Improving Shellsort Through Evolution

Gene Michael Stover

created Sunday, 2002 June 16
updated Friday, 2006 July 21

Contents

1. Let P be a population of randomly generated bit-strings (DNA).
2. While not done
   (a) For each bit-string in P, generate an organism.
   (b) Find the fitness of each organism in P. Associate the organism's fitness with the bit-string from which it was generated.
   (c) Create a new generation, N, containing the same number of bit-strings as P. Each element in N is a child of parents chosen from P. Elements in P have a higher chance of being chosen as parents if their fitness values are better.
   (d) Throw-away the current P & declare N to be the new P.
3. Report the best of the current population P.

Figure 1: Pseudocode for a generic genetic algorithm

1 Background

Shellsort is an in-place sorting algorithm. It is described in many data structures books, including [6] and [1]. The implementation I used is shell.lisp. Notice that the comparison function (`lessp`) & the copy function (`copyfn`) are parameters so that other functions in the program may use this implementation of Shellsort to track the cost of a sequence of increments.

In a sense, Shellsort is a parameterized family of sorting algorithms. The parameter is a sequence of integral increments. Each increment is used for one pass through the list being sorted. The increments begin with their largest & decrease in size each time through the algorithm. The final increment must be 1.

The efficiency of Shellsort depends on the sequence of increments. The efficiency has proven difficult to analyze. The most efficient sequence of increments known was proposed by Sedgewick.

2 The Genetic Algorithm

Most genetic algorithms resemble the pseudocode in Figure 1. Genetic algorithms in general are described in [3] & in many other books.

The goal of this application of a genetic algorithm is to evolve a good sequence of increments for Shellsort. Sequences that make Shellsort run more efficiently are better than sequences which make it run less efficiently. A Shellshort's efficiency is measured by counting the number of comparisons & swaps it does.

The important parts of any application of a genetic algorithm are representation of the organisms being optimized, the method of comparing them (fitness),

the method of constructing an organism from the bit-strings manipulated by the genetic algorithm, & the methods of selection, mating, & mutation.

3 Organisms

For this application of a genetic algorithm, an organism is a sequence of increments. The increments are sorted from largest to smallest. The smallest is always 1, & there are no duplicates.

For example, an organism might be the sequence `(4 3 2 1)`. Another organism might be `(17 8 7 3 1)`. A trivial organism is `(1)`.

The sequence `(4 3 2)` is not an organism because it does not include 1. Another sequence that is not an organism is `(3 4 2 1)` because it is not sorted from largest to smallest. The sequence `(4 4 3 1)` is not an organism because it includes a duplicate 4.

I used a population size of 10,000. I chose this number as a medium between experience (which argued for 30,000 or more) & impatience (which wanted to see results as soon as possible).

4 Fitness

Fitness of an organism is the number of comparisons & swaps a Shellsort performs when parameterized with the organism. The fitness of an organism is calculated by sorting many test cases & counting the total number of comparisons & swaps. Smaller fitness values are better than large fitness values.

So they may be compared meaningfully, the fitnesses for all the organisms in a generation are calculated using the same test cases. The program keeps a list of fitness cases. Each fitness case is an array of randomly chosen integers.

When the program begins, there is exactly one fitness case. Every fifth generation, the program throws away the existing fitness cases & creates a new list of random arrays. The number of arrays it creates is $1 + min(generation/5, 20)$. In other words, the number of fitness cases is one-fifth the generation number, plus 1, to a maximum of 21 fitness cases. At least one of the new arrays is guaranteed to have a length of 500,000, but the others have randomly chosen lengths.

The first motivation behind throwing-out old fitness cases & creating new ones was to prevent the evolver from finding a sequence that was optimized only for the initial case instead of being an optimum sequence for Shellsort in general. The reason for increasing the number of fitness cases each time was to make the world progressively more difficult for the sequences.

Why every fifth generation instead of, say, every third or every tenth? I chose 5 from the air. I didn't experiment with other periods. Why limit the number of fitness cases to 21? Again, the choice was mostly arbitrary, maybe something of a gut feeling, & I did not experiment with others.

5 Encoding

The simplest data structure for an evolver to manipulate is a string of bits. The parameter being evolved for Shellsort is a list of integers, which encodes nicely as a string of bits. So I chose a string of bits as the genetic encoding. In other words, a plain string of bits forms the "DNA" for an organism.

Each organism is encoded as a single string of bits. Each increment in the organism is encoded as a fixed-width, big-endean, unsigned integer. The width of the increment is the minimum number of bits that will encode an integer at least as large as the length of the test case arrays.[1] For a given increment, the highest bit comes first, then the next highest, down to the two's bit (2^1) & then the unit's bit (2^0).

The string of bits contains enough bits to encode a large number of increments. The string never encodes a partial increment. The number of increments to encode in a string of bits is chosen as $log_2 L$, where L is the length of the longest test case the algorithm will present to the organisms. I chose $log_2 L$ somewhat arbitrarily but also because the sequence of increments suggested by Doctor Shell himself was the powers of 2, & to sort an array of length L using that sequence, Shellsort must visit $log_2 L$ increments. So I figured that $log_2 L$ was enough space to encode a good (better than Shell's) sequence.

Notice that the encoding does not include any mention of the constraints on the organism. (See Section 3 for a description of the constraints on organisms.) The constraints are enforced when the encoding (the DNA) is converted to an organism.

To convert an encoded bit-string (DNA) to an organism, first decode all the unsigned integers from the bit-string & save them in a list (or array or whatever). Limit each integer with modulo L, where L is the length of the most lengthy test case with which we'll test the organism. Insert 1 into the list to ensure that the organism satisfies the "contains a 1" constraint. Sort the list from largest integer to smallest. Remove duplicates. The result is a valid organism. The result might not be an organism that contains an efficient sequence of increments for Shellsort, but it is a valid organism in that it satisfies the constraints for organisms as described in Section 3.

6 Selection

To select a parent, the program chooses some organisms at random from the current generation. Of those chosen, the one with the best (lowest) fitness is the winner & becomes a parent.

The number of organisms chosen at random is $\frac{1}{100}^{th}$ the size of the population, or 10, whichever is larger.[2]

[1] In retrospect, this was probably over-kill. It saved some space, but not much. It increased the complexity of the code & of the description of the algorithm, if only slightly. It probably would have been adequate to settle on a width of 32 bits for the increments.

[2] The choice of "$\frac{1}{100}^{th}$ the size of the population, or 10, whichever is larger" was fairly

This is called *tournament selection*. I chose it because its implementation is simple. There are many other methods of selection, & we did not explore them.

To chose the two parents for a single new organism, the program does this tournament selection process twice. Each tournament provides one parent.

7 Mating

Mating is done using single-point crossover on the bit-strings of the parents.

Given two parents, the program creates a bit-string for their child.[3] The program chooses an arbitrary index that falls in the new array. The child's array before the index is filled by copying the corresponding bits from the first parent. The child's array at & after the index is filled by copying the corresponding bits from the second parent.

Notice that the mating process has nothing to do with the constraints placed on valid organisms. It's just a single-point bit-string crossover. It's about as generic as a mating process gets in simulated evolution.

I chose single-point crossover as the method of mating because it is simple & generic. It's known to work well, or at least adequately, for genetic algorithms in general.

8 Mutation

Each new child has an opportunity to be a mutant. I set that opportunity at 1 in 1,000, but the actual number probably doesn't matter very much. After the child is created by the mating process (Section 7), the program rolls the classic pseudorandom die. If the child comes up a mutant, the program selects a random bit within the child's bit-string & assigns a new, randomly chosen bit value (0 or 1) to that location.

So 1 in 1,000 children are mutants, but some mutations are null-mutations because the new value assigned to the mutated bit is the same as the old value. In other words, 1 child in 1,000 contains zero or one mutated bits.

After the mutation process (if it occurs at all), the child's bit-string is complete & is ready to be converted to an organism so the program can find its fitness.

9 Output

Of course I made many runs of the program while developing it, but I declared two runs "official" before beginning them. They happen to be run number 7 & run number 8. Their results are described here.

arbitrary. In retrospect, it would have been easier & probably just as effective to use a constant, such as 10 or 17.

[3]Remember that the length of the bit-strings is fixed & known, to creating space for the bit-string for a new organism is simply a matter of creating a fixed-length array of bits.

run	best sequence of generation 105
7	(499871 494198 451488 128823 35957 13353 5467 2673 1097 340 171 58 24 9 4 1)
8	(498201 461299 275360 62025 18168 8186 3716 1325 444 177 61 23 13 4 1)

Figure 2: The best sequences from the 105^{th} generations of run number 7 & run number 8.

```
(263681 146305 66305 36289 16769 8929 4289 2161 1121 505 305 109
89 29 19 11 5 1)
```

Figure 3: Sedgewick's sequence of increments for arrays up to length 500,000.

For both runs, I allowed the program to begin with an initial population of randomized bit-strings. I allowed it to run for 105 generations.[4] For each run, I declared the best organism (sequence of increments) of the final generation as the output of the run. I did not scan the best-of-generation for previous generations, so it is possible that an earlier generation in the run found a better solution. So the two output sequences presented here are for the best-of-generation for generation 105 from run number 7 & run number 8.

Also for both runs, the maximum length of the test arrays was 500,000 elements.

The best sequences produced by run number 7 & run number 8 are in Figure 2.[5]

10 Results

How do the sequences from runs 7 & 8 compare to sequences already used for Shellsort?

To my knowledge, the best previously known sequence of increments for Shellsort was proposed by Sedgewick. It is the sequence whose elements are from $9(4^i) - 9(2^i) + 1$ or $4^i + 3(2^i) + 1$, including the element 1, sorted from largest to smallest, with duplicates removed[6]. Sedgewick's sequence for arrays of lengths up to 500,000 is in Figure 3.

There is also a sequence produced using evolution & reported in [2]. That sequence is in Figure 4.

[4] Actually, the program crashed in the midst of generation 106: memory leak. Bugs happen.

[5] For what it's worth, run number 7 began on 6 July & ended on 6 August 2002. Run number 8 began 23 August & ended 22 September 2002.

[6] Curiously, I've never seen mention that 1 should be inserted, that the sequence should be sorted, & that duplicates should be removed. Maybe it's obvious to everyone else, but I think it's the kind of detail that's worth mentioning.

```
(91433 72985 13229 5267 2585 877 155 149 131 23 8 1)
```

Figure 4: The sequence evolved by Bennett, Hannon, & Zehner

sequence	abs. work	rel. work	abs. seconds	rel. seconds
run 7	137094361	0.921	154.028	0.952
run 8	137337049	0.923	155.390	0.961
Sedgewick	148802578	1.000	161.755	1.000
Bennet et al	151668522	1.019	175.166	1.083

Figure 5: Comparison between sequences for Shellsort.

So how do the sequences compare? Figure 5 compares them, using Sedgewick's sequence as a basis.

The comparison which produced the table used 100 arrays of random integers. One of the arrays was of length 500,000. The other arrays had randomly chosen length. The contents of each array were randomly chosen integers. Each sequence sorted all 100 arrays, & the program counted the number of comparisons & copies as well as the actual amount of time required to sort all 100 arrays.

The first column in Figure 5 is the name of the sequence. The rows are sorted by the fifth column, relative seconds.

The second column, "absolute work", is a count of the number of comparisons & data copies. The third column is work relative to that done by Sedgewick's sequence.

The fourth column is the actual number of seconds required for the sequence to sort all 100 test arrays. The fifth column is the number of second relative to those required by Sedgewick's sequence.

Both sequences produced by the evolver described in this article perform better than Sedgewick's sequence. This is true whether performance is measured by counting comparisons & data copies or by tracking actual time required for the tests. When tracking actual time, the sequences from our evolver are nearly 5 percent better than Sedgewick's. When counting comparisons & copies, out sequences are nearly 8 percent better.

Also, Dr. Sedgewick provides a performance measuring program in [4]. According to a modified version of that program, driver0.c, the sequences from run 7 & run 8 perform well, though not as well as in the performance test I mentioned previously. Figure 6 shows the output of one run of `driver0.c`.

11 Conclusion

Two independent runs of the evolver, each using a population size of 10,000 & for 105 generations, produced sequences of increments which rival the best previously known sequences for Shellsort. Arguably, run number 7 produced a better sequence than the best previously known.

A population size of 10,000 isn't large, & 105 generations isn't a lot; the populations had not lost diversity by that generation. To me, this suggests that it would be worth experimenting with a larger population size & allowing

N	O	K	G	S	P	I	7	8	B
12500	0.140	0.060	0.060	0.060	0.060	0.060	0.060	0.060	0.070
25000	0.340	0.140	0.130	0.130	0.160	0.140	0.130	0.130	0.150
50000	1.290	0.350	0.340	0.300	0.360	0.310	0.290	0.310	0.340
100000	3.420	0.870	0.770	0.740	0.940	0.780	0.730	0.750	0.840
200000	7.870	2.030	1.760	1.630	2.160	1.740	1.640	1.670	1.880
400000	n/a	4.830	3.880	3.670	4.910	3.840	3.600	3.650	4.110
800000	n/a	10.760	8.450	8.050	10.810	8.360	8.030	8.110	8.960

key letter	algorithm
O	Shell's original
K	Knuth
G	Gonnet
S	Sedgewick
P	Pratt
I	Incerpi-Sedgewick
7	Stover 7
8	Stover 8
B	Bennett et al

Figure 6: The output of one run of `driver0.c`. The numbers in the inner elements of the table are times, in seconds.

the evolver to run for more generations. Maybe a population size of 50,000 or 100,000 for a few hundred generations would be worthwhile.

A The Source Code

The source code is available for download at http://cybertiggyr.com/gene/shiva-0/shiva.tar.bz2. On a unix system, you could fetch & extract it like this: "`wget -O- http://cybertiggyr.com/gene/shiva-0/shiva.tar.bz2 |bzcat |tar xf -`". Then `cd shiva`.

Here are some thoughts about it, made years after I wrote it. (I wrote it in 2002; I'm making these notes in 2006.)

- It uses a mix of Lisp & C.
- The main/driver loop, which does the evolution work, is in Lisp.
- The sorting is done in the C programs.
- The parts communicate through Sun/ONC RPC. And files.
- If I remember correctly, configuring the parts requires knowledge of your network. It's not automatic. It is possible that the code itself assumes your network is my network. (Even my network in 2006 isn't my network from 2002.)
- I did all of that for performance. I figured it would be faster if the sorting were done in parallel & by C.

  In retrospect, things would have been much simpler if I had written the entire system in Lisp with no parallel processing. I wouldn't be surprised if the existing program is four times the size it could have been if I hadn't used multiple processes.
- There is a memory leak. (Yep, you can have memory leaks even when you have garbage collection.) After 105 generations, it filled all of real memory & stopped. That's why I only ran for 105 generations.

  I haven't used the code in years, so I don't remember for sure, but it's possible that the memory leak is from memoizing Lisp's side of the fitness function. If so, the memory leak could be fixed by clearing the memoization table between generations.
- The build system is more complicated than I use now. I think it creates a `Makefile.in` by scanning the source code, then generates `Makefile` from `Makefile.in`.[7]

[7] At least I can say that, as my skills have improved, my solutions & systems have become simpler.

A Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/shiva-0/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/shiva-0/shiva-0.pdf.

References

[1] Sara Baase. *Computer Algorithms*. Addison-Wesley, second edition, 1991. ISBN 0-201-06035-3.

[2] Tiffany Bennett, Jennifer Hannon, and Elizabeth Zehner. Crew fall 2001 report. http://cs.allegheny.edu/ rroos/crew/fall2001.html, 2001.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, October 1989.

[4] Robert Sedgewick. Analysis of shellsort and related algorithms. *web site*, September 1996. http://www.cs.princeton.edu/ rs/shell/.

[5] Gene Michael Stover. Improving shellsort through evolution. *personal web site*, November 2002. `http://CyberTiggyr.COM/gene/shiva-0/`.

[6] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley Publishing Company, 2725 Sand Hill Road; Menlo Park, CA 94025, second edition, 1997.