1997

Genetic Algorithms for the Extended GCD Problem

Jonathan P. Sorenson
Butler University, jsorenso@butler.edu

Recommended Citation

V. Piehl, J. Sorenson, and N. Tiedeman, Genetic algorithms for the extended GCD problem, to appear in the Journal of Symbolic Computation. Posters presented at ISSAC'97 and the 1997 CUR Poster Session on Capitol Hill.

Article Submitted to Journal of Symbolic Computation

Genetic Algorithms for the Extended GCD Problem*

Valerie Piehl[1], Jonathan P. Sorenson[2] and Neil Tiedeman[3]

[1] *North Central High School, 1801 East 86th Street, Indianapolis, Indiana 46240, USA, vpiehl@msdwt.k12.in.us*

[2] *Department of Computer Science and Software Engineering, Butler University, 4600 Sunset Avenue, Indianapolis Indiana 46208, USA, sorenson@butler.edu, http://www.butler.edu/∼sorenson*

[3] *SAGIAN, 7451 Winton Dr., Indianapolis, Indiana 46268, USA, Neil.Tiedeman@sagian.com*

Abstract

The extended greatest common divisor (GCD) problem is, given a vector $\mathbf{a} = (a_1, \ldots, a_n)$ of positive integers, compute g, the greatest common divisor of these integers, and find a vector $\mathbf{x} = (x_1, \ldots, x_n)$ of integer coefficients such that

$$g = \sum_{i=1}^{n} a_i x_i.$$

It is desirable to find a solution vector $\mathbf{x}$ where $\|\mathbf{x}\|$ is small. We present several genetic algorithms for this problem. Our algorithms search among small multisubsets of $\{a_1, \ldots, a_n\}$; a solution for a particular multisubset is extended to a complete solution by padding $\mathbf{x}$ with zeros. We also present the results of our implementations of these methods.

*This work was carried out primarily in the Department of Mathematics and Computer Science at Butler University when the first and third authors were students, and completed by the second author while on sabbatical at Purdue University during the Fall of 1998.
Preliminary versions of this work were presented at the *Council for Undergraduate Research Poster Session on Capitol Hill*, April 1996, and at the *ISSAC'97* poster session.
This research was supported by NSF Grant CCR-9626877, the Holcomb Research Institute, and the Butler Summer Institute.

1. Introduction

We present several genetic algorithms for solving the extended greatest common divisor problem. After defining the problem and discussing previous work, we will state our results.

The extended greatest common divisor (GCD) problem is, given a vector $\mathbf{a} = (a_1, \ldots, a_n)$ of positive integers, compute g, the greatest common divisor of these integers, and find a vector $\mathbf{x} = (x_1, \ldots, x_n)$ of integer coefficients such that

$$g = \sum_{i=1}^{n} a_i x_i.$$

This problem arises in the computation of Smith and Hermite normal forms of integer matrices. For this application, to avoid excessive growth in intermediate results, we want the solution vector $\mathbf{x}$ to be "small." This requirement leads to what we will call the *decision problem* version of the extended GCD problem:

> Given a vector $\mathbf{a}$ of positive integers of length n and a bound k, does there exist an integer solution vector $\mathbf{x}$ such that
>
> $$g = \sum_{i=1}^{n} a_i x_i$$
>
> and $\|\mathbf{x}\| \leq k$?

Here $\|\cdot\|$ could be any norm or metric. Examples include the L_0 metric and the L_1, L_2, and L_∞ norms. (Recall that the L_0 metric is the number of nonzero entries, the L_1 norm is the sum of the absolute values of the entries, the L_2 norm is the Euclidean norm, and the L_∞ norm is the maximum entry in absolute value.) Other choices for $\|\cdot\|$ are possible, but we will limit our interest to these four.

Majewski and Havas [1994] showed that the extended GCD decision problem is $\mathcal{NP}$-complete when using either the L_0 metric or the L_∞ norm. Rössner and Seifert [1996] showed that no good approximation algorithms exist for either of these two choices. It seems likely that this problem is intractable for almost all norms or metrics of interest. Yet, finding *some* solution vector $\mathbf{x}$ is computable in polynomial time, and a number of algorithms exist to do this.

Among all known polynomial-time algorithms, the method of Havas et al. [1995, 1998], based on the LLL lattice-basis reduction algorithm, gives the best values for $\|\mathbf{x}\|$ in practice. However, a running time proportional to n^4 limits its practicality. The faster *sorting-GCD* method of Majewski and Havas [1995b] has a running time proportional to n^2, and gives solution vectors that are nearly as good as those the LLL method provides. Other algorithms include Blankinship's extension of Euclid's GCD algorithm [Blankinship, 1963], Bradley's improvement to Blankinship's algorithm [Bradley, 1970], and a tree-based method due to Majewski and Havas [1994]. (See also Lam et al. [2000].) Although these methods

are only linear in n, the quality of their solution vectors is decidedly inferior to that of the sorting-GCD and LLL methods. Nevertheless, we borrow ideas from all three of these algorithms.

Storjohann [1997] gave a polynomial-time solution to a modulo-N version of the extended GCD problem. (See also Mulders and Storjohann [1997].) We will not discuss this problem variation here.

In light of the intractability of the extended GCD decision problem, it makes sense to consider heuristic and randomized algorithms for this problem. In this paper we present several genetic algorithms, randomized heuristic algorithms based on the biological processes of evolution, for the extended GCD problem. The basic idea of our algorithms is to select a small multisubset of the entries of $\mathbf{a}$, compute a solution to the extended GCD problem on this set, and extend that solution to the entire input vector by padding with zeroes. To compute the extended GCD of the subset, we look at using an algorithm similar to Bradley's, a tree-GCD style algorithm somewhat similar to that of Havas and Majewski, and the sorting-GCD method. Each of these genetic algorithms, in turn, usually provide solution vectors that are smaller than the deterministic method upon which they are based, and have running times roughly linear in n. In fact, our approach can be applied to any algorithm for the extended GCD problem. We did not apply our genetic approach to the LLL-based algorithm; as mentioned earlier, it is rather slow, and we used input vectors with entries of up to 64 bits, where the LLL method would require multiprecision arithmetic to handle intermediate results. Our hardware supports 64-bit integer arithmetic.

In the next section we present a brief review of the Euclidean algorithm and its modification to compute the extended GCD. We give a brief overview of genetic algorithms in Section 3, and present our early attempts at genetic algorithms for this problem in Section 4. We present our subset-based genetic algorithms in Section 5. We conclude with a discussion of our experimental results in Section 6.

2. Background

As mentioned in the introduction, the decision problem for the extended GCD is $\mathcal{NP}$-complete for the L_0 metric and the L_∞ norm, but it is possible to find some solution vector $\mathbf{x}$ in polynomial time. In this section, we review Euclid's algorithm for computing GCDs and show how to extend it to compute the solution vector $\mathbf{x}$. We also discuss a *coefficient reduction* method due to Bradley.

