Summary of "A Theoretician's Guide to the Experimental Analysis of Algorithms"

Kafa Alameh

July 5, 2018

1 Introduction

Analyzing algorithms can be distinguished in three different ways: worst case analysis, average case analysis, and experimental analysis. In the past, emphasis on rigorous and theoretical modes of analysis dominated experimental studies. However, more experimental work is being recently tackled because of the insights it can give about real-world algorithmic performance. For this reason, theoreticians are being involved with experimental work, inspiring new research, and establishing connections to applications. In this paper, the author is interested in the influence of a theoretical analysis of algorithms on experimentation, scientific bias, and how a theoreticians' background can drive a more scientific approach to experimentation.
The following are different reasons why an algorithm is implemented, each giving rise to a different type of paper:

1. **Application paper:** describes the impact of the algorithm in the context of the application. Such papers are interested in the output of the algorithm and not its efficiency.
2. **Horse race paper:** provides why the algorithm is better than its competitor.
3. **Experimental analysis paper:** illustrates the strengths and weaknesses of the algorithm in practice.
4. **Experimental average-case paper:** investigates the average-case behavior of the algorithm.

In this paper, the author focuses on the third type of papers by discussing principles on how to write an experimental paper. He mentions pet peeves (personal annoyance: something that is misleading in papers or practices), pitfalls (temptations or practices that can result in a large waste of time or computation), and suggestions on how to improve experimental research.

2 Discussions of the Principles

Principle 1. Perform Newsworthy Experiments

All scientific papers should have new interesting results, especially experimental algorithms papers. For example, problems with no direct applications are harder to justify using experiments than using theory. In addition, problems with no direct applications do not have real-world instances. The experimenter is hence left to invent and justify test cases which referees may question the testing code. Another side of newsworthiness has to do with the algorithm studied. A typical question one would ask is whether the algorithm studied will be a useful alternative in practice.

- Pitfall 1. Dealing with dominated algorithms:
  Dominated algorithm: one that is always slower than some already well-studied competitor. A lot of the experimental literature is devoted to dominated algorithms.

- Pet Peeve 1. Authors and referees who don't do their homework:
  What to do if you discover that your algorithm is dominated? If it is new and not written about, few people will be interested to know about it. If it is known and not adequately studied before, there are several situations:

  1. If the algorithm is in use or the ones that dominate it are complicated: a study of the algorithm's performance and comparison to the more sophisticated one may be of interest.
  2. If the algorithm in use has a wide range of applications so that studying how to adapt the algorithm may have general consequences.
  3. If the algorithm is known and expected to behave poorly, it is worth including the negative result in a general paper that talks about undominated algorithms.
  4. If the algorithm is of interest and the fact that it is undominated is unexpected. It is worth noting that it is dominated in a comprehensive study.

  Note that since run-time depends strongly on implementations or machine/operating system interactions, establishing domination is not easy. In addition, you must convince the readers that your results are correct and you did not misimplement the algorithm.

A third dimension of newsworthiness is looking for generality and relevance in your results. For instance, don't just report that algorithm A outperforms algorithm B. Identify and explain why it happens.

- Pitfall 2. Devoting too much computation to the wrong questions:
  Computation time includes your own time.

  1. Don't spend too much time excessively studying results for a few instances.
  2. Don't run experimental tests before finishing up an efficient implementation or choosing what data to collect.

- Suggestions 1 & 2 Think before you compute & Use exploratory experimentation to find good questions:
  Take your time to find a good problem. Before starting collecting data, decide what questions you want to answer. Since finding a solution will take time and resources, think before you compute and use exploratory experimentation to find good questions.

- Pitfall 3. Getting into an endless loop in your experimentation:
  Do not get into the endless loop of experimentation. Learn how to draw the line and leave some questions for future work.

- Pitfall 4. Start by using randomly generated instances to evaluate the behavior of algorithms, but end up using algorithms to investigate the properties of randomly generated instances

Principle 2. Tie Your Paper to the Literature

You should study the literature before attempting any experimental project. Studying the literature can guide the experimentalist in the right direction of research by suggesting the right questions to ask and experiments to avoid.
If an earlier paper has studied instances that are still considered interesting, a comparison should be made between your results and those in the paper. You should note that an earlier paper dealing with the same algorithm doesn't necessarily mean dealing with the same test instances or with the same implementation of your algorithm.
If your aim is to give comparisons to a previously studied algorithm, you should obtain the code of its implementation and provide experiments on your machine.
If it's not possible to obtain the implementation, you must develop a comparable implementation. If the two implementations do not have the same run-time, an explanation should be provided. Note that your implementation must be the efficient one.
Even when previous experiments were performed on some unfamiliar architecture, try to make some ballpark comparisons of machine speeds in order to address the question of run-time competitiveness.

Principle 3. Use Instance Testbeds that Can Support General Conclusions

There are two types of instances: instances from real-world applications and instances that are randomly generated given a list of parameters. However, random instances need to be structured in a way that they reflect real-world applications.
Random instance generators allow experimentalists to generate instances of large sizes, which allows them in return to determine the biggest instance they can run on their machine and some idea about the speed and memory needed for future machines to run even larger instances.

- Pet Peeve 2. Concentration on unstructured random instances:
  Random instances may tell us little about real-world performance and deceive us about the difficulty of the problem.

- Pet Peeve 3. The millisecond testbed:
  In case of a 1 second difference in run-time, this 1 second difference is irrelevant. Moreover, if an algorithm is 10 faster than another, the difference is only relevant when one is testing large instances and should test if the 10 times factor persists as the instance size grows.

- Pet Peeve 4. The already-solved testbed:
  While testing, one would tend to restrict the testing to instances for which the optimal solution is known in order to compare the results. However, this doesn't allow the experimentalist to address how the algorithm's performance scales with instance size. Moreover, by evaluating approximation algorithms, we can explore the domain for which the algorithm were intended. Restricting to instances for which the optimal value is known doesn't give information about how the algorithm's performance scales with instance size.

Principle 4. Use Efficient and Effective Experimental Designs

- Suggestion 3. Use variance reduction techniques:
  When dealing with randomized algorithms, one need to do a large number of runs to obtain reliable averages. Using variance reduction techniques, one can reduce the number of runs.

- Suggestion 4. Use bootstrapping to evaluate multiple-run heuristics:
  Suppose you want to perform k runs of an algorithm and take the best solution found. A naive approach to the problem would be to run the algorithm some number n times and compute the average of the n best of k. However, a better approach would be to sort the results in order of solution value and compute the expected best value of the solution.

- Suggestion 5. Use self-documented programs:
  The experimenter's time can be reduced by having helpful README files, descriptive file names and output files that contain useful information about experiment the that generated them.

Principle 5. Use Reasonably Efficient Implementation

This is a controversial principle since efficiency comes at a cost in programming effort.

- Pet Peeve 5. Claiming inadequate programming time/ability as an excuse:
  Researchers may claim that if they had the time or skill to use the same speed-up mechanisms as the previous implementation, their implementation would likely be competitive. First, one cannot measure the additional speed-up that could be gained unless the speed-up mechanisms were implemented. Second, a speed-up mechanism that works on some algorithm doesn't necessarily mean it will work on another. Third, since one implementation is slower than the other, one cannot handle large instances. consequently, comparative data will be missed.

If one is studying the quality of solutions produced by an approximation algorithm (which are independent of the speed-up mechanisms), one may tend to save programming time at the expense of additional computation time.
More efficient code means you can perform experiments on more and larger instances.
It also makes it easier to support claims of competitiveness.

- Pitfall 5. Too much code tunning:
  Make use of efficient data structures by profiling early your implementation.

Principle 6. Ensure Reproducibility

Reproducibility means that if you run the same code on the same instance and on the same machine/compiler/operating system, you should get the same results (run-time, solution quality) or the same averages in case of randomized algorithms. Your experiments should be extensive enough to assure you that the conclusions are correct and are not an artifact of your experimental setup.

- Pet Peeve 6. Supplied code that doesn't match a paper's description of it:
  Some researchers provide code that doesn't precisely implement the algorithm the paper is describing. An important algorithmic measurement, which is run-time, is inherently irreproducible since it depends on the machine.

- Pet Peeve 7. Irreproducible standards of comparisons:
  Suppose you are evaluating approximation algorithms for which no optimal solution is found. The following are suggestions to evaluate the relative goodness of a solution:
  1. *Simply report the solution value.*
  2. *Report the percentage excess over best solution currently known.*
     The current best solution and your instances must be available.
  3. *For randomly generated instances, report the percentage excess over an estimate of the expected optimal.*
     The estimate must be stated or the method of computing it is specified.
  4. *Report the percentage excess over a well-defined lower bound.*
     It is a useful approach assuming the lower bound can be calculated or approximated and close to the optimal solution.
  5. *Report the percentage excess/improvement over some other heuristic.*
     For the result to be reproducible, the other heuristic must be specified and preferably simple.

- Pet Peeve 8. Using running time as a stopping criterion:
  In case of approximation algorithms, one may simply run an algorithm for some time and take the best solution found so far. However, such results are irreproducible because different solutions will be obtained if one uses a different implementation on the same machine or the same implementation on a different one. In order to

solve this problem, you may set some criterion such as the number of neighborhoods searched as a stopping criterion.

- Pet Peeve 9. Using the optimal solution value as a stopping criterion:
  Algorithms that simply look for very good solutions until some stopping criteria is reached without having any mechanism for verifying optimality. Often, the optimal solution of the approximation algorithm is not known and hence the stopping criteria cannot be used.

- Pet Peeve 10. Hand-tuned algorithm parameters:
  Many algorithms have parameters that must be specified before performing a run. One can either fix those parameters or define them in such a way that they depend on measurable instances. In the first case, as long as the paper reports the parameters, there is no problem with reproducibility. However, in the second case, an explanation must be provided about the different settings chosen. The time spent in determining the parameters must also be stated.

- Pet Peeve 11. The one-run study:
  In order to have reproducible results, a study should cover a wide range of instances and a large number of runs.

- Pet Peeve 12. Using the best result found as an evaluation criterion:
  With randomized algorithms, researches often report the best result found from several runs instead of reporting the average result.

Principle 7. Ensure Comparability

Write your papers in such a way that future researchers can compare their results to yours. This can be achieved by making relevant data, code, and instances available publicly.

- Pet Peeve 13. The uncalibrated machine:
  Even though most papers mention the processor speed, operating system and language/compiler of the machine they used to perform their experiments on, it can still be difficult to estimate the relative difference in speeds between earlier and recent work.

- Suggestion 6. Use benchmarks codes to calibrate machine speeds:
  This allows future researches to calibrate their machines in the same way and normalize old results to their own machines.

- Pet Peeve 14. The lost testbed:
  When you make your instances available for future researchers, this may give them the chance to beat you by optimizing their codes for precisely the instances you provided. However, this could be avoided by having large and varied testbeds. This can also be avoided by making your source code available, if possible, along with machine calibration and testbed instances.

- Pitfall 6. Lost code/data:
  Losing code or data can be the cause of code modification without saving a copy of the original version, unforeseen cleanups, failure to keep backups and poorly organized directories.
  By losing code, you lose future comparability and by losing data, you lose the chance to answer questions raised by readers or future papers.

Principle 8. Report the Full Story

Meaningful summaries of your data must be provided. Averages (and how you computed them), distribution of results and any normalization or scaling of the measurements must be stated.

- Pet Peeve 15. False precision:
  Representing averages to far more digits of accuracy than is justified by the data. In addition, anomalous results should be reported instead of hidden and justified or explained if possible.

- Pet Peeve 16. Unremarked anomalies:
