Faster Shellsort Sequences: A Genetic Algorithm Application

Richard Simpson
simpson@cs.mwsu.edu

Shashidhar Yachavaram
shashi@cs.mwsu.edu

Department of Computer Science
Midwestern State University
3410 Taft,
Wichita Falls, TX 76308
Phone: (940) 397-4191 Fax: (940) 397-4442

Abstract

The recent popularity of genetic algorithms (GA's) and their application to a wide variety of problems is a result of their ease of implementation and flexibility. Evolutionary techniques are often applied to optimization and related search problems where the fitness of a potential result is easily established. Problems of this type are generally very difficult and often NP-Hard, so the ability to find a reasonable solution to these problems within an acceptable time constraint is clearly desirable.

One such problem that has been researched thoroughly is the search for Shellsort sequences. The attributes of this problem make it a prime target for the application of genetic algorithms. Noting this, the authors have designed a GA that efficiently searches for Shellsort sequences that are top performers. This research has resulted in the discovery of several new sequences that are statistically 3% more efficient, from the standpoint of comparison counting, than the best sequences known.

Key Words: Genetic Algorithms, Shellsort, Complexity Analysis

1. Introduction

Shellsort is a sorting algorithm that has been the focus of considerable research. Determination of the complexity of this algorithm can be extremely complex. The algorithm sorts using an integer sequence of numbers to define the order of item comparison. Consequently, the complexity of Shell sorting is heavily dependent on this sequence. Although the complexity of Shellsort has been determined for some sequences, an optimal sequence together with its corresponding complexity has not been discovered.

A method that shows promise in the search for these sequences is Genetic Algorithms (GA's). A GA is a evolutionary procedure used to search for solutions to complex problems that resist normal research approaches. This paper concerns itself with the application of Genetic Algorithms to the problem of finding new, more efficient Shellsort sequences. Using comparison counts as the main criteria for quality, several new sequences have been discovered that sort one megabyte files more efficiently than those presently known.

This paper provides a brief introduction to genetic algorithms, a review of research on the Shellsort algorithm, and concludes with a description of the GA approach and the associated results.

2. Genetic Algorithms

John Holland developed the technique of GA's in the 1960's. The original intent was to study biological adaptation and ways this could be simulated within computer systems. In his innovative 1975 book, *Adaptation in Natural and Artificial Systems*(Holland, 1975), Holland introduced population-based algorithms together with the closely associated concepts of chromosomes, crossover operators and mutation. In this book he made the first attempt to analyze evolutionary computation theoretically by studying the concept of schemas. This original work, together with the research of others, such as Rechenberg(Rechenberg, 1997), resulted in the creation of many different branches of evolutionary computation. In addition to genetic algorithms these include work in genetic programming(Koza, 1992), evolution strategies(Rechenberg, 1997) and attempts to justify biological evolutionary theories using computer simulation(Ackley & Littman, 1992)(Hinton & Nowlan, 1987).

The pioneering work by these researchers and many others has resulted in a broad collection of tools and techniques that modern computer professionals can use in addressing some of the hardest problems faced today. Many of these problems have no polynomial time solutions and, consequently, can only be addressed using heuristics, genetic

algorithms and related schemes that generate reasonable, but not necessarily, optimal solutions. The application of evolutionary computation has been applied in the areas of optimization, automatic programming, machine learning, evolution and learning, population genetics and biochemistry, just to name a few.

The basic genetic algorithm includes a population of chromosomes and the operations selection, crossover, and random mutation. The application of genetic algorithms to a problem first involves the representation of solutions to the problem in the form of chromosomes. Each *chromosome* is a bit string or some other data structure (e.g. tree) that is mapped to a single solution of the problem. A collection of chromosomes, often referred to as a *population*, defines a temporary set of possible solutions that is continually modified. The quality of a chromosome is referred to as its *fitness* and is generated by a *fitness function*. A GA repeatedly processes the population, creating new generations by applying the operations of selection, crossover and mutation. The *selection operator* finds individuals in the population which have high fitness and returns them for crossover. Two fit individuals, called *parents*, are then combined to produce *offspring*. This *crossover mechanism* is often as simple as swapping sub-strings of bits found in the parents' chromosomes. The last step in this process is *mutation*, a random, low probability, flipping of bits of the offspring chromosome. The main purpose of this operator is to increase the diversity of individuals within a population. Each new population, a *generation*, will hopefully increase in average fitness over time. At a point determined by a termination criterion, the algorithm terminates and displays the best chromosomes discovered thus far. The combination of these steps results in the following simple genetic algorithm.

BASIC GA

Begin with a random population of chromosomes called the *current-population*.
Repeat until termination criteria are satisfied
    Apply the fitness function to each chromosome in the *current-population*
    Repeat until *n* offspring have been created
        Select a pair of parent chromosomes based on fitness
        Apply a crossover operator to this pair generating 1 or more children
        Mutate these children with low probability
        Add the children to *new-population*
    End-repeat
End-repeat.

