. A binary operator that sorted seeds into alphabetical order before swapping loci and one that randomizes the order of a seed before swapping loci both might yield differences in performance. An obvious unary variation operator that would avoid producing invalid seeds is to swap a seed in from the part of a code produced when closing a seed. This would allow us to perform some kind of traversal of the code space and might produce variation that continued the process of exploiting rather than suffering from the epistatic character of the search space. While some preliminary runs suggested that 3 was a good size for a seed in the length six distance three case, seed size is also a largely unexplored issue.

5 Other Potential Applications

In order to apply a greedy fitness evolutionary algorithm to a problem you must locate a greedy algorithm for the problem that admits modification of its behavior by seeds of the sort we used to search for error correcting codes. In this section we will give a few examples.

5.1 Graph Coloring

Readers unfamiliar with combinatorial graphs will find an excellent and complete introduction in [10]. The most common graph coloring problem is to find, for any given graph, a minimal proper coloring. Graph coloring is used to perform conflict resolution, from coloring a map so that adjacent countries are different colors, to finding times for committee meetings so that committees with overlapping membership meet at different times. The number of colors used represents a scarce resource (colors used in printing, times for committees to meet) and so the goal is to minimize the number of colors used.

Algorithm 2 Greedy Graph Coloring

Input: A graph with an ordered vertex set and an ordered set of colors.
Output: A proper coloring of the graph.
Algorithm:

Considering the vertices in order, assign to each vertex the smallest color not already assigned to one of its neighbors.

Algorithm 2 must constructively produce a proper coloring. It is also easy to recast as the fitness function for a greedy fitness evolutionary algorithm.
A seed is a set of vertices pulled out of order and colored first by the algorithm. Such seeds are the chromosome for the greedy fitness evolutionary algorithm. Fitness is the number of colors the algorithm used to color the graph. It is possible to prove that a seed exists that yields an optimal coloring (fewest possible colors), but the current proof does not place a upper bound on the size of the seed required. The proof also shows that the order the vertices in the seed are considered by the algorithm can be important and should be allowed to vary independently of the order of the vertices in the graph.

5.2 Vertex Covers

The application used to motivate binary decision diagrams in [4] is actually closely related to the coding theory application used in this paper. If the length of the shortest path between two vertices is taken as the distance between those two vertices then graphs take on the structure of a metric space. Dominating sets are a form of code relative to this metric, where the vertices in the vertex cover are analogous to code words. The emphasis is different, with the focus being covering all vertices rather than packing relative to a minimum distance. It would be straightforward to apply greedy fitness genetic algorithms to the vertex cover problem and also to the problem of finding codes in graphs.

5.3 Difference Sets

A difference set is a collection of integers with the property that all differences between pairs of members of the set are distinct. An example is \{1,2,4,8,13\} which has pairwise differences 1,2,3,4,5,6,7,9,11, and 12. A greedy algorithm to find difference sets would consider the positive integers in the usual order, adding the next integer to the set that did not cause a repeated difference. The seed would simply be a few initial numbers forced into the difference set, and fitness would be the number of members in the difference set when those members are drawn from a fixed range 1\ldots n.

6 Acknowledgments

This research was supported by a competitive grant from the National Science Foundation Plant Genome Program (award number: DBI-9975868) to Patrick S. Schnable, Daniel Ashlock and others and an award from the Iowa Corn Promotion Board to Patrick S. Schnable. This is Journal Paper No. J-19649 of the Iowa Agriculture and Home Economics Experiment Station, Ames, IA 50011, Project No. 3409, supported by Hatch Act and State of Iowa funds. We are grateful to Tom Cassavant of the Department of Electrical Engineering at the University of Iowa for pointing out the possibility of using DNA barcodes.

References

near Jr., editor, *Advances in Genetic Programming*, chapter 3. The MIT Press, Cambridge,

John R. Koza, Kalyanmoy Deb, MArco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and
Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Con-

problem. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference

ary Programming VII*, volume 1447 of *Lecture Notes in Computer Science*, pages 377–386,


[6] Emma Hart, Peter Ross, and Jeremy Nelson. Solving a real-world problem using and evolv-


07458, 1996.