Euclid's GCD Algorithm

The Euclidean algorithm is based on the following two simple facts:

$$\begin{aligned} \gcd(a, 0) &= a; \\ \gcd(a, b) &= \gcd(b, a \bmod b). \quad (b \neq 0) \end{aligned}$$

From this we obtain the following simple recursive algorithm:

Euclid($a, b;\ g$):

    if($b = 0$)
        $g := a$;
    else
        Euclid($b, a \bmod b;\ g$);

Next, we modify the algorithm to compute coefficients x, y such that $\gcd(a, b) = ax + by$:

Euclid($a, b;\ x, y, g$):

    if($b = 0$)
        $g := a$; $x := 1$; $y := 0$;
    else
        $q := \lfloor a/b \rfloor$;
        $r := a - qb$; /* *Here* $r = a \bmod b$. */
        Euclid($b, r;\ w, x, g$);
        $y := w - qx$;

To see that this works, in the recursive call we compute w, x so that $g = bw + rx$. Plugging in for r, we obtain $g = bw + (a - qb)x = ax + b(w - qx)$.

This algorithm requires $O(\log \max\{a, b\})$ arithmetic operations (see, for example, Bach and Shallit [1996]).

In practice, for efficiency reasons, we used an iterative version of this algorithm. Other, more efficient algorithms exist for computing GCDs [Jebelean, 1993, 1995, Schönhage, 1971, Sorenson, 1994, 1995, Weber, 1995], but we used integers of at most 64 bits, and for this range, Euclid's algorithm is quite sufficient.

Bradley's Algorithm

Next, to compute the extended GCD of a list of integers, we apply the following simple identity:

$$\gcd(a_1, \ldots, a_n) = \gcd(\gcd(a_1, a_2), a_3, \ldots, a_n).$$

With the proper care regarding the coefficients, this leads to the following algorithm, which is due to Bradley [1970]:

Bradley($\mathbf{a};\ \mathbf{x}, g$):

    Euclid($a_1, a_2,\ y_2, z_2, g$);
    for($i := 3$; $i <= n$; $i := i + 1$)
        Euclid($g, a_i,\ y_i, z_i, g$);
    $x_n := z_n$;
    for($i := n - 1$; $i >= 2$; $i := i - 1$)
        $x_i = z_i y_{i+1}$; $y_i = y_i y_{i+1}$;
    $x_1 := y_2$;

This algorithm takes $O(n \log \max\{a_i\})$ arithmetic operations in the worst case.

Coefficient Reduction

As an example, on the input vector $\mathbf{a} = (544, 204, 154, 101)$, Bradley's algorithm generates the solution vector $\mathbf{x} = (1700, -5100, 750, 1)$. Observe that the coefficients in $\mathbf{x}$ are quite large compared to the original input vector $\mathbf{a}$. We can perform a simple *coefficient reduction* technique, also due to Bradley, to improve our solution. As we use it, the idea is to transform the equation $ax + by = m$ to $ax' + by' = m$ where x',y' are smaller than x, y in absolute value. It works as follows. Let $g = \gcd(a, b)$. Note that m is always a multiple of g. Also, WLOG we assume $|a| < |b|$. We then set

$$\begin{aligned} q &:= \lfloor x/(b/g) \rfloor_0; \qquad \textit{truncate towards } 0 \\ x' &:= x - q(b/g) = x \bmod (b/g); \\ y' &:= y + q(a/g). \end{aligned}$$

Here we use the floor function subscripted by 0 to mean truncation towards 0, so that $\lfloor -2.3 \rfloor_0 = -2$.

We modify Bradley's algorithm above to apply coefficient reduction on pairs of coefficients in $\mathbf{x}$, moving from right to left. So, on the example input above, we would first reduce $154 \cdot 750 + 101 \cdot 1$, which is already reduced. Next, we reduce $204 \cdot (-5100) + 154 \cdot 750$, giving $204 \cdot (-4561) + 154 \cdot 36$. Here $a = 154$ and $b = 204$ as we must have $|a| < |b|$, so $b/g = 102$, and $750 \bmod 102 = 36$. We then reduce $544 \cdot 1700 + 204 \cdot (-4561)$ giving $544 \cdot (-10) + 204 \cdot (-1)$. The final solution vector is $\mathbf{x} = (-10, -1, 36, 1)$, a great improvement over the solution before coefficient reduction.

We wish to make one final observation in this section. If we permute the entries of the input vector $\mathbf{a}$, then Bradley's algorithm may produce a different solution vector $\mathbf{x}$ in the sense that it is not simply a permutation of the entries of the original solution vector. To illustrate the point, below are four different permutations of our example input vector, together with solution vectors as generated by Bradley's algorithm with coefficient reduction:

$$\begin{array}{ll} \mathbf{a} = (544, 204, 154, 101) & \mathbf{x} = (-10, -1, 36, 1) \\ \mathbf{a} = (204, 154, 101, 544) & \mathbf{x} = (-2, 2, 1, 0) \\ \mathbf{a} = (154, 101, 544, 204) & \mathbf{x} = (-40, 61, 0, 0) \\ \mathbf{a} = (101, 544, 204, 154) & \mathbf{x} = (237, -44, 0, 0) \end{array}$$

3. Genetic Algorithms

In this section, we present a brief overview of genetic algorithms.

The genetic algorithm is a heuristic and randomized method primarily used to solve optimization-style problems. It is based on the idea of evolution from biology. The algorithm outline is as follows:

1. Create a population of (suboptimal) solutions to your problem. Each solution is called an *individual*, and its computer representation should be

thought of as the individual's *DNA*. The *fitness* of an individual should be a measure of optimality.

2. Repeat the following steps (a generation):
   (a) *Selection*: Eliminate some portion of the population based on their fitness.
   (b) *Crossover*: Have some pairs of individuals exchange part of their DNA.
   (c) *Mutation*: With some nonzero probability, modify some portion of each individual's DNA.

   Stop when either a sufficiently optimal solution is found, or if no progress has been made during the last few iterations.

The algorithm performs a controlled, randomized search of the space of all possible individuals. If all goes well, the algorithm should converge towards an optimal solution, although it is difficult to prove this will happen. The challenge in designing a good genetic algorithm is in choosing how to represent an individual's DNA, and in defining the three genetic operators (selection, crossover, mutation) so that they are fast to compute and also push the population in the right direction. Once all this is done, there still remains many parameter choices to make. One must choose an initial population size, fitness cutoffs, and crossover and mutation probabilities. These parameters greatly affect the convergence rate of the algorithm.

We now briefly discuss some of the more common ways to define the genetic operators. There are, of course, many methods that we do not mention.

Selection. There are two fundamentally different ways to perform selection: proportionate selection and rank selection. In proportionate selection, each individual is included in the next generation with a probability proportional to its fitness as compared to the average fitness of the population. Thus, a highly fit individual is likely, but not guaranteed, to survive from one generation to the next. In rank selection, in effect the entire population is sorted by fitness, with a fixed number of the highest fitness individuals surviving to the next generation. Thus, a highly fit individual is guaranteed to survive. In both methods, the surviving individuals are often "cloned" at random to fill out the population to its former size.

