Teaching Algorithms by Tweaking Algorithms

Olga Lepsky[1] [Michael Werner[2]]

Abstract This paper describes an approach to teaching algorithms that requires students to do experimentation and think about ways to improve/apply the algorithms. They program and benchmark the algorithms on inputs of increasing size and compare the results to the predicted complexity curve. They then attempt variations (tweaks) on the algorithm to see if they can improve performance, or apply it to a different input. The approach was tested in two computer science courses: Algorithm Design and Analysis, and Bioinformatics Algorithms.

Keywords: Algorithms, Bioinformatics

1. INTRODUCTION

At our institution the majority of the undergraduate computer science majors go on to careers as software engineers with a few continuing directly into graduate education. We have developed a hands-on approach to teaching algorithms that emphasizes programming, benchmarking and tweaking. This report builds on our experience over a three-year period involving two different courses: "Analysis of Algorithms" and "Bioinformatics Algorithms". We follow standard texts [10, 13], the students frame the problem, describe the solution and express an appropriate algorithm in pseudocode. They derive a timing function and the Big-O behavior of the algorithm.

Students then write programs in C++, C# or Java to implement the algorithms. After thorough testing to verify correctness, the programs are benchmarked for running time. They run a series of trials on inputs of increasing size, graph the results and compare the actual timing curve to the one predicted by a theoretical analysis.

The novelty of our approach is that students are encouraged to go beyond standard algorithms by coming up with variations on them. The novel or tweaked algorithms are analyzed, implemented, tested for correctness, and benchmarked. The search for new approaches goes beyond the textbook into the research literature.

Still, a theoretical computer scientist may wonder if we are giving the wrong message to students by suggesting they can program their way out of an intractable algorithm. We refute this argument:
1) We still do the analysis. The students are led through a standard derivation resulting in either a summation or recurrence which is then expressed in closed form.
2) Although analysis predicts the shape of the timing curve there is still room for improvement by changing the constants involved. So tweaking the algorithm may be able to lower the curve even if it can't change its shape.
3) There are cases where lower bounds research indicates that the best algorithm may not have been found yet, for example, matrix multiplication [2, 3, 15].
4) Average case analysis is hard to do on an undergraduate level but is of great interest to the practitioner. Our students run repeated trials of several algorithms on randomized inputs to help judge which algorithm shows best results on practice.
5) Memory space requirements are hardly mentioned in algorithms texts, yet we found that problems in bioinformatics were often constrained by lack of memory.

---

[1] Wentworth Institute of Technology, 550 Huntington Ave, Boston, MA 02115, lepskyo@wit.edu

[2] Wentworth Institute of Technology, 550 Huntington Ave, Boston, MA 02115, wernerm@wit.edu

6) Tweaking the problem may work even when there is no room to tweak the algorithm. For example, a company which employs traveling salesmen may be satisfied with a "pretty good" routing algorithm when an optimum one is infeasible.
7) Historically, pioneers in algorithms research have not been afraid to experiment and tinker, i.e. Knuth's classic text "The Art of Computer Programming" [12] has implementations of algorithms written in assembly language. R. Sedgewick accompanies all of his algorithms textbooks [16] with programming examples.

8) Most of the students in our school are preparing to be software engineers. They benefit from practice in choosing, implementing, and tweaking the algorithms.

The next sections show examples of tweaking algorithms.

2. TWEAKING IN THE ALGORITHM DESIGN AND ANALYSIS COURSE

2.1 Interpolation search in a list

The students are presented with the formula for (linear) interpolation search [13] and asked a series of questions related to the applicability of the algorithm. For what arrays does interpolation search work? (Answer: for sorted arrays of numeric types.) For what arrays will interpolation search work faster than binary search? How can we tweak interpolation search if we want to apply it to non-numeric elements (characters, strings, names, dates)? Here the students need to think of ways to encode/decode these elements as numbers. How can we tweak interpolation search if it is known that elements are distributed close to a nonlinear model (logarithmic, quadratic, cubic, exponential). The possible answer is to find the approximation by changing a formula for linear interpolation to the formula for interpolation based on a particular nonlinear function.

