Algorithms Research: Finding a Problem

Eliezer A. Albacea
Institute of Computer Science
University of the Philippines Los Baños
College, Laguna, PHILIPPINES
eaalbacea@uplb.edu.ph

ABSTRACT

In this paper, we survey the areas in algorithms that one can start a research on. It covers the areas on bounds on complexity like upper and lower bounds. Then, it deals on tackling problems involving special cases and problems on practical algorithms.

Keywords

Algorithms, bound of complexity, lower bound, upper bound, practical algorithms.

1.INTRODUCTION

The area of algorithms is a very wide area of research. In fact, several books have been written that covers all aspects of this area of research in Computer Science. One particular comprehensive book in this area is the Introduction to Algorithms by Cormen, Leiserson and Rivest (1990). Several journals are also dedicated to publishing exclusively for algorithms papers. Examples of journals exclusive to algorithms papers are Journal of Discrete Algorithms, Algoritmica, Journal of Algorithms and many more. Also being held every year are several conferences solely on the subject of algorithms.

In this paper, we shall identify aspects of algorithms where one can do research. Specifically, we shall concentrate only on design and analysis of algorithms. Of course, one can simply design and analyze a new algorithm in some areas of algorithms like graph algorithms, computational geometry, data structures, optimization algorithms and many other areas, but identifying where to start is usually a problem. Our concern in this paper is to identify where to start looking for a problem. In particular, we shall be looking at lower bounds, upper bounds, special cases and practical implementations. This paper will mostly report the author's experience in doing algorithms research.

2.BOUNDS ON COMPLEXITY

The primary definition of complexity is running time. For a given problem, time complexity is a function that maps problem size into the time required to solve the problem. Typically, we are interested in the (inherent) complexity of computing the solution to problems in a particular class of problems.

For example, we might want to know how fast we can hope to sort a list of n items, initially in an arbitrary order, regardless of the algorithm we use. In this case, we seek a lower bound, denoted by L(n), on sorting, which is a property of the sorting problem and not of any particular algorithm. This lower bound says that no algorithm can do the job in fewer than L(n) time units for arbitrary inputs, i.e., that is every sorting algorithm must take at least L(n) time in the worst case. Thus, the lower bound considers all the algorithms including those undiscovered ones.

On the other hand, we might also like to know how long it would take to sort such a list using a known sorting algorithm with a worst-case input. Here, we are after for an upper bound, denoted by U(n), which says that for arbitrary inputs we can always sort in time at most U(n). That is, in our current state of knowledge, we need not settle for an algorithm which takes more than U(n) time, because an algorithm which operates in that many steps is known. For this reason, algorithms are normally analyzed to determine their worst-case behavior, in the hope of reducing U(n) even further demonstrating that some new algorithm has worst-case performance which is better than any previous algorithm.

One way of seeing the distinction between lower and upper bounds is to note that both bounds are minima over the maximum complexity of inputs of size n. However, L(n) is the minimum, over all possible algorithms, of the maximum complexity. In trying to prove better lower bounds, we concentrate on techniques that will allow us to increase the precision with which the minimum, over all possible algorithms, can be bounded. Improving an upper bound means finding an algorithm with better worst-case performance. This difference leads to the differences in techniques developed in complexity analysis.

While there are apparently two complexity functions for problems, lower and upper bounds, the ultimate goal is to make these two goals coincide. When this is achieved, the optimal algorithm will have been discovered and we obtain L(n) = U(n). For some of the problems, this goal is not yet realized. Thus, the research area is to reduce the gap between L(n) and U(n) because normally L(n) is lower than U(n) for some problems.

2.1Lower Bounds

The more difficult of the bounds on problem complexity is the lower bound. There is no algorithm to analyze; few general principles to apply; proofs of result in this area often require

outright cleverness. The results must apply to any algorithm, including undiscovered ones.

For some problems, the lower bound is trivial. Consider the problem of finding the maximum element of an array of n keys. Any algorithm that solves this problem will need at least n-1 comparisons. To see this, we obviously need to compare every element with the maximum of elements previously considered. Hence, a lower bound of $\Omega(n)$.