Mühlenbein [1997] argues that, in general, rank selection is superior to proportionate selection. However, the implied sorting of the population can slow down the algorithm considerably.

Crossover. With some fixed probability, adjacent individuals exchange parts of their DNA. One could choose a point (or locus) at random along the DNA strand, and the section to the right of that point is swapped. Multiple-point crossover is also possible. Here, several crossover points are selected, and alternating sections are swapped.

Mutation. With a fixed probability, each locus (bit or word) of the the DNA strand will mutate. If the DNA locus is a bit, mutation will take the form of a bit flip. If the DNA locus is more complex, then the form mutation takes can vary. Mutation should change the individual, but ideally the fitness should be affected only to a small degree. The literature largely agrees on choosing a mutation probability that is inversely proportional to the length of the DNA strand, so that some limited mutation is likely.

For a more thorough introduction to genetic algorithms and further discussion on the relative merits of the available choices for genetic operators, see, for example, [Bäck, 1996, Michalewicz, 1996, Mitchell, 1996, Mühlenbein, 1997].

4. First Attempts

It is not customary in research articles to discuss failures. We break with this tradition here because we learned interesting things from our first two algorithms, despite the fact that the first algorithm was a complete failure, and the second was only a partial success.

In our first attempt at designing a genetic algorithm for the extended GCD problem, we chose the most obvious representation. We used a solution vector as an individual's DNA, with fitness defined as inversely proportional to the norm. Back in Section 2, we observed that permuting the input vector often produced different solution vectors. So we created our initial population by randomly permuting the input vector $\mathbf{a}$ and using Bradley's algorithm to compute different solution vectors. We then defined our genetic operators as follows:

(a) *Selection*: We used rank selection; the more fit half of the population was kept for the next generation.

(b) *Crossover*: Here we combined the crossover operation with the process of filling out the population to its former size. We would randomly choose 3 individuals that survived selection, say $\mathbf{x}$, $\mathbf{x}'$, and $\mathbf{x}''$, and compute a new individual as $\mathbf{x}+\mathbf{x}'-\mathbf{x}''$. It is easy to see that this new individual is in fact a solution to the extended GCD problem, for

$$\begin{aligned}\sum_{i=1}^{n} a_i(x_i + x_i' - x_i'') &= \sum_{i=1}^{n} a_i x_i + \sum_{i=1}^{n} a_i x_i' - \sum_{i=1}^{n} a_i x_i'' \\ &= g + g - g \quad = \quad g.\end{aligned}$$

(c) *Mutation*: A solution vector $\mathbf{x}$ was mutated by choosing two integers i, j, with $1 \le i < j \le n$ at random, and replacing $(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n)$ with $(x_1, \ldots, x_i + a_j, \ldots, x_j - a_i, \ldots, x_n)$. It is easy to show this preserves the GCD.

To our dismay, no matter how we modified the various parameters, we discovered the best solution produced by this algorithm was always present in the

initial population. This is because members of the initial population consisted of vectors with many zeros, so that mutations and crossovers almost always made individuals significantly worse. (Recall that mutations should have a relatively small effect on fitness.) In other words, generating random permutations of the input was a better method than our genetic algorithm.

Naturally, this observation led to a simple but powerful idea. Apply the genetic algorithm not to the solution vectors themselves, but to the *algorithm* for computing the solution vectors: in this case, permutations.

For our second attempt, we represented an individual as a permutation on n elements. The fitness of an individual is measured by first applying the permutation to the input vector $\mathbf{a}$, computing a solution vector $\mathbf{x}$, and then taking the norm of $\mathbf{x}$. Again, fitness was inversely proportional to the norm. As the set of permutations on n elements form a group under function composition, permutations have lots of nice mathematical properties which we used in defining our genetic operators.

(a) *Selection*: We used rank selection, retaining about 40% of the previous generation. We then composed 2–6 individuals chosen at random to create new individuals to fill out the population to its previous size.

(b) *Crossover*: With roughly a 20% probability per individual, we chose a second individual at random, and performed crossover by writing both permutations as products of 2-cycles. We used single-point crossover on this product-of-2-cycles representation. For example, if our two individuals were (1234) and (124), we first write them as products of 2-cycles: (14)(13)(12) and (14)(12); if the crossover point is after the second 2-cycle, we get (14)(13) = (134) and (14)(12)(12) = (14).

(c) *Mutation*: With roughly a 20% probability, we composed a permutation with a random 2-cycle.

Our initial population had size 12, independent of n.

The parameters mentioned above (12, 40%, 20%, etc.) were calculated by the third author. As part of his senior honors thesis, he wrote a second genetic algorithm, whose purpose was to optimize the parameters of our permutation-based genetic algorithm for the extended GCD problem. Here, an individual was a list of parameter values, and the fitness of an individual was the time needed for the extended GCD genetic algorithm to obtain a solution below a fixed norm using the given parameter settings.

We have a few comments to make about our permutation-based genetic algorithm:

- The algorithm uses $O(mn \log \max\{a_i\})$ arithmetic operations per generation, where m is the population size (we used $m = 12$). As the number of generations needed was observed to be independent of n, the overall running time of the algorithm is essentially linear in n.

- The algorithm does very well when the input vector **a** has no entry exceeding 16 bits. However, 32 bit inputs yield solution vectors with norms of around 16 bits, and larger inputs led to even worse results. This problem seemed unaffected by changes in any of the parameter values or by changes in n. This performance is much worse than that observed by the sorting-GCD method, for example.
- Most of the solution vectors generated by this method have only 2 or 3 nonzero entries, and solution vectors with more than 4 or 5 nonzero entries were extremely rare. So in effect, the permutations were actually used by the algorithm to find small subsets of the input vector entries.

We take particular advantage of the last observation above in the following section, were we describe a genetic algorithm for finding fixed-sized subsets of the entries in the input vector.

5. Multisubset-Based Algorithms

All of our multisubset-based algorithms have the same outline.

As a preprocessing step, the input vector **a** is sorted in ascending order, and a new position 0 is inserted at the front, where we store a zero. Thus, we now have $0 = a_0 \leq a_1 \leq \cdots \leq a_n$. Our motivation for this will be explained later. We compute the GCD of the integers in **a**. If the GCD is not 1, we divide out by the common factor, as this makes the entries smaller and does not affect the solution vector **x**.

An individual I is a fixed-length list of d positions in the sorted input vector. Positions may be duplicated, and 0 may be included. An individual is generated by choosing integers uniformly in the range $0 \ldots n$ at random with replacement. Thus, if we write $z_1, z_2, \ldots, z_d$ for the positions of an individual I, then I represents the multisubset $\{a_{z_1}, a_{z_2}, \ldots, a_{z_d}\}$. A population of size m consists of the indivduals $I_1, \ldots, I_m$.

The fitness of an individual I is computed by finding the extended GCD of I's multisubset as defined above. If the GCD computed in this way is not 1, we assign a fitness of 0. If the GCD is 1, we can construct a solution vector for all of **a** simply by using 0 coefficients in **x** for those entries not included in the multisubset. The fitness is the reciprocal of the L_1 norm of this solution vector. We chose this norm because it is invariant under the duplication of entries from **a**. Scaling is not needed, as we use a rank-based selection algorithm.