2.2 Huffman's algorithm for variable-length text encoding

The problem is to devise a binary encoding for a message which minimizes the required number of bits [9]. Starting with a frequency distribution table for the letters in the chosen alphabet a forest of single-node trees is formed with the weight (frequency) stored in the root. On every step of this algorithm, one chooses the trees with the smallest weight and joins them as children of a new tree, labeling the left/right edges with 0/1. The binary encoding of each letter is found by concatenating the edge labels of the unique path from the root to the leaf representing the letter. However, without further constraining the problem the tree produced is not guaranteed to be unique. It is worthwhile to have students experiment with different rules to ensure uniqueness of the Huffman encoding: how to break a tie when deciding which pairs of nodes to combine next, which node in the pair becomes left and which - right child of a new tree.

2.3 Stable Marriage Algorithm

This algorithm was introduced in 1962 by Gale and Shapley [6]. In-depth discussion of the stable marriage problem and its extensions can be found in the monograph by Gustfield and Irving [8]. The students find the problem amusing and applicable and the algorithm easy to understand and remember. The stable marriage algorithm has a shortcoming: it is biased - it favors the preferences of the group whose members make the proposals over the preferences of the group whose members choose whether to accept or refuse. The students are asked how we can tweak the algorithm to reverse the bias, and, better yet, to eliminate it. (We got some good suggestions from the students.) The next question arises: if it is proven that the original algorithm always results in a stable matching, does the modified algorithm also achieve it? The last question may be answered by doing analysis or by extensive testing.

The stable marriage algorithm was applied to college admissions, matching two roommates, and matching medical school graduates to residency training programs [6, 8]. An interesting project is to apply (possibly tweaking) the stable marriage algorithm to other useful matches: the students and co-ops, groups of several roommates, etc.

2.4 Numerical algorithms for solving nonlinear equations

We cover three numerical algorithms for solving nonlinear equations: bisection, false position, and Newton's [13]. These methods can be applied to solve the equation $y=f(x)$ on an interval $[a, b]$, where $f(x)$ is a continuous function, but with the following limitations: bisection and false position method require $f(a)$ and $f(b)$ to have opposite signs, Newton's method requires f to be smooth and the derivative of f to exist and be nonzero at approximations and at the root. If we apply the above methods to functions and intervals where the limitations do not hold, the methods may diverge (see online Newton method applet [7]). The students are asked to implement these methods, check what happens when limitations do not hold, and tweak the above methods by suggesting detours going around these limitations. For example, if using Newton's method the derivative at a root approximation x_n, becomes too small or too big, we can switch to the secant method [5] (approximating derivative with the quotient) between the two previous approximations x_{n-1} and x_n.

2.5 Hashing

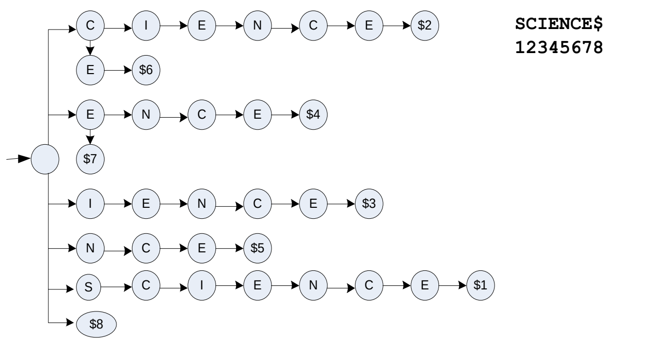
Hashing is a technique that has proven useful since the 1950s [12]. Students readily grasp the concept and using open hashing (each hash bucket implemented as a list), they are able to program the insert and search functions. We have them benchmark their programs on substantial texts, hashing each word. Once students have ascertained correctness of their algorithm they work on improving the timing. They may adjust the hash function to achieve a better scattering, vary the number of buckets, and change the container used for the buckets. We looked at various hash functions for storing student records based on their first and last names or their dates of birth. We compared sizes of the hash tables, their load factors, number of collisions, time efficiency of search in the average and worst case scenarios, and selected the best hash function. Hashing can be used for other problems like encoding passwords and card games. These algorithms were researched, presented, and implemented by a group of students.

3. TWEAKING IN THE BIOINFORMATICS COURSE

3.1 Suffix trees

Suffix trees are used in bioinformatics for full-text indexing (see Figure 1 below). To start, students construct trees with each edge labeled by a single character. They benchmark building trees from texts of increasing size. They quickly discover that the brute force suffix tree is memory bound.