3. Sorting and the Shell Sort Problem

In 1959, Donald L. Shell(Shell, 1959) proposed an interesting sorting algorithm now known as the Shellsort. The algorithm, sometimes known as the diminishing increment sort, is a repeated application of the insertion sort. Insertion sort is an $O(n^2)$ time algorithm that sorts an array, top to bottom, by inserting the i^{th} data value into the presorted values 1 to i-1 using the standard shift down to insert procedure. Shellsort begins by assigning h to represent a positive increment. The array is then partitioned into h sub-arrays containing the elements in positions i, $i+h$, $i+2h$, ... , where $i = 1, 2, .., h-1$. This creates h sub-arrays in which the elements are h positions apart. Applying the insertion sort to the h sub-arrays, defined by a sequence of decreasing increments of h, the file is efficiently sorted.

The list of decreasing h values forms a sequence that controls the execution of the Shellsort algorithm. The complexity of this sorting algorithm varies with the sequence. While the worst case complexity of specific sequences are known, the general complexity of Shellsort and the average running time as a function of N are still open problems(Sedgewick, Analysis of Shellsort and Related Algorithms, 1996).

Pratt, an early researcher of Shellsort, calculated that the running time of Shellsort is $O(N^{3/2})$ for the increment sequence 1, 3, 7, 15, 31, 63, 127,. ..., 2^N-1(Pratt, 1972). Sedgewick raised the bound by showing that the sequence of h values 1, 8, 23, 77, 281, 1073, 4193,. .. = $4*4^k + 3*2^k + 1$, generates an algorithm which runs in $O(N^{4/3})$(Sedgewick, A new upper bound for Shellsort, 1986).

Recent research by Weiss has resulted in empirical evidence that suggests "almost" geometric sequences perform better than others(Weiss & Sedgewick, More on Shellsort Increment Sequences, 1990)(Weiss & Sedgewick, Tight Lower Bounds for Shellsort, 1990)(Weiss, Empirical Study of the Expected Running Time fo Shellsort, 1991)(Weiss, Shellsort with a Constant Number of Increments, 1996). These are sequences that have terms that differ from a normal geometric sequence by a small amount.

The efficiency of Shellsort is such that it is works very well for file sizes that approach 106 data values. At this point the algorithm begins to run more slowly than the better-known $O(n \log n)$ sorting algorithms such as Quicksort. The best average case sequence known to the authors at this time is 1, 5, 19, 41, 109, 209, ...(Sedgewick, A new upper bound for Shellsort, 1986). This sequence can be generated using the following heuristic developed by Knuth(Knuth, 1998).

$h_j = 8*2^j - 6*2^{(j+1)/2} + 1$, if j is odd
$h_j = 9*2^j - 9*2^{j/2} + 1$, if j is even

This sequence, as well as several others, is used for comparison with the sequences generated by the genetic algorithm employed by this paper.

4. Research Methods and Results

The support software used in this research is a object oriented C++ genetic algorithm library created by Matthew Wall of MIT. The software and necessary documentation can be readily downloaded via ftp from ftp://lancet.mit.edu/pub/ga/. This library is very generalized and contains the necessary support classes to build integer list chromosomes as required in defining a Shellsort sequence. Use of this library simplified the task of developing the algorithm and is highly recommended by the authors.

The first stage of the investigation was the search for sequences that work well for data sets of size $N=10^3$. For this experiment a chromosome is defined to be an integer array that contains the first seven numbers in the sequence. These short seven integer sequences are referred to as tail sequences. The fitness of each tail chromosome was determined by running Shellsort using the sequence defined by the chromosome on ten different random files. The number of comparisons made during sorting were counted and averaged over the ten files. The fitness of each chromosome was then defined as a function of this number, with a smaller comparison count being a higher fitness.

After running the GA on an initial population of randomly generated sequences, a collection of top performing tail sequences was generated. From these initial attempts an optimized GA was then designed and constructed to perform a more efficient search. The improved version generated the seven number sequence 1, 4, 13, 36, 83, 219, 893. This sequence allows sorting of data sets up to 1000 items and is the highest performing tail sequence the authors have discovered to date.

Sorting files of size one megabyte requires longer sequences. For the purpose of this paper, a length of fifteen was selected. This was nothing more than an average of the lengths of several sequences used in practice. Extending this sequence to fifteen integers was hindered by the runtime associated with the fitness calculation of megabyte files. The use of ten files for testing was abandoned and a single, constantly changing, one-megabyte file was used instead. Initially, a large population of fifteen integer chromosomes, having the above efficient tail set as constant, was randomly generated. After each mutation the chromosomes were sorted to maintain the feasibility of the sequence. Applying the GA to this population resulted in a set of sequences that had reasonable fitness. The maximum and minimum values found for each of the upper eight positions in this collection were determined. This resulted in a suitable set of intervals that were used to restrict random selection during the mutation process. This had the desirable side effect of eliminating the sorting required during the first phase. Using these methods, an optimized version of the GA was developed that restricted each value in a chromosome to the associated intervals. This optimization allowed the average fitness in the populations to increase much faster than in the completely random case.