Our genetic operators are defined as follows:

(a) *Selection:* We used *tournament selection*, a randomized rank-based selection algorithm. To form a new individual in the next generation, q individuals are chosen at random from the previous generation, and of those q, the most fit individual is chosen for survival. We used values for q ranging from 2 to 5. In this method, the probability an individual survives is based

on its rank, not its relative fitness. So we get some of the benefits of rank selection without the overhead of sorting the population by fitness.

(b) *Crossover:* For $0 \leq i < m/2$, with a 50% probability, individuals I_{2i+1} and I_{2i+2} perform crossover. We use a simple 2-point crossover.

(c) *Mutation:* Each entry z_j in an individual I's list is mutated with probability $1/d$. To mutate, an integer chosen uniformly from the range $-3, \ldots, 3$ is added to z_j. This effectively changes one element of the multisubset represented by I.

We can now explain why the input vector $\mathbf{a}$ is sorted. Recall that mutation should have only a small effect on the fitness of the individual. The idea is that by adding a small positive or negative integer to a position in an individual, we replace an entry in its multisubset with another entry that is about the same size. It is by no means certain this will result in a small change in fitness, but it is likely, as we expect this entry's coefficient will be about the same size as before; this is what we saw in practice.

We chose d and m, the size of an individual and the size of the population, based on the size of the maximum entry in $\mathbf{a}$; both are linear in $\log \max\{a_i\}$. The constant of proportionality depended on the particular fitness evaluation method used.

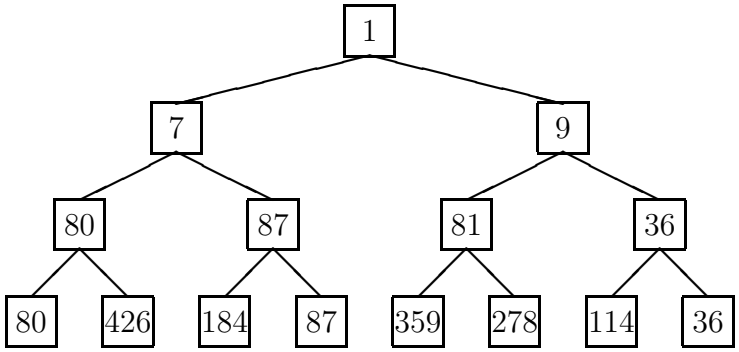
To compute the extended GCD of the multisubset of an individual (that is, evaluate its fitness), we used one of three algorithms: Bradley's algorithm, a tree-based method that we explain below, and the sorting-GCD method. Using Bradley's algorithm results in a genetic algorithm with performance similar to, but somewhat better than, the performance of the permutation-based algorithm described in the previous section.

The Tree-Based Method

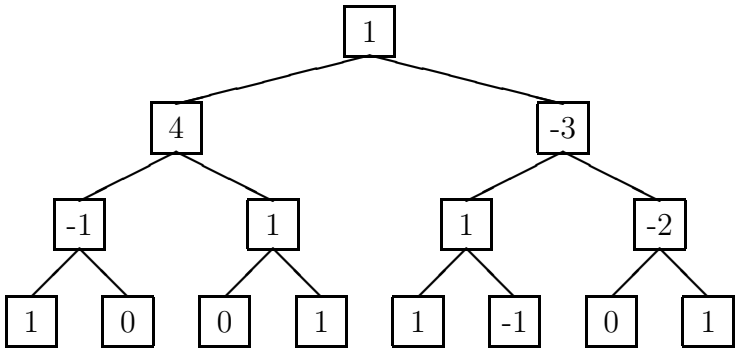
For the Tree-based GCD method, which is based loosely on the algorithm of Majewski and Havas [1994], we form a complete binary tree, with the members of the multisubset as leaves. Thus, we require that d be a power of 2.

We process the tree by level from the leaves. For purposes of describing the algorithm, we will say the root is at level 0, its children are level 1, and so forth. Thus, a node's level is the distance to the root. Each internal node stores a linear combination of its children. For levels ≥ 2, coefficients are chosen from the set $\{0, \pm 1\}$. The two nodes at level 1 are calculated using coefficients from the set $\{0, \pm 1, \pm 2\}$. The root node is calculated using a full Euclidean GCD computation. We store the coefficients with the children.

Below is an example. We use the input vector $\mathbf{a} = (80, 426, 184, 87, 359, 278, 114, 36)$. Here we have the tree of linear combinations, computed as described above:



Here is the corresponding tree of coefficients:



The idea is to get the GCD (1 in our case) at the root. We can then multiply the coefficients along each path from the root to each leaf node to obtain the coefficients of the solution vector. In our example, the solution vector would be $\mathbf{x} = (-4, 0, 0, 4, -3, 3, 0, 6)$.

This method is heuristic, in that it does not guarantee a correct value for the GCD in the root node, but in practice it gives the correct result with high enough probability to make it useful in the context of our genetic algorithm. It should be clear from the description and example above that the coefficient vectors it produces will generally have a fairly small norm.

6. Implementation Results

In this section, we present the results of our implementations of the various genetic algorithms discussed in the previous section.

To begin, we wish to demonstrate how the populations evolve under the three algorithms. For each of these plots, we used $n = 10000$, and the entries in $\mathbf{a}$ were chosen uniformly at random between 1 and e^{30}. For each genetic algorithm, we have the generation number along the x-axis, with the L_1 norm along the y-axis. We plot the average norm of the population at each generation, the reciprocal of the average fitness (recall the fitness of $\mathbf{x}$ is $1/\|\mathbf{x}\|$), and the norm of the best individual found so far. The averages are only over those individuals that represent correct solutions to the problem.