Figure 1. A suffix tree for the text string "SCIENCE$". Each path from the root to a leaf traces out a suffix whose starting point is stored in the leaf. To search for a substring start at the root and try to trace it out staying within the tree, i.e. the substring "C" occurs at positions 2 and 6 in the text.



The next step is compaction. Strings of edges with no branching are compacted to a single edge labeled by a substring of the text rather than just one character (see Figure 2a). This significantly reduces memory need. However, the labels themselves consume considerable memory. The next well-known improvement is to replace the labels by two indices; the start and stop of a section of text matching the substring (see Figure 2b).

Figure 2a. A compacted suffix tree.

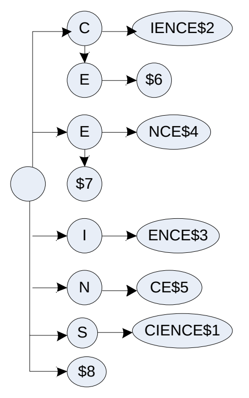
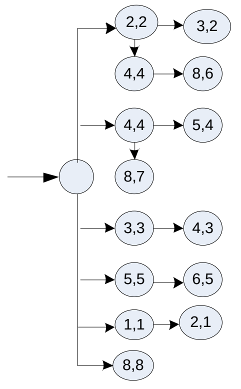
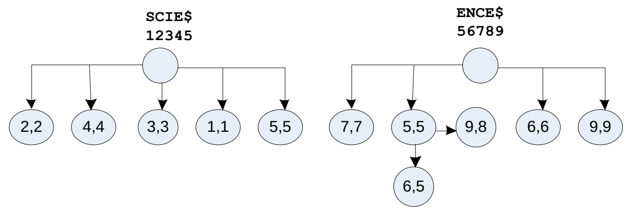


Figure 2b. The labels have been replaced by indices of the start and end of the label in the original text. For leaves the second number indicates the starting point of the suffix rather than the index of the $ sign.



Finally students came up with the idea of constraining the length of the patterns to be matched. It enabled the construction of a forest of much smaller trees by partitioning the text and including some overlap between the partitions to accommodate patterns that began in one partition and ended in the next one (see Figure 3). This produced a very significant improvement and also led to a way of parallelizing the problem. Subforests were built and searched on different processors. This was tested on a DNA text string 60,000 base pairs long. An overlap of 15 allowed efficient search for 1-mers of length 16. The program was written in threaded Java and tested on a Sun server with 16 processors. Forests consisting of 1 to 16 trees were tried, each built and searched in its own thread. Significant reductions in the memory space requirements and execution times were seen as the number of trees increased. Although suffix tree forests are not a novel solution [1] the parallel implementation may be.

Figure 3. A suffix forest with an overlap of one permitting efficient search for substrings of length 2 or less.



.

3.2 Hidden Markov Models

We introduce undergraduates to state machines in their programming languages course. It is a minor leap to make the transitions between states probabilistic in nature. Add to this the notion of an alphabet of observable emissions with the probability of emitting each letter dependent on the state. When the emission stream is observable and the state sequence is hidden you have a Hidden Markov Model [14].

Our objective is for students to achieve proficiency in working with HMMs. We start with the forward algorithm, i.e. find the probability of emitting a given string using the model parameters (fixed emission and transition probability matrices). Simulating a short sequence by hand or on a spreadsheet helps to understand the algorithm.

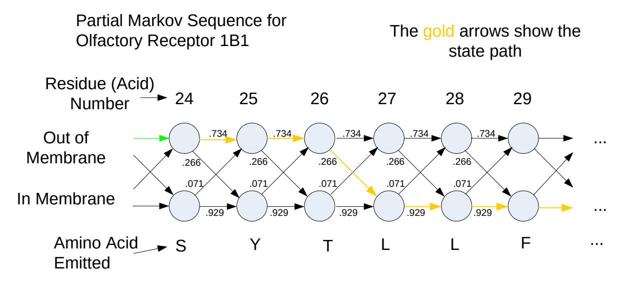


Figure 4. A Hidden Markov Model for predicting transmembrane regions in olfactory receptor proteins.

The Viterbi [14] algorithm is then introduced. The objective is to find the most probable state sequence to have produced the observed emission stream. We begin with a toy problem, "the fair bet casino" [10]; the idea is to predict when an unscrupulous casino switches back and forth between tossing a fair and a loaded coin. The students can experiment using their own code or published code such as by T. Kanungo [11]. Next, a real problem from biology is introduced, namely finding transmembrane regions in olfactory receptor proteins given the sequence of amino acids comprising the protein [4] (see Figure 4). Training the model gives an opportunity to do some tweaking. The students determined experimentally that using amino acid hydrophobicity properties gave better results than choosing random parameters followed by Baum Welch [18] training.

