SORTING THE SORTABLE FROM THE UNSORTABLE

Tracey Baldwin McGrail, Robert W. McGrail

Department of Mathematics, Marist College
Poughkeepsie, NY 12601
Tracey.McGrail@Marist.edu
Department of Computer Science, Bard College
Annandale-on-Hudson, NY 12504
mcgrail@bard.edu

Abstract

This article describes a discovery-based introduction to elementary genetic algorithms for students of introductory computer science via a series of programming laboratory exercises. The exercises focus on sorting weighted scores, a problem that is both accessible to the novice programmer and seemingly feasible by means of standard sorting methods. Students soon discover that standard, deterministic techniques prove insufficient and so must settle for approximation by genetic algorithm. Experimentation with this approach reveals the folly of sorting weighted scores.

Introduction

This paper describes a series of three introductory-level laboratory exercises designed to help introduce students to basic genetic algorithms. Students consider the problem of arranging a finite sequence of ordered pairs, called **weighted scores**, in descending order. The problem appears to be close enough to the sorting of numbers to admit a solution using a variation on one of the standard sorting algorithms. Through the usual program-test-debug process students soon discover that most of the known sorts are fundamentally inappropriate for the task. Furthermore some of the students are able to verify that none of the popular algorithms fit the bill. They are then led to consider the use of a genetic algorithm to approximate the correct order. During the program-test-debug process they prove that not all sequences of weighted scores can be sorted.

The likely pedagogical benefits of these exercises are many. For starters, they collectively constitute yet another directed course project, which is a fairly popular learning device in computer science courses. In fact, this has the potential for a very effective project, since the problem of weighted sorting is seemingly simple yet deceptively complex. During failed attempts to find a solution the serious student is likely to remain confident that a deterministic algorithm is well within her grasp, yet is unlikely to become bored by the process.

The exercises then motivate genetic algorithms, a popular area of contemporary computing research, at a most opportune moment. Students are given the choice to either press on for a deterministic algorithm or resort to approximation. Meanwhile, the notion of genetic algorithm is added to their toolbox during classroom lecture. Those who choose this path soon unravel the impossibility of sorting weighted scores.

Intellectual journeys of this particular type are important to proper pedagogy within the mathematical sciences. Educational proponents of every field within this general domain, such as computer science, mathematics, and physics, have long trumpeted the virtues of abstraction within the curriculum. Understanding through universal reasoning is unlikely when the number of concrete instances is severely limited. Science education is then hardly distinguishable from training in more vocational programs. For instance, computer science students often learn about stock topics, such as the sorting of numbers, without considering alternative scenarios, such as the sorting of weighted scores.

From this point of view, this study of weighted scores draws strong analogy to the learning of foreign language. For example, systematic study of French helps the native English speaker better understand the grammatical structure of her own tongue. Similarly, attempts to properly order weighted scores will help one develop a firm grasp of the sorting of ordinary numbers.

However, the main motivation for this particular series of exercises is to inspire student-centered research at the earliest possible stage. Academic computer science is especially equipped for such work. Computer science students need not wait for several semesters of background material before embarking upon exploratory research. This stands in stark contrast to the current state of most other mathematical disciplines. Also, computer science

instructors have much more freedom to dedicate sufficient time for deep exploration into individual topics than their counterparts in lab science disciplines, such as chemistry and biology, who are constrained by a standardized curriculum that requires the introductory course to serve as part of a broad survey of the field. Hence teachers of first-year computer science students have greater opportunity to acclimate their students to the culture of research through the assignment of research-grade project topics.

Summary

This next section proceeds with background definitions, notation, and a summary of previous results on weighted scores. This is followed by a section describing a simple genetic algorithm that approximates an "optimal order." Subsequently, the three lab exercises are presented in proper chronological order. The concluding section includes some observations germane to the overall effectiveness of the series.

Weighted Scores

This section presents the mathematics of the issue central to this paper. A related problem is the subject of (Eppstein and Hirschberg 1997) and so the material below strongly reflects the introductory section of that article. This section constitutes a summary of (McGrail and McGrail 2004). The reader should consult that paper for a complete treatment of these ideas. In fact, much of the language below is presented verbatim from (McGrail and McGrail 2004).

A certain course's collection of assignments consists of **n** equally weighted exercises. The grading policy for this course allows each student to play some game of chance that would allow her to generate some arbitrary nonnegative integer **k**. She then keeps the best **k** of her **n** grades, thereby dropping the lowest **n - k** grades.

For example, assume that there were a total of ten equally-weighted assignments during the course of the semester (**n = 10**). Moreover, suppose that Sue rolled a single die which produced **k = 6**, and that her grades were the following.