First, we have the genetic algorithm using Bradley's algorithm in Figure 1. In

Figure 1: Genetic Algorithm Evolution

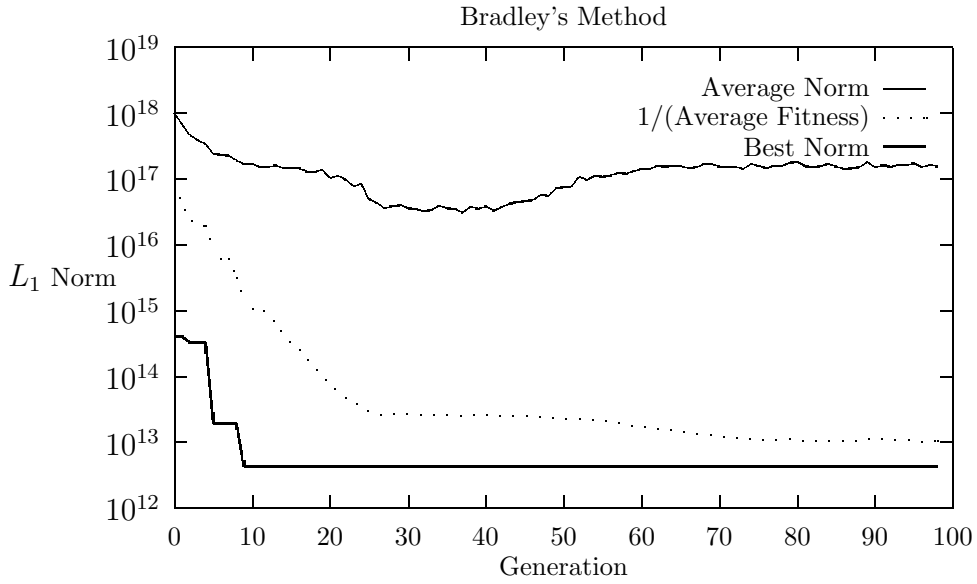


Figure 2: Genetic Algorithm Evolution

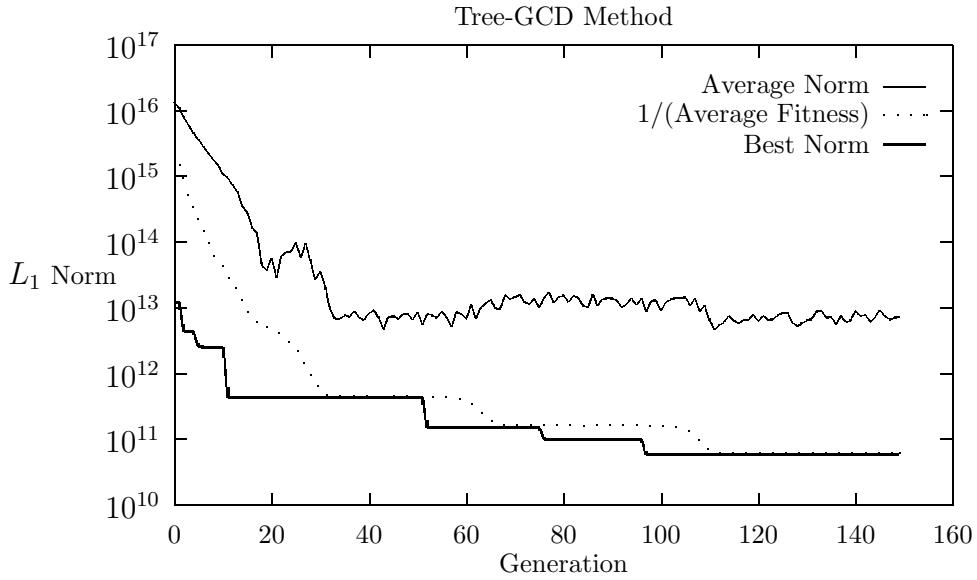


Figure 3: Genetic Algorithm Evolution

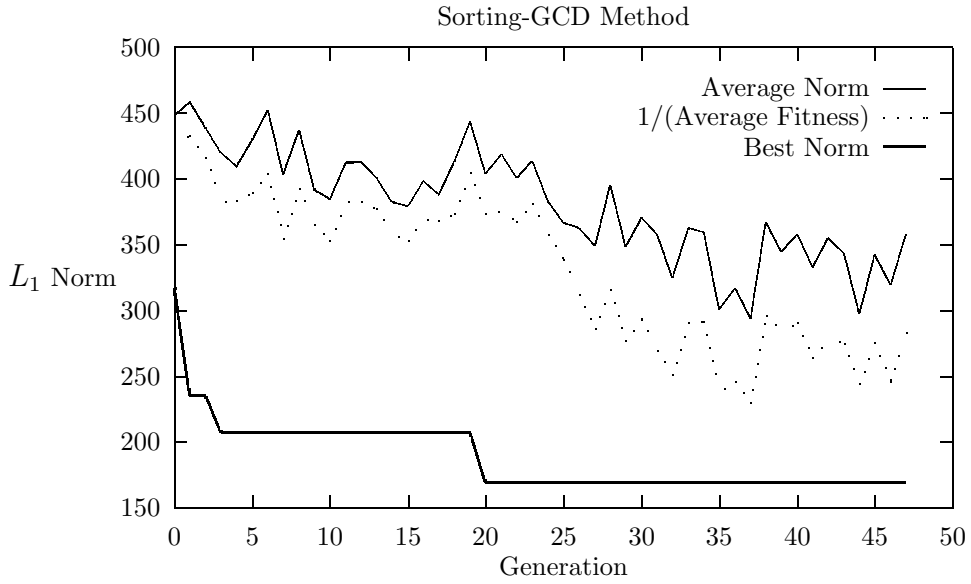


Figure 2, we have the tree-GCD genetic algorithm, which was explained in some detail in the previous section. In Figure 3, we have the genetic algorithm based on the sorting-GCD method. In all three, one can see a period of evolution early on, with the average fitness reaching a plateau. The algorithms stop once they detect they have reached a plateau. This is done by keeping a history variable λ, which is initialized to 0, and after each generation, we set $\lambda = (0.8)\lambda + 0.2f$, where f is the average fitness of the current generation. If λ has not improved in the last 10 generations, the algorithm stops. In essence, Figures 1–3 show that the genetic algorithms work.

We implemented 5 algorithms: Bradley's algorithm, the sorting-GCD method, and the 3 genetic algorithms mentioned above. All 5 algorithms were run on the same set of pseudorandom data. We chose integers uniformly from the range $1 \ldots \text{limit}$, where limit was $\lfloor e^i \rfloor$, $i = 5, 10, 15, 20, 25, 30, 35, 40, 43$. We chose powers of e for compatibility with results from other papers (for example Majewski and Havas [1995a]). The lengths of our input vectors were $n = 10, 100, 1000, 10000, 100000$. Note that the sorting-GCD method could not run for $n \geq 10000$ due to memory restrictions (it uses space proportional to n^2). In Figures 4–9, we present the average L_1 norms obtained by the various algorithms.