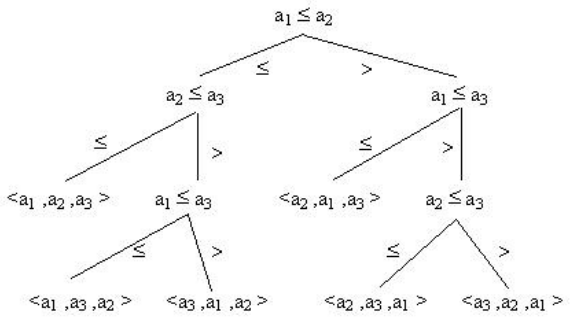
2.1.1 Information-Theoretic Lower Bound

However, for other problems the lower bound is not trivial. There is what we call the information-theoretic lower bound that can be used. We illustrate this method of identifying a lower bound using sorting, i.e., we solve the lower bound for sorting.

A sorting algorithm that sorts elements by comparing the items until a sorted order is obtained is called a comparison-based sorting algorithm. Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree represents the comparisons performed by a sorting algorithm when it operates on an input of a given size.

Example: A decision tree for sorting three elements is illustrated in Figure 1:

Figure 1. Decision tree for sorting 3 elements.



The length of the longest path from the root of a decision tree to any of its leaves represents the worst-case number of comparisons the sorting algorithm must perform. Consequently, the worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree. A lower bound on the heights of decision trees is therefore a lower bound on the running time of any comparison sort algorithm.

Theorem: Any decision tree that sorts n elements has height $\Omega(n \log n)$.

Proof: Consider a decision tree of height h that sorts n elements. Since there are n! permutations of n elements, each permutation representing a distinct sorted order, the tree must have at least n! leaves. Since a binary tree of height h has no more than 2^h leaves, we have $n! \le 2^h$ which by taking logarithms implies $h \ge \log(n!)$ since the log function is monotonically increasing. From Stirling's approximation, which is given by

$$n! = \sqrt{2\pi n}\ (n/e)^n\ (1 + \theta(1/n)).$$

This implies $n! > (n/e)^n$ where e = 2.7182... is the base of the natural logarithm. Thus,

$$h \ge \log (n/e)^n$$
$$h \ge n \log n - n \log e$$
$$h = \Omega (n \log n).$$

Theorem: The lower bound for the sorting problem is $\Omega (n \log n)$.

Proof: This follows from the theorem above.

2.1.2 Oracles: A Tool for Establishing Lower Bounds

An oracle is a fiendish enemy of an algorithm which at every opportunity tries to make the algorithm do as much work as possible. The idea is that an algorithm tries to solve the problem in as few time as possible while the oracle tries to foil this by making the algorithm do as much work as possible.

Example: Finding the Minimum of a Set with n Elements

As we have mentioned earlier, an oracle attempts to make the algorithm does as much work as possible, while the algorithm itself attempts to do as little work as possible. In short, the lower bound is constructed via an adversary technique, i.e., the algorithm attempts to do as little work as possible while the oracle spoils this by making the algorithm do as much work as possible.

For purposes of lower bound computation, we assume without loss of generality that the elements in the set are distinct. The basic operation of the algorithm is a comparison operation that compares two elements of the set at each step. When two elements are compared, the smaller value is termed as the "winner" while the higher value is termed as the "loser". At any stage of the in the algorithm, we will have two types of elements:

L - elements that have lost to some other elements

W - elements that have not lost to some other elements

In the beginning we only have type W elements. The algorithm designer has the option to compare two elements x and y where x or y may come from L or W, i.e.,

x	y
L	L
L	W
W	L
W	W

The adversary strategy (or the strategy of the oracle) in order to make the algorithm do as much work as possible is to adopt the following answer when x and y are compared:

x↓ y→	W	L
W	either	x<y
L	x>y	either

Note that the objective of the oracle is to make the size of W stay in its present size (note that the size of W cannot increase so that the best it can do is to make it stay in its present size every time a comparison is made). For example, if x is from L and y is from W, the oracle will make sure that x>y so that the size of W will be maintained.

The objective of the algorithm, on the other hand, is to reduce the size of W. The algorithm stops when the size of W is one. This one element in W if, of course, the minimum element.

The algorithm will be successful in reducing the size of W if it compares two elements from W each time. This is because if it compares two elements from W, one of the elements will move to L regardless of what the adversary says.

Clearly, W whose original size is n is reduced to one after n-1 comparisons of elements coming from W. Thus the lower bound for this problem is n-1.