87, 78, 100, 95, 65, 80, 82, 92, 95, and 88

Her lowest four grades are dropped leaving her with

87, 100, 95, 92, 95, and 88

which correspond to a final grade of approximately **92.8**.

Clearly the simplest way for the professor to assign final grades is to use any standard sorting algorithm to sort the list of grades in descending order and then keep only the first **k** grades.

A wrinkle appears in the system when the **n** grades are of non-uniform weight. In this scenario each grade is composed of two values, one depicting the total number of possible points one can earn on that assignment and the other signifying the number of points awarded to the student on that assignment. So a score is an ordered pair **(s,w)** where **w** is a positive integer and **s** is a nonnegative integer no greater than **w**. For instance, **(20,20)** and **(0,20)** represent the best and worst possible scores of weight **20**, respectively. Consequently, the collection of grades for a particular student is a list of **n** scores. In order to avoid confusion with the ordered pair syntax, a list of scores is presented using square brackets and semicolons in the format below.

$$\mathbf{L=[(s_1,w_1);(s_2,w_2); \ldots;(s_n,w_n)]}$$

Furthermore, define the average of this list to be the sum of all of the first coordinates divided by the sum of all of the second coordinates, or

$$\mathbf{avg(L) =(s_1+s_2+\ldots+s_n)/(w_1+w_2+\ldots+w_n)}.$$

For example,

$$\mathbf{avg([(55,60);(0,10);(1,10);(100,100)]) = 156/180}$$

which rounds off to **86.7** percent.

The problem of sorting weighted scores can be stated as follows:

> Given a list **L** of **n** scores, reorder **L** to a new list **L'** such that, for each integer **k** strictly between **0** and **n**, the first **k** scores in **L'** have average as large as any other collection of **k** elements of **L**.

For example, consider this list.

[(55,60);(0,10);(1,10);(100,100)]

The reader may verify that the properly ordered version follows below.

[(100,100);(55,60);(1,10);(0,10)]

Based on this example, the naïve approach is to sort the scores in order of decreasing proportion using one of the standard sorting algorithms, such as BUBBLESORT, INSERTIONSORT, MERGESORT, QUICKSORT, or SELECTIONSORT (Aho, et al. 1974, Cormen, et al. 2001). However, it is shown in (McGrail and McGrail 2004) that each of these algorithms fails to do so properly. For example, any sort that uses a "divide-and-conquer" approach relies on the notion that the relative order of any two elements in a subcollection must agree with their relative order in a sorted version of the list. This is not the case with scores. To illustrate this point, it is easy to verify that the following two lists of scores

[(9,10);(80,100);(5,10)]

and

[(100,100);(5,10);(80,100)]

are sorted. Consequently, the relative order of the scores **(80,100)** and **(5,10)** in a particular list depends on the other elements in the list.

An algorithm loosely based on SELECTIONSORT is presented in (McGrail and McGrail 2004) that attempts to

sort lists of scores. Testing revealed that the randomly generated list of scores

[(2,4);(2,2);(4,7);(6,10)]

was reordered by this algorithm to

[(2,2);(2,4);(6,10);(4,7)].

However, the proper order is

[(2,2);(4,7);(6,10);(2,4)]!

Another randomly generated list of scores

[(2,4);(6,10);(3,3);(4,7)]

encountered during testing has no correct order. For a correct order to exist, the best two scores, which are

(3,3) and (2,4),

must be contained in the best three scores, which are

(3,3), (6,10), and (4,7).

This means that not all lists of scores are sortable, so that no sorting algorithm is possible!

These results naturally lead to the following series of questions: Does there exist a reasonably quick algorithm that sorts all sortable lists? Furthermore, is there a notion of "best" order for unsortable lists that coincides with sorted order for sortable lists? If so, how does one efficiently realize this order? This set of questions is answered in part in the next section.

A Genetic Approach to Best Order

In this section the notion of a best order for a list of weighted grades is proposed. This best order is achieved by means of maximizing a certain fitness measure applied to orderings of a given list. In the case of a sortable list, this fitness measure achieves a maximum for the sorted order. This naturally leads to a genetic algorithm for approximating best order, which then also estimates sorting order for sortable lists.

To that end, consider the list of scores

$$\mathbf{L=[(s_1,w_1);(s_2,w_2); \ldots;(s_n,w_n)].}$$

Define the truncated list L_k to be

$$\mathbf{L_k =[(s_1,w_1);(s_2,w_2); \ldots;(s_k,w_k)].}$$

We let the fitness of **L** be the sum of the averages of the $\mathbf{L_k}$ for **1<=k<n** or

$$\mathbf{Fitness(L)=avg(L_1)+avg(L_2)+\ldots+avg(L_{n-1}).}$$

For example, consider the list of scores

L=[(2,4);(6,10);(3,3);(4,7)]

which was shown to be unsortable in the previous section. It is easy to check that the following reordering of L provides the maximum fitness score

L'=[(3,3);(2,4);(6,10);(4,7)]

with **Fitness(L')** rounding off to **2.36134**.