Figure 4: Extended GCD Norm Results

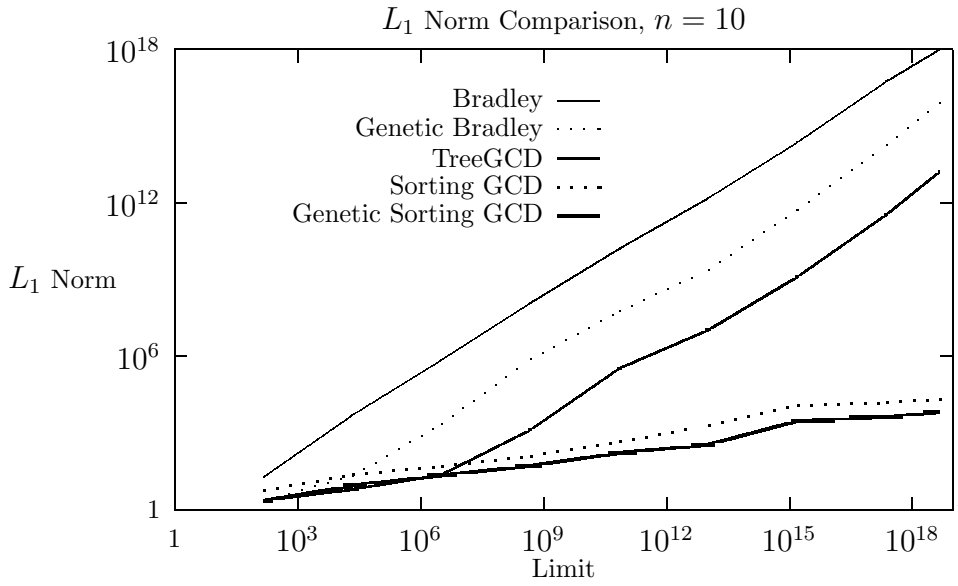


Figure 5: Extended GCD Norm Results

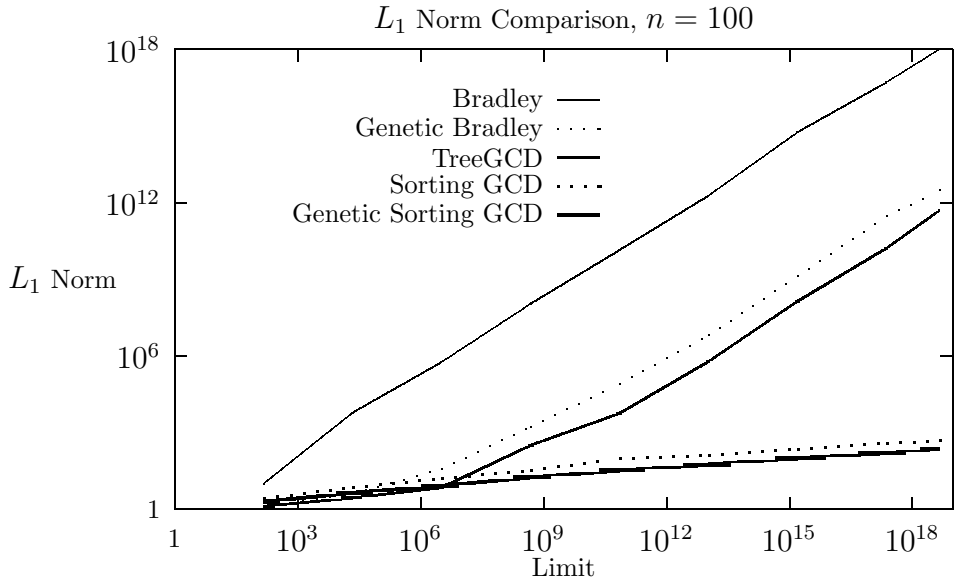


Figure 6: Extended GCD Norm Results

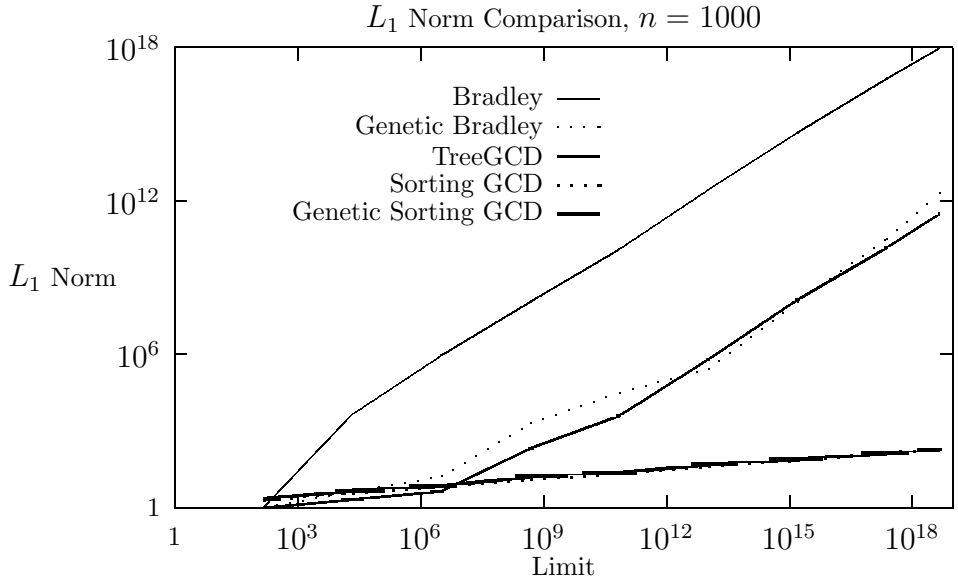


Figure 7: Extended GCD Norm Results

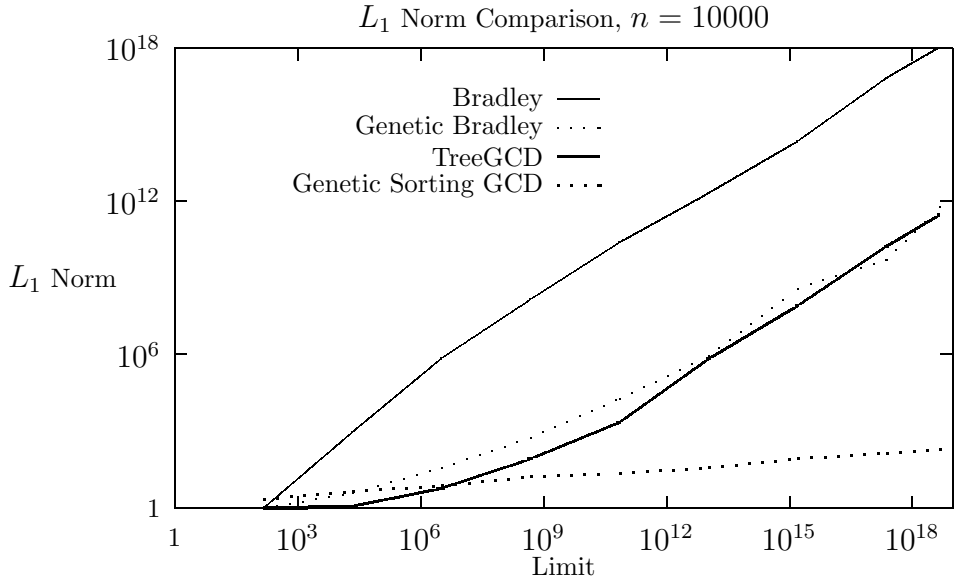
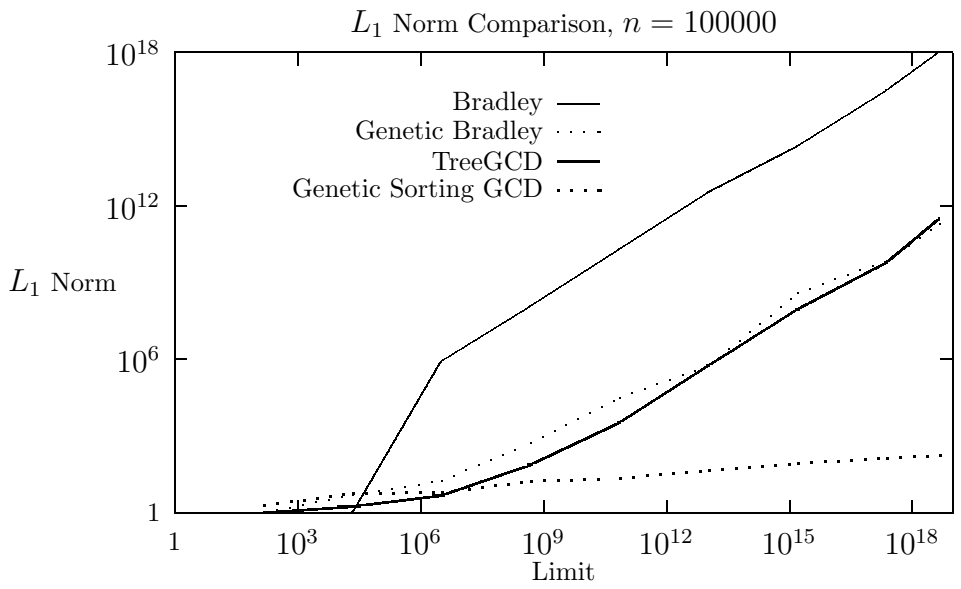