4. CONCLUSION

In the "Algorithms Design and Analysis" course, students tried modifying algorithms to achieve a better performance (example 2.5), to apply it to a slightly different or a more general problem (examples 2.1, 2.3, 2.4, 2.5), to improve the output (ensuring uniqueness for a particular input in example 2.2 and eliminating bias in example 2.3). While easier problems were given to all the students in the homework / laboratory assignments, the more challenging questions on modifying algorithms were given as extra credit bonuses and later discussed with the whole class. Exams showed that the students understood and better remembered the algorithms for which possible tweaks were discussed. Students told us in their evaluations that tweaking the algorithms made the class more challenging and deepened their understanding of algorithms.

The second course, "Bioinformatics Algorithms" is run as a kind of research seminar with the professor and students working together on programs used by researchers in biology. Existing programs often run slowly and require significant resources in terms of processing power and memory. Example 3.1 (suffix tree forests) achieved vastly better results by tweaking the problem to limit the size of search strings. Example 3.2 tweaked the HMM training. The design of these approaches was done as a group activity. The sessions were lively and involved drawing lots of diagrams on the white board. The problems were challenging and the students were fully engaged. Students polled at the end of the course agreed that the hands-on coding, tweaking and benchmarking approach had significantly advanced their problem solving abilities.

Looking forward, we hope to extend this teaching technique into some new areas including graphics programming, network protocols, and parallel processing. We also plan to conduct an empirical study statistically evaluating the effectiveness of this method for teaching algorithms.

REFERENCES

[1] Barsky S. and Thomo U., "A new method for indexing genomes using on-disk suffix trees," *CIKM,* 2008, 649-658.
[2] Bshouty N., "A Lower Bound for Matrix Multiplication," *SIAM J. Comput*, Vol. 18, 1988.
[3] Leiserson C. and Stein R., *Introduction to Algorithms*, MIT Press, 2001.
[4] Cristianini H., *Introduction to Computational Genomics: A Case Studies Approach,* Cambridge University Press, 2006.
[5] Dahlquist B., *Numerical Methods*, Dover Publications, 1974.
[6] Gale D. and Shapley L.S., "College admissions and the stability of marriage," *American Mathematical Monthly*, Vol. 69, 1962, 9-14.
[7] Garrett P., *Newton's method (applet)*, http://www.math.umn.edu/~garrett/qy/Newton.html.
[8] Gustfield D. and Irving R.W., "The Stable Marriage Problem: Structure and Algorithms," MIT Press, 1989.

[9] Huffman D.A., “A method for the construction of minimum redundancy codes,” *Proceedings of the IRE*, Vol. 40, 1952, 1098-1101.
[10] Jones N. and Pevzner P., *An Introduction to Bioinformatics Algorithms*, MIT Press, 2004.
[11] Kanungo T., *Hidden Markov Model Toolkit in Extended Finite State Models of Language*, Cambridge University Press, http://www.kanungo.com/software/software.html, 1999.
[12] Knuth D., *The Art of Computer Programming*, Addison-Wesley, 1973.
[13] Levitin A., *The Design and Analysis of Algorithms*, 2nd ed., Addison-Wesley, 2007.
[14] Rabiner L. R., “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition,” *Proceedings of the IEEE*, Vol. 77, NO.2, Feb. 1989.
[15] Raz R., “On the Complexity of Matrix Product,” *SIAM Journal of Computing* 32(5), 2003.
[16] Sedgewick, R., *Algorithms in C++*, Addison Wesley, 1998.
[17] Viterbi A.J., "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory* 13 (2), 1967, 260–269.
[18] Welch L. R., “Hidden Markov Models and the Baum–Welch Algorithm,” *IEEE Information Theory Society Newsletter*, Dec. 2003.

Olga Lepsky

Adjunct Professor of Computer Science at Wentworth Institute of Technology, she earned a Ph.D. in Mathematics from Brown University.

Michael Werner

Professor of Computer Science at Wentworth Institute of Technology, he earned a Ph.D. in Computer Science from Northeastern University.