The three best sequences found to date using the GA approach are as follows:

Seq1 = {422399,118473,51731,26336, 13113,5733, 2550,1513,893,219,83,36,13,4,1},

Seq2 = {577357, 206129, 79243, 33706, 11542, 6295, 3323, 1562, 893, 219, 83, 36, 13, 4, 1}, and

Seq3= {575131, 237406, 105612, 38291, 22927, 8992, 3568, 1488, 893, 219, 83, 36, 13, 4, 1}.

From the standpoint of comparison counting these sequences perform better than published sequences known to the authors.

For comparison purposes, two well-known sequences were used []. The first is a seventeen number sequence by Sedgewick. The second is by Incerpi and Sedgewick which consists of fifteen numbers. The sequences are as follows.

#1: {587521, 260609, 146305, 64769, 36289, 16001, 8929,3905, 2161, 929, 505, 209, 109, 41, 19, 5, 1}

#2: {1391376, 463792, 198768, 86961, 33936, 13776, 4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1}

The Sedgewick sequence appears to be as fast as any sequence that the authors were able to find in the literature.

These sequences were compared to Seq1 and Seq3 by sorting ten random files and averaging the number of comparisons (times 1000) that were generated in each case. Table 1 gives the average number of comparisons that were made by the indicated sorting algorithm when applied to these files. Data set sizes ranged from 12,500 words to 800,000 words.

Table 1: Average number of comparisons (*1000).

N	Incerpi-S	Sedgewick	Seq1	Seq3
12500	673	641	633	626
25000	1485	1407	1387	1363
50000	3258	3056	3019	2982
100000	7107	6617	6527	6435
200000	15313	14196	13988	13803
400000	32889	30426	29640	29516
800000	70511	64699	62957	62766

Analysis of the above table indicates that Seq3 provides the best improvement over Sedgewick, requiring 3% fewer comparisons on average when sorting 800,000 data items.

In an attempt to obtain additional evidence of this sequence's superiority to existing sequences the authors

downloaded Sedgewick's Shellsort performance utility. The source code was obtained from Robert Sedgewick and compiled locally using gcc with -O2 optimization on a Sun UltraSparc workstation. The runtime was computed as *utime + stime*. Table 2 summarizes the results.

The values in Table 2 are consistent with those in the Table 1. When sorting 800000 data values we find that Seq3 performs 3% faster than the Sedgewick sequence.

Table 2: Average runtimes in microseconds.

N	Incerpi-S	Sedgewick	Seq1	Seq3
12500:	1	2	2	2
25000:	4	4	4	4
50000:	10	9	9	9
100000:	22	21	21	20
200000:	50	48	47	47
400000:	114	107	105	103
800000:	251	234	230	227

5. Conclusion and Future Studies

Shellsort is a simple sorting algorithm with an extremely complex analysis and the runtime is dependent upon the choice of the increment sequences. Efficient sequences that can be used to sort data sets of size 10^3 and 10^6 would be very useful in practice. The use of Genetic Algorithms allows one to search in a reasonably efficient manner for sequences that work well for these data set sizes. The speed and ease in which new sequences were generated by the GA approach clearly indicates the potential that GA's have in addressing this problem. It also has the added benefit that once a sequence is discovered the GA need not be run again.

Continued research is now directed at improving this sequence as well as determining its complexity given that its associated generating function can be determined.

A distributed version of the above algorithm, running on PVM is in development. This version will search a much larger solution space in a shorter amount of time by using the combined processing power of multiple workstations.

Since this paper focused only on sequences of size 15, future research will be directed toward both 14 and 16 element sequences when sorting one megabyte files.

6. References

Ackley, D., & Littman, M. (1992). Interactions between learning and evolution. *Artificial Life II.* Addision-Wesley.

Hinton, G. E., & Nowlan, S. J. (1987). How learning can guide evolution. *Complex Systems* , 495-502.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Knuth, D. (1998). *The Art of Computer Programming Volume 3: Sorting and Searching.* Reading, MA: Addison-Wesley.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Pratt, V. R. (1972). *Shellsort and Sorting Networks.* Garland, New Yourk: Stanford University.

Rechenberg, I. (1997). Evolutionsstrategie:Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution. *Frommann-Holzboog* .

Sedgewick, R. (1986). A new upper bound for Shellsort. *Journal of Algorithms* , 159-173.

Sedgewick, R. (1996). Analysis of Shellsort and Related Algorithms. *Fourth Annual European Sysmposium on Algorithms*, (pp. 1-11). Barcelona.

Shell, D. L. (1959). A High Speed Sorting Algorithm. *Communications of the ACM* , 30-32.

Weiss, M. (1991). Empirical Study of the Expected Running Time fo Shellsort. *Computer Journal* , 88-91.

Weiss, M. (1996). Shellsort with a Constant Number of Increments. *Algorithmica* , 649-654.

Weiss, M., & Sedgewick, R. (1990). More on Shellsort Increment Sequences. *Information Processing Letters* , 267-270.

Weiss, M., & Sedgewick, R. (1990). Tight Lower Bounds for Shellsort. *Journal of Algorithms* , 242-251.