Figure 8: Extended GCD Norm Results



In Tables 1–5, we give samples of the data we obtained, including all four metrics/norms mentioned in the introduction. Note that changing the parameters of the genetic algorithms does affect the norms obtained.

Our algorithms were optimizing the L_1 norm and paid no attention to any of the other norms. Optimizing for the L_∞ norm or the L_2 norm would lead to different results, and would require some changes to the algorithm. For example, duplicate entries in the multisubsets would need to be removed for fitness calculation. However, we expect the relative performance of the algorithms would be the same when optimizing for the L_2 or L_∞ norms. For the L_0 metric, genetic algorithms are not really necessary, because in practice we can almost always find a solution vector with 2 nonzero entries; simply find two entries from $\mathbf{a}$ that are relatively prime.

Our code was written in C++, and was run on a Pentium II 233MHz, running Red Hat Linux 5.1, kernel version 2.0.34. We used the `g++` compiler, with `-O3` optimization. The data in the tables above represent averages of 10 pseudorandom inputs.

Although not of primary interest, we found that among the genetic algorithms, the genetic Bradley algorithm was fastest, with the genetic sorting-GCD next, and the tree-GCD genetic algorithm was the slowest.

Finally, all code used to generate the data shown here is available from the second author's web page at `http://www.butler.edu/~sorenson`.

Table 1: Bradley's Algorithm

Limit	n	L_0	L_1	L_2	L_∞	*Generations*
exp[10]	10	2.4	5131.1	3722.29	3003.3	
exp[10]	10^2	2.2	5895.2	4443.13	3831.7	
exp[10]	10^3	2.3	4539.4	3307.83	2686.3	
exp[10]	10^4	1.4	973.5	735.788	661.3	
exp[10]	10^5	1	1	1	1	
exp[20]	10	2.7	124223424	89773333	71007916	
exp[20]	10^2	2.6	109098202	83238293	74979714	
exp[20]	10^3	2.3	115429610	85359830	72385780	
exp[20]	10^4	2.4	1.46488e+08	1.09034e+08	9.2149e+07	
exp[20]	10^5	2.3	1.22298e+08	9.37944e+07	8.26743e+07	
exp[30]	10	2.6	1.52625e+12	1.23083e+12	1.13388e+12	
exp[30]	10^2	2.8	1.84404e+12	1.44551e+12	1.32253e+12	
exp[30]	10^3	2.3	2.79907e+12	2.17665e+12	1.92957e+12	
exp[30]	10^4	2.8	2.09763e+12	1.56845e+12	1.39493e+12	
exp[30]	10^5	2.4	3.47518e+12	2.5372e+12	2.06808e+12	
exp[40]	10	2.4	5.60223e+16	4.29385e+16	3.7051e+16	
exp[40]	10^2	2.2	5.13667e+16	3.81264e+16	3.24796e+16	
exp[40]	10^3	2.1	5.99606e+16	4.68138e+16	4.21136e+16	
exp[40]	10^4	2.4	6.60131e+16	4.82894e+16	4.04037e+16	
exp[40]	10^5	2.7	3.29392e+16	2.44134e+16	2.04052e+16	

Table 2: The Genetic Bradley Algorithm

Limit	n	L_0	L_1	L_2	L_∞	*Generations*
exp[10]	10	3.5	21.5	14.2564	12	66.6
exp[10]	10^2	2.8	4.1	2.60524	2.1	71.8
exp[10]	10^3	2.6	4.3	2.78787	2.3	72.6
exp[10]	10^4	2.5	3.7	2.4155	1.9	68.3
exp[10]	10^5	2.7	4.9	3.29406	2.8	66
exp[20]	10	3.2	779689	574167	453636	64.8
exp[20]	10^2	3.8	1661.8	1263.69	1126.4	75.7
exp[20]	10^3	3.6	1967.2	1513.35	1313.7	72.5
exp[20]	10^4	3.9	563.2	408.299	360.6	80
exp[20]	10^5	4	546.5	429.945	406.1	74.5
exp[30]	10	3.2	2.45731e+09	1971803100	1781246000	71.8
exp[30]	10^2	4	6.39582e+06	5.15079e+06	4866526	86.6
exp[30]	10^3	4.3	265942	208853	190929	112.1
exp[30]	10^4	3.9	874880	669073	590272	96.3
exp[30]	10^5	4.3	526034	423951	394962	120.6
exp[40]	10	3.4	1.89823e+14	1.48684e+14	1.31929e+14	71
exp[40]	10^2	3.9	2.87416e+11	2.40097e+11	2.29689e+11	96.3
exp[40]	10^3	3.9	3.26761e+10	2.60485e+10	2.47175e+10	91.4
exp[40]	10^4	4	5.09587e+09	4.27824e+09	4.12371e+09	98.5
exp[40]	10^5	3.8	7.47944e+09	6.43911e+09	6.20406e+09	78.5

Table 3: The Tree-GCD Genetic Algorithm

Limit	n	L_0	L_1	L_2	L_∞	*Generations*
exp[10]	10	5.1	6.3	2.9647	1.8	64
exp[10]	10^2	2.9	2.9	1.69528	1	61
exp[10]	10^3	2	2.1	1.48676	1.1	57.4
exp[10]	10^4	1.2	1.2	1.08284	1	57.3
exp[10]	10^5	1.7	1.8	1.37213	1.1	64.4
exp[20]	10	7.5	1368.4	712.827	517.4	86.4
exp[20]	10^2	10.4	313.4	136.995	75.2	89.6
exp[20]	10^3	7.6	213.4	70.7983	37.7	92.6
exp[20]	10^4	8.3	85	36.9939	23.6	86
exp[20]	10^5	6.9	76.5	37.6748	25.6	85.2
exp[30]	10	8.2	1.09776e+07	4.97237e+06	2.95693e+06	104.7
exp[30]	10^2	14.1	603532	181316	84965.4	86.9
exp[30]	10^3	12	642316	212216	87741	85
exp[30]	10^4	9.1	684442	246797	109648	81
exp[30]	10^5	9.8	577347	210681	105401	77.8
exp[40]	10	8.1	3.57135e+11	1.47626e+11	9.21625e+10	102.2
exp[40]	10^2	15.3	1.6656e+10	5.43374e+09	2.73079e+09	102.8
exp[40]	10^3	14.2	1.42869e+10	5.00247e+09	2.54945e+09	97.2
exp[40]	10^4	10.2	1.71791e+10	1.192e+10	1.05225e+10	88.3
exp[40]	10^5	8.7	6.16643e+09	2.2328e+09	1242068050	93.2

