EXPERIMENTAL ALGORITHMICS

- A Brief Explanation

Divye Gupta

B.Sc. Student

Saint Mary's University

1. Introduction

The method used to simplify the complex operations by using a set of rules or a process is called an algorithm. The two main concerns that we have to keep in mind while developing an algorithm is design of the algorithm and analysis of the algorithm.

2. Algorithm Analysis and Design

Algorithm Analysis analyzes the efficiency of an algorithm under specific conditions and assumptions. It measures the amount of time and storage an algorithm takes to execute and perform the required functions. The analysis is done by testing the algorithm with different loads of data and measuring the corresponding time taken to perform the function. Algorithms are developed not only to solve all the computer problems but more importantly to solve real world problems through computers. Algorithm Design focuses on modifying an algorithm to make it faster with returning the high quality of execution.

2.1 The Approach

Developing an algorithm for a problem is not hard, difficulty comes when you have to develop an algorithm keeping quality and time in mind. It's not easy to develop an algorithm that is fast as well as efficient in quality. You may have to compromise on one element to satisfy the other. Abstraction and mathematical proofs have been the tools to develop an algorithm from centuries. An algorithm is typically described in pseudocode rather than real code so that the explanation is clear and concise. Once, we obtain the pseudo code, it can then be implemented in any language of our choice. Finding the closest upper bound of the algorithm performance is the first thing we have to keep in mind for developing an algorithm. In the introduction, the author forgets to mention the three big notations that can be used to measure the complexity of an algorithm: Big O notation, Big Omega notation and Big Theta notation. She uses Big O notation to measure the upper bound of an algorithm. Ideally, Big O notation is used to describe the limiting behaviour of an algorithm. Big O is used to describe the order of the algorithm and hence describes the growth of a function. For example, the complexity of QuickSort algorithm is $O(n^2)$ for the worst case. That means, no matter how bad the data is, while sorting the data, QuickSort won't grow over the order of 2.

Let's have a look at the pseudocode of QuickSort algorithm and how it works:

```
Algorithm QuickSort(s, firstPosition, lastPosition)
=====================================================
if (firstPosition < lastPosition)
   Partition(s, firstPosition, lastPosition, pivotPosition)
   QuickSort(s, firstPosition, pivotPosition-1)
   QuickSort(s, pivotPosition+1, lastPosition)
Algorithm Partition(s, firstPosition, lastPosition, pivotPosition)
=================================================================
Choose the middle position in [firstPosition, lastPosition] to be the pivotPosition
Exchange the value in the first position with the value in pivotPosition
Let an index called lastSmall point at the new pivotPosition (the first position)
for each value from the second position onward
   if the value in that position is < the pivot (the value in pivotPosition)
      Increment lastSmall and exchange the value in that position with the one at lastSmall
Exchange the value at the pivotPosition with the value at the lastSmall position
Return this new pivotPosition (as a reference parameter if this is a void function)
```

The basic idea of QuickSort is to choose a value from the elements and call it at as "pivot". In this algorithm we pick the middle value as pivot. So once the partition function is completed, all the elements smaller than the pivot are placed on the left hand side of the pivot and all the elements greater than the pivot are placed on the right hand side of the pivot. Pivot comes at the right position where it should be in the sorted list. QuickSort function is called again with left-sublist and right-sublist.

The analysis and design of an algorithm not only find the properties of an algorithm but also opens a way for discovery and innovation. It helps improve the algorithms and implement new algorithms that are much faster and efficient than the existing algorithms. However, it is not that easy to find an accurate time prediction of an algorithm by finding the limiting behaviour of the function *(asymptotic time bound).* One of the problems that we face is finding the worst case of an algorithm. Author nicely explains this problem by giving us the example of Quicksort algorithm. An algorithm like Quicksort may work slow for 100 elements but it may be fast for a 1000 elements and may run thousand times faster for a million elements. The gap between the worst case of an algorithm and the best case increases as we increase the number of elements. Another problem that author mentions is the difficulty in using the "algorithm" for real world programs. A program goes through a number of processes when it runs on an operating system. Those processes requires the call to main memory of the CPU unlike the algorithm. On the other hand, an algorithm is tested upon the sample data.