2.1.3 Problem Reduction

Another approach that can be used to prove the lower bound of a problem P is to show that an algorithm for solving P, along with a transformation on problem instances, could be used to construct an algorithm to solve another problem Q for which a lower bound is known. Some popular examples of this are: reduction of context-free language recognition to matrix multiplication and the mutual reductions between Boolean matrix multiplication and transitive closure..

Example: A string is a cyclic shift of another string when the characters in the cyclic shift are in the same relative order as the original string, but starts at a different position. For example, the strings

ringst

ngstri

are cyclic shifts of

string.

Problem P (Cyclic Shift): Consider two strings S and C of the same size. Determine whether C is a cyclic shift of S or not.

Problem Q (String Matching): Given a text T with n characters and a pattern P with m characters. Check if P is a sub-string of T.

We show that P can be reduced to Q. First, we form a string SS, then one can verify that C is a cyclic shift of S if and only if C is a sub-string of SS. We can therefore apply the string matching algorithm to the pattern C and text SS.

Since the lower bound of string matching is known, the lower bound for cyclic shift is therefore also known.

2.2Upper Bounds

We mentioned that the upper bound for a problem is dictated by the best worst-case existing algorithm, that is, we consider all the discovered algorithms and their complexities, the upper bound is the complexity of the best algorithm.

An upper bound is set when a new algorithm is introduced and this new algorithm has a better complexity than any of the existing algorithms for the problem. All that is needed is to analyze the new algorithm and show that it is better than existing ones.

Upper bounds, however, is sometimes differentiated as practical upper bound and theoretical upper bound. Practical upper bounds are produced by algorithms that can easily be implemented and the theoretical upper bounds are produced by algorithms that when implemented will run worse than the practical upper bound. Normally, theoretical upper bound bounds are shown theoretically to have a lower complexity than practical upper bounds. One opportunity for research is to narrow the gap between theoretical upper bound and practical upper bound.

Example: A practical upper bound for the selection problem was shown by Blum, et. al. (1973) to require 5.4305n comparisons. Schonhage, et.al. (1976) improved this by showing a theoretical upper bound of 3n + o(n) comparisons. Albacea (1992) tried to narrow this gap by producing three algorithms that runs using 5.3975n, 4.9118n and 4.8937n comparisons.

This area of research is aside, of course, from narrowing the gap between a theoretical upper bound and lower bound for a problem. There are two approaches of narrowing the gap between the two bounds. One is to take the algorithm exhibiting the upper bound and then try to reduce its complexity by improving the said algorithm. Or alternatively, one can simply design a totally new algorithm with a better complexity from the existing algorithm exhibiting the upper bound.

Example: The problem of finding the closest pair in a set of n points in a Euclidean plane was solved using a divide and conquer method. The original solution runs in $O(n^2 \log n)$ time, but the same algorithm was later improved to run in $O(n \log^2 n)$ time and ultimately improved to run in O(n log n) time. This is an example taking an algorithm that exhibit the upper bound and then the same algorithm is improved.

The approach of totally designing a new algorithm is exhibited in the solution to the problem of finding the convex hull of a set of points in a plane. This was solved first in Graham (1972) using the algorithm which was later called the Graham's Scan algorithm. The algorithm runs in O(n log n) time. But this was later improved in Jarvis (1973) with the introduction of the Jarvis March algorithm which runs in O(nh) time where h is the number of points in the convex hull. The improvement to Graham's Scan, however, is achieved only when $h < \log n$.

3.SPECIAL CASES

There are so many problems that have been solved using a general input. Sometimes one can produce algorithms with better complexity when the input is restricted. For example, a problem on graphs may have been solved already or in some cases is difficult to solve. By restricting the inputs to say trees one may be able to produce more efficient algorithms.

Example: The bipartite drawing problem for general bipartite graphs was shown by Garey and Johnson (1983) to be an NP-complete problem. But this was shown to be solvable in polynomial time for bipartite permutation graphs by Spinrad, et.al. (1987) and for trees by Shahrokhi, et.al. (2001). Albacea (2005) solve the problem for 2-dimensional meshes in polynomial time also. Specifically, Shahrokhi, et.al. (2001) presented an $O(n^{1.6})$ time algorithm for trees. Paglinawan and Albacea (2004) improved the running time for trees to O(n log n). Albacea (2006a) solved the case of complete binary trees by giving an O(n) algorithm. Later, the case for binary trees was solved in O(n) time in Albacea (2006b).