Table 4: The Sorting-GCD Algorithm

Limit	n	L_0	L_1	L_2	L_∞	*Generations*
exp[10]	10	7.9	23.9	10.0037	6.5	
exp[10]	10^2	7.2	7.2	2.66418	1	
exp[10]	10^3	3.8	3.8	1.94142	1	
exp[20]	10	9.8	123.9	47.6979	27.6	
exp[20]	10^2	30.8	34.3	6.35692	1.9	
exp[20]	10^3	12.6	12.6	3.51751	1	
exp[30]	10	10	1984.5	784.748	526.3	
exp[30]	10^2	76.2	124.5	16.2528	4.8	
exp[30]	10^3	42.3	43.1	6.65854	1.4	
exp[40]	10	10	15140.1	6062.39	3795.2	
exp[40]	10^2	89.5	380.1	48.2272	13.4	
exp[40]	10^3	135.8	141.4	12.2563	2.1	

Table 5: The Genetic Sorting-GCD Algorithm

Limit	n	L_0	L_1	L_2	L_∞	*Generations*
exp[10]	10	6	9	3.93723	2.5	45
exp[10]	10^2	4.4	4.6	2.20733	1.2	45.4
exp[10]	10^3	4.6	4.7	2.20395	1.1	40.7
exp[10]	10^4	4.3	4.5	2.18478	1.1	42.9
exp[10]	10^5	5.2	5.5	2.45082	1.3	46.4
exp[20]	10	9.6	54.6	22.0814	13.5	47.4
exp[20]	10^2	16.5	18.2	4.66055	2	37.8
exp[20]	10^3	15.3	17.1	4.52248	1.8	40.9
exp[20]	10^4	16.2	17	4.28985	1.5	45.1
exp[20]	10^5	15.4	16.8	4.40803	1.8	42.7
exp[30]	10	10	355.3	138.539	83.5	56.3
exp[30]	10^2	42.8	55	9.13402	3.1	51.4
exp[30]	10^3	44.1	49.7	7.80123	2.1	47.2
exp[30]	10^4	35.6	38.4	6.61106	2	47.1
exp[30]	10^5	38.7	44	7.37274	2.2	45.1
exp[40]	10	10	4275.4	1695.46	1078	54.1
exp[40]	10^2	61.7	163.8	24.5038	7.9	55.7
exp[40]	10^3	85.4	135.5	16.3212	4	46
exp[40]	10^4	90.2	138.8	16.2389	4	38.2
exp[40]	10^5	86.4	138.1	16.8467	4.7	50

7. Summary

In this paper, we have presented a multisubset-based framework for constructing genetic algorithms for the extended GCD problem. This framework allows for the "plugging in" of any reasonable deterministic algorithm or heuristic method for computing the extended GCD. The genetic approach is used to look for an optimal subset of the entries of the input vector to feed to the algorithm that was "plugged in."

In conclusion, it seems fair to say that it is possible to construct genetic algorithms for the extended GCD problem that perform better than known deterministic methods for this hard problem. In practice, our genetic sorting-GCD algorithm was the best overall, obtaining better coefficients that the sorting-GCD method alone.

References

Eric Bach and Jeffrey O. Shallit. *Algorithmic Number Theory*, volume 1. MIT Press, 1996.

Thomas Bäck. *Evolutionary algorithms in theory and practice.* Oxford University Press, 1996.

W. A. Blankinship. A new version of the Euclidean algorithm. *American Mathematical Monthly*, 70:742–745, 1963.

Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Communications of the ACM*, 13(7):433–436, 1970.

David Ford and George Havas. A new algorithm and refined bounds for extended GCD computation. In *Second International Algorithmic Number Theory Symposium*, pages 145–150. Springer-Verlag, 1996. LNCS 1122.

George Havas and Bohdan S. Majewski. A hard problem that is almost always easy. In *Algorithms and Computation*, pages 216–223. Springer-Verlag, 1994. LNCS 1004.

George Havas, Bohdan S. Majewski, and Keith R. Matthews. Extended GCD algorithms. Technical Report 302, University of Queensland, 1995.

George Havas, Bohdan S. Majewski, and Keith R. Matthews. Extended GCD and Hermite normal form algorithms via lattice basis reduction. *Experimental Mathematics*, 7(2):125–136, 1998.

Tudor Jebelean. A generalization of the binary GCD algorithm. In M. Bronstein, editor, *1993 ACM International Symposium on Symbolic and Algebraic Computation*, pages 111–116, Kiev, Ukraine, 1993. ACM Press.

Tudor Jebelean. A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers. *Journal of Symbolic Computation*, 19:145–157, 1995.

Charles Lam, Jeffrey O. Shallit, and Scott Vanstone. Worst-case analysis of an algorithm for computing the greatest common divisor of n inputs. In J. Buchmann, T. Hoholdt, H. Stichtentoth, and H. Tapia-Recillas, editors, *Coding Theory, Cryptography and Related Areas*, pages 156–166. Springer-Verlag, 2000.

Bohdan S. Majewski and George Havas. The complexity of greatest common divisor computations. In *First International Algorithmic Number Theory Symposium*, pages 184–193. Springer-Verlag, 1994. LNCS 877.

Bohdan S. Majewski and George Havas. Extended GCD calculation. *Congressus Numerantium*, 111:104–114, 1995a.

Bohdan S. Majewski and George Havas. A solution to the extended gcd problem. In A. H. M. Levelt, editor, *1995 ACM International Symposium on Symbolic and Algebraic Computation*, July 1995b.

Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 3rd edition, 1996.

Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

Heinz Mühlenbein. Genetic algorithms. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, chapter 6, pages 137–171. Wiley, 1997.

Thom Mulders and Arne Storjohann. The modulo n extended GCD problem for polynomials. In *1998 ACM International Symposium on Symbolic and Algebraic Computation*, pages 105–112, 1997.

Carsten Rössner and Jean-Pierre Seifert. The complexity of approximate optima for greatest common divisor computations. In *Second International Algorithmic Number Theory Symposium*, pages 307–322. Springer-Verlag, 1996. LNCS 1122.

A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.

Jonathan P. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16: 110–144, 1994.

Jonathan P. Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In A. H. M. Levelt, editor, *1995 ACM International Symposium on Symbolic and Algebraic Computation*, pages 254–258, Montreal, Canada, July 1995. ACM Press.

Arne Storjohann. A solution to the extended GCD problem with applications. In *1997 ACM International Symposium on Symbolic and Algebraic Computation*, pages 109–116, 1997.

Ken Weber. The accelerated integer GCD algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.