Notice that, by the definition of sorted order of the previous section, if **L** is already sorted, each quantity $\mathbf{avg(L_i)}$ is maximum over all subcollections of **L** of size **i**. Hence for a sortable list, the sorted order achieves maximum fitness.

The original problem can be safely rephrased in terms of finding a best order as follows:

> Given a list **L** of **n** scores, reorder **L** to a new list **L'** so that the sum over **k** of $\mathbf{avg(L'_k)}$ achieves a maximum.

Now one need only apply an elementary template to construct an approximation via genetic algorithm (Mitchell 1996). The general algorithm is described below. Here assume that the input list **L** is represented by a linear array. One proceeds as follows:

1. Let **A** be an array of **10** lists of scores. Initially **A** simply holds **10** copies of **L**.
2. For **0<=i<5**, replace **A[i+5]** with a new list generated by randomly switching two of the scores in **A[i]**.
3. Compute the fitness of each list **A[i]**.
4. Reorder the array **A** in decreasing order of fitness.
5. Repeat steps **2**, **3** and **4** for some specified number of iterations.
6. The output list is **A[1]**.

Some Observations

More can be discovered about the problem of sorting weighted scores by studying the fitness landscape (Mitchell 1996). For a list **L** of **n** scores, one considers the collection of orderings of **L**. The distance between any two orderings of **L** could be defined as the number of positions in which they differ. So **L** and **L'** are "close" if **L'** can be realized from **L** by switching two scores. The fitness landscape can be viewed as an **(n+1)**-dimensional graph in which each ordering of **L** is a point in **n** dimensions and the associated fitness is plotted along the **(n+1)**st axis (Mitchell 1996).

From this point of view, an optimal order would occur where there is a peak in the landscape. Once this peak in fitness is achieved, any change in the order will cause the fitness to decrease, thus stranding **A[1]** in this position. Experimentation with the previous GA has yet to produce a peak that is not a global maximum. In other words, all runs of the algorithm have found a maximal order. On average, the GA produces correct results after approximately **50** generations for an input list of length **10**.

The Laboratories

The following series of programming laboratories guides the student through the concepts mentioned previously. The first laboratory introduces the student to the general notion of sorting numbers. However, it insists that students invent their own algorithm. This is intended to place the student into a discovery-based mindset. In the

second laboratory, the student explores the applicability of the standard sorting algorithms to the problem. Finally, in the third, the student is asked to provide a general solution to the problem. The presentation is abbreviated to avoid repetition of the first three sections of this article.

Laboratory 1: Sorting Blocks

This laboratory introduces the student to the general notion of sorting numbers. It asks the student to design an algorithm that sorts a collection of numbered blocks into descending order. The caveat is that none of the standard sorting algorithms such as BUBBLESORT, INSERTIONSORT, MERGESORT, QUICKSORT, or SELECTIONSORT (or even slight variations of these) are considered acceptable. The written report should include

- A description of the algorithm;
- Examples of the algorithm in action illustrated via box-and-pointer diagrams; and
- An implementation of the sort.

Laboratory 2: The Standard Approach

In this laboratory, the student considers the applicability of the standard sorting algorithms to the problem of sorting weighted grades. The student is given a brief description of the problem that does not include the results discussed in this article. She is asked to determine whether any of BUBBLESORT, INSERTIONSORT, MERGESORT, QUICKSORT, or SELECTIONSORT solve the problem of sorting weighted grades. The written report should include the following for each of the aforementioned methods:

- An explanation of why a direct application of the algorithm works if she believes so;
- If she believes that a modification of the sort will work she must provide an implementation along with an argument supporting its correctness;
- Specific examples of lists that should foil any version of the algorithm; or
- Some intelligent comments about why the appropriateness of an algorithm is not as clear as some of the others if the student is unsure about the conclusion.

Laboratory 3: A New Beginning

At this juncture, it is assumed that the student is already convinced that most standard sorting algorithms do not apply to the problem of sorting weighted scores. In addition, genetic algorithms have been introduced to the student in the classroom as a means for sorting ordinary numbers.

In this exercise, the student is asked to use their full repertoire of algorithmic techniques to provide the best available insight to the problem. The written report should include one of the following:

- An algorithm that successfully sorts all sequences of weighted scores very quickly;
- An algorithm that successfully sorts all sequences of weighted scores, but does not do so very quickly; or
- A reasonably paced algorithm that sorts most lists of weighted scores and nearly sorts all of the rest.