4.IMPROVING THE BEST PRACTICAL ALGORITHM

Given an algorithm considered to be the best practical algorithm. One obvious research area is to find an improvement or an alternative algorithm that runs faster when implemented.

Example: It is well known that Quicksort is the most practical sorting algorithm. Hence, when one is presented with the sorting problem one most probably will use Quicksort to solve the problem. However, Quicksort has some weaknesses. One such weakness is its worst case being $O(n^2)$. Although fast on the average, with a bad input, it may take a lot of time. Worst being recursive, the algorithm will run out of memory when executed with a bad input. The research opportunity is this case is to improve the worst case of Quicksort without sacrificing its average case performance. Albacea (1995) improved the worst case of Quicksort by introducing a Quicksort-based sorting algorithm called Leapfrogging Sampelsort. This new algorithm improved the worst case from $O(n^2)$ to $O(n\log^2 n)$. However, this was done at the expense of degrading a little bit the average case running time.

5.CONCLUSIONS

In order to find a problem, one can go through the literature and check if a problem has an established lower bound. If none, then this certainly is an opportunity for research. Regardless of whether the lower bound is established or not, one can check the upper bound. One research opportunity is to reduce the upper bound. You can, however, reduce the upper bound only when the upper bound and the lower bound for the problem do not match. But, if the lower bound and upper bound match, then one can simply get the best practical algorithm and try to improve it.

A practical algorithm can be improved by refining the existing algorithm. The refinement should either make the algorithm run faster or produce an algorithm with a better theoretical running time than the existing algorithm. Alternatively, one can simply introduce a new practical algorithm that will run faster the existing one.

On the other hand, if the input to the problem is a general input, then one can certainly work on the special cases.

6.REFERENCES

[1] Albacea, E.A. Complexity of Serial and Parallel Algorithms, PhD Thesis, Australian National University, 1992.

[2] Albacea, E. A. Leapfrogging samplesort. Lecture Notes in Computer Science 1023, Dec 1995, 1-9.

[3] Albacea, E.A. Design and Analysis of Algorithms: An Introduction, JPVA Publishing House, 2003.

[4] Albacea, E.A. Bipartite drawing of 2-dimensional meshes, Proceedings 5th Philippine Computing Science Congress (PCSC 2005), University of Cebu, Cebu City, March 4-5, 2005.

[5] Albacea, E.A. A linear algorithm for bipartite drawing with minimum edge crossings of complete binary trees, Philippine Computing Journal, Vol 1, Number 1, March 2006a, 1-5.

[6] Albacea, E.A. Bipartite drawing with minimum edge crossings of binary trees and 3-Cayley trees, 6th Philippine Computing Science Congress (PCSC 2006), Ateneo de Manila University, Metro Manila, March 28-29, 2006b.

[7] Blum, M, Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E., Time Bounds for Selection, J. Comput. Systems Sci., 7 (1973), 448-461.

[8] Cormen, T.H. Leiserson, C.E. and Rivest, R.L. Introduction to Algorithms, MIT Press, 1990.

[9] Garey. M.R. and Johnson, D.S. Crossing number is NP-complete, SIAM J. Algebraic and Discrete Methods 4 (1983), 312-316.

[10] Graham, R.L. An efficient algorithm for determining the convex hull of a finite planar set, Information Processing Letters 1 (1972), 132-133.

[11] Jarvis, R.A. On the identification of the convex hull of a finite set of points in the plane, Information Processing Letters 2 (1973), 18-21.

[12] Paglinawan, N.M. and Albacea, E.A. Bipartite drawing of trees with minimum edge crossings, Proceedings 4th Philippine Computing Science Congress (PCSC 2004), University of the Philippines Los Baños, February 14-15, 2004.

[13] Preparata, F.P. and Shamos M.I., Computational Geometry: An introduction, Springer-Verlag, 1985.

[14] Shahrokhi, F., Szekely, L.A., and Vrto I. On bipartite drawings and the linear arrangement problem, SIAM J. Computing 30 (2001), 1773-1789.

[15] Shnonhage, A., Paterson, M., and Pippenger, N., Finding the median, J. Comput. Systems Sci. 13 (1976), 184-199.

[16] Spinrad, J, Brandstadt, A., and Stewart, L. Bipartite permutation graphs, Discrete Applied Mathematics 19 (1987), 279-292.