3. Experimental Algorithmics

It's impossible to comment about the algorithm without experimenting with it on a computational environment. Author excellently explains us the meaning of the term Experimental Algorithmics and why it is so important to use it in Algorithm Analysis and Design. We get a much more accurate information about the performance of an algorithm when we test it on real applications. Just like chemistry, classic laboratory experiments are being used to analyze the algorithms. The experiments include, carefully placing the data in the algorithm with different sets of data and then analyzing and comparing each of them. Such a technique is best to maintain good balance between theoretical knowledge and practically using it. We learn more when we actually practice the problem than just reading it. Algorithm Engineering is a similar term to Experimental Algorithmics. The only difference between the two terms is that

Algorithm Engineering focuses more on design of an algorithm whereas Experimental Algorithmics focuses on analysis of the algorithm.

To support Experimental Algorithmics author has provided three brilliant examples in which Experimental Algorithmics has produced remarkable progress and some new results. In this paper, we will only have a look at the first two examples discussed by him. The examples also shows a variety of questions and methods that can be explained using Experimental Algorithmics. Following are the examples that are explained in the paper.

I. Memory-efficient models of computation

The traditional approach to figure out whether a computer is memory efficient or not is counting the number of basic operations performed on a theoretical model of a computer like turing machines. An abstract computer or theoretical model of a computer allow a detailed analysis of how a computer works. The basic operations include different processes a computer executes to perform a specific function. However, on today's computers different processes take different amounts of time to execute. Sometimes, accessing the memory can take relatively longer than actually performing a function. Author talks about the placement of operands to be the primary issue of the wait but sometimes the wait can also be due to a race condition. Race condition is a situation when a system wants to run two or more processes or two or more processes want to access the memory at the same time but the system can only perform these operations one by one. Hence, the other processes have to wait for their turn to access the memory resulting in more time to access the memory.

New analytical models of computation help us to describe the cache *(component that stores data so that future requests for that data could be served faster)* performance of an algorithm. An algorithm can either be cache-oblivious, that takes advantage of the cache without knowing the size of the cache or it can be cache efficient that takes care of cache miss. Author gives an example of Quicksort's cache performance that solely depends on how the partitioning of the list is structured and what time the sublists are processed. Hence, an algorithm that may have less or minimum instructions can be really inefficient because of the poor cache behaviour. The research on analyzing memory access patterns and better algorithms is a never ending topic but engineers are able to find algorithms that are cache efficient in fields like dynamic tree, tree structures, matrix operations, sorting and permutation problems.

II. Phase transitions in combinatorial problems

What exactly is a phase transition? Melting of ice to form water? Or condensing of steam? Those are real examples of phase transitions but how are we going to use that in algorithms? Phase transition is simply the transition of a problem from a state where it has a number of solutions to a state where it can have no solution. Author talks about phase transition for NP-complete problems. An NP-complete problem is a problem that can be solved in polynomial time (O(n^k) by a turing machine or a theoretical model of a computer *(as stated above)*. NP-complete problems cannot be solved in realistic time hence polynomial time is taken. Even in polynomial time, there’s no known way to solve NP-complete problems. Author has explained the importance of phase transition by using the example of QuickSort . QuickSort may act differently depending on the pivot chosen on every iteration. Sometimes it may be faster than any other algorithm but other times it can be a big pain. Hence the gap between the best and worst case could be really huge.

Next thing author talks about is the origin of phase transition but missed to talk about **Erdos-Renyi Model** of threshold in which phase transition was first observed in 1959.

$$B = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge$$
$$(\overline{x_1} \vee \overline{x_2} \vee \overline{x_5}) \wedge (x_3 \vee x_4 \vee x_5).$$

Dr. Catherine discusses this problem as an example of K-satisfiability problem. This is a boolean satisfiability problem that shows that there should be at least one solution that satisfies this propositional formula. A problem must have at least 3 variables to become a NP complete problem otherwise it’s very easy to find out the solution of the problem and hence become a known problem. This problem contains five variables, four clauses and three variables per clause:

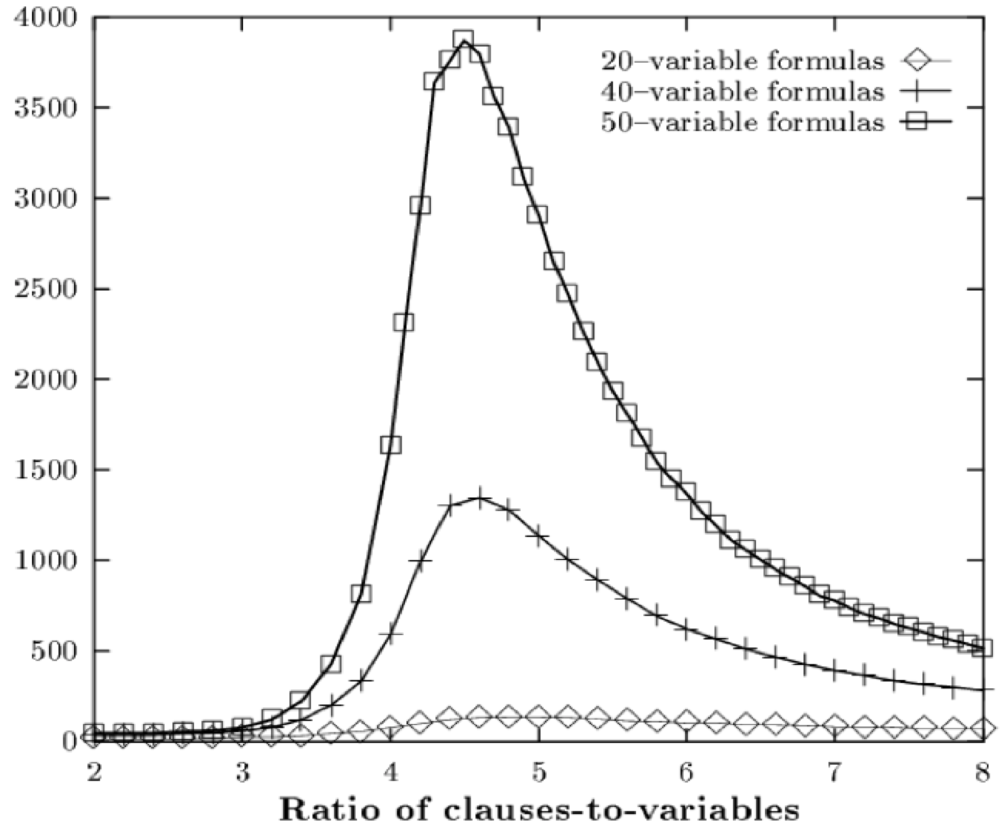
Five variables: x1, x2, x3, x4, and x5

Four clauses: (x1 V x2’ V x3), (x1’ V x2’ V x4’), (x1’ V x2’ V x5’), (x3 V x4 V x5)

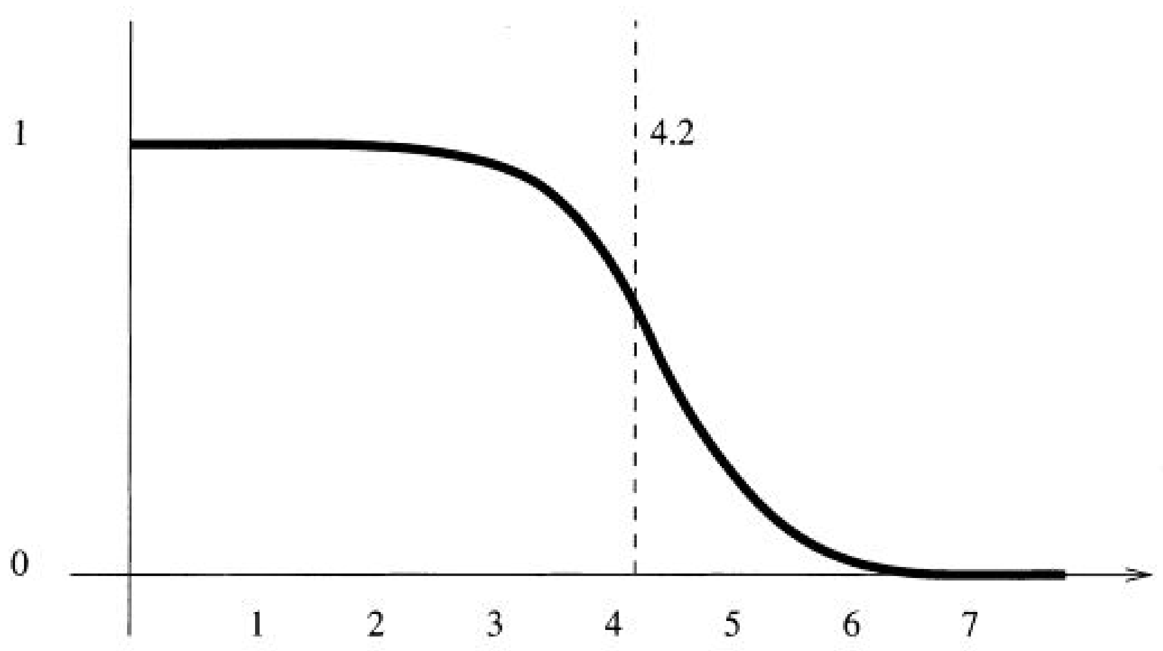
The problem is satisfiable only if it returns true for any combination of values of the clauses. The solution to this problem is x1, x3 = 1 or true and x2, x4, x5 = 0 or false.

Is it easy to find a solution to such K-satisfiability problems? Not really. The best known algorithm to find a solution to such a problem is to find 2^n possible truth assignments. For a problem with 50 variables it would take about 435 years to compute the solution of the problem that is obviously very inefficient.

Now, if we have a completely random B_{nmk} problem where *n* is number of variables, *m* is number of clauses and *k* is number of variables per clause. The ratio of number of variables and number of clauses is *m/n or X* is taken out to find out the satisfiability of a problem. After doing certain number of experiments on such problems researchers have come to a conclusion that when *k* is greater than 2, there exist a value X_k or *threshold value* when X is less than X_k the probability that B is satisfiable is nearly 1 but otherwise the probability goes down to near 0.



Various experiments have been conducted to find the satisfiability of an NP-problem over the course of years. Most of the experiments have come to the conclusion of easy-hard-easy phenomenon for random satisfiability problems. It's hard to prove that a problem is satisfiable when we have a large number of clauses, since the clauses can impose too many conflicts on the variables. On the other hand, it is easier to prove that a problem is satisfiable when we have a small number of clauses compared to the number of variables because many satisfying truth assignments can exist. It's observed that this easy-hard-easy transition is applied on various other problems like Graph Coloring and Number Partitioning. Let's have a look at the following graph showing phase transition of a random satisfiability problem:



On the Y axis, we see the probability of a problem to be satisfiable. X axis is the ratio of number of clauses to number of variables (*m/n or X*). In this case, our X_k *or threshold value* ≃ 4.2 . Hence, the problem is satisfiable when the threshold value is below 4.2 but the probability drops down when the threshold value becomes larger than 4.2.

4. Conclusion

Research on experimental analysis of algorithms is evolving day by day. It has become mandatory to analyze an algorithm to know about its practical relevance. This paper has discussed some great information about experimental algorithmics by giving us theoretical knowledge as well as applying them in real experiments. Author has beautifully used half of the paper to set the base of the reader by explaining all the terms related to analysis and design of an algorithm. After setting the base, she explains the importance by using some sample problems. The best thing about her examples is that she has used three completely different types of examples which are nowhere related to each other, that shows the multiplicity of Experimental Algorithmics.

5. Resources

- Kleinberg, Jon. "The Mathematics of Algorithm Design."

  https://www.cs.cornell.edu/home/kleinber/pcm.pdf

- Otero, M. (2014, May 26). The real 10 algorithms that dominate our world.

  https://medium.com/@_marcos_otero/the-real-10-algorithms-that-dominate-our-world-e95fa9f16c04

- (2006). Intro to Algorithms: CHAPTER 1: INTRODUCTION.

  http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap01.htm

- (2005). Abstract Machine -- from Wolfram MathWorld.

  http://mathworld.wolfram.com/AbstractMachine.html.

- Blelloch, G. E., & Harber, R. (2013). Cache and I/O efficent functional algorithms. *ACM SIGPLAN Notices*. ACM.

  https://www.cs.cmu.edu/~rwh/papers/iolambda-cacm/cacm.pdf

- Butaru, M., & Habbas, Z. (2008). The Phase Transition Behaviour of Non-binary Forward Checking Algorithms. *Stud. Inform. Univ.*, *6*(3), 236-260.

  http://studia.complexica.net/index.php?option=com_content&view=article&id=109

- (2011). NP-completeness. https://web.cs.dal.ca/~nzeh/Teaching/3110/Notes/np.pdf.

- Navarro, J. (2005). Generation of Hard Non-Clausal Random Satisfiability

  https://www.aaai.org/Papers/AAAI/2005/AAAI05-069.pdf.