The student's claims must be supported by testing randomly-generated examples. Moreover, if the student fails to achieve an algorithm satisfying either of the first two categories, she should provide some insight as to why she could not do so.

Epilogue

This series of exercises was first employed in a section of Computer Science I at Bard College during the spring semester of 2005. Below are some observations pertaining to the effectiveness of each laboratory exercise as well as the overall impact of the project on that course.

Exercise 1 went much better than expected. It was feared that many students would try to locate a solution on the Internet and submit a modified version as their own. The strange diversity of algorithms submitted suggested otherwise.

Of the three, Laboratory 2 appeared to have the most profound effect on the class. Most were able to deduce the irrelevance of local comparisons between scores, and so correctly concluded that INSERTIONSORT, MERGESORT, and QUICKSORT are fundamentally inappropriate for this application as shown in (McGrail and McGrail 2004). Many also eliminated one of BUBBLESORT or SELECTIONSORT from contention, but usually for the wrong reason. All of the students were convinced that at least one of BUBBLESORT and SELECTIONSORT would inspire a working solution. In fact, a handful of students each submitted a sizable, untested candidate program accompanied by a bold, but insufficiently supported, claim of correctness. However, there was universal acceptance that the sorting of weighted scores requires a fairly complex methodology.

In (Eppstein and Hirschberg 1997), it is revealed that no greedy algorithm exists for finding the best **k** scores in a collection. This eliminates any version of SELECTIONSORT from contention. On the other hand, the observed efficiency of the GA strongly suggests that some version of BUBBLESORT might work. However, a precise formulation of such is unknown to the authors at the time of the writing of this paper.

On the other hand, there are versions of SELECTIONSORT that satisfy the last option of Laboratory 3. The reader is referred to (McGrail and McGrail 2004) for such an example. A sizeable minority of the students took this path.

Most of their implementations were slow and buggy, and so did not facilitate extensive testing. Just one of these students was able to unearth an unsortable list and so prove the unsortability of lists of weighted scores.

All of the proponents of BUBBLESORT from Laboratory 2 chose to implement a GA for the third stage. Many of these students were willing to settle for an approximation algorithm because the general GA approach reminded them of the general BUBBLESORT method. Others were simply eager to implement their own GA.

The GA implementers enjoyed relatively remarkable success. Each of them was able to develop the proper fitness measure and a correctly working program over a relatively short period of time. Moreover, the GA implementation helped them to better understand the use of random number generators in programs. This left them well equipped to develop effective testing routines. All of them discovered counterexamples which led them to the correct conclusions.

Variations on a Theme

One can argue that the fitness measure from the GA section is not correct since it gives disproportional weight to the arrangement of the early part of a list. For instance, suppose that the professor in question always assigns fifteen graded exercises per semester. Also assume that the game of chance consists of simply rolling one die to get result **i** and then letting **k** be **n - i**. Then **k** ranges from **9** to **14**. In this case, the arrangement of the first nine scores should not matter to the fitness measure.

A solution for this general situation proceeds as follows. For each **1<=k<n**, let **P(k)** be the probability that exactly **k** scores remain after the game of chance. Replace each term of the form $\mathbf{avg(L_i)}$ in the fitness measure with the new term $\mathbf{avg(L_i)P(i)}$, yielding the new fitness measure

$$\mathbf{avg(L_1)P(1)+avg(L_2)P(2)+\ldots+avg(L_{n-1})P(n-1).}$$

Then the reader can verify that sorted lists still achieve maximum fitness. Moreover, for the case above, $\mathbf{avg(L_i)P(i) = 0}$ for **i < 9**, so the arrangement of the first nine scores has no bearing on the fitness.

A Note Concerning the Benefits of Anonymity. Many of the negative consequences of the internet age to the cause of education are likely familiar to the reader. Rarely do interesting problems and exercises proliferate long before they are followed by easily downloadable solutions or before they become the topic of conversation on public forums accessible to students. In the interest of keeping ready-made solutions out of the reach of novices, it is recommended that adopters of these exercises use alternative nomenclature for terms such as "score" and also vary the collection of sorting algorithms employed in Laboratory 2.

References

Aho, A, V., Hopcroft, J.E., and Ullman, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Boston, MA: Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms, 2nd Edition*. Cambridge, MA: MIT Press.

Eppstein, D. and Hirschberg, D. 1992. Choosing Subsets with Maximum Weighted Average. *The Journal of Algorithms*, 24(1): 177-193.

McGrail, R. W. and McGrail, T. B. 2004. A Grading Dilemma or the Abyss between Sorting and the Knapsack Problem. *The Journal for Computing in Small Colleges*, 19(5): 97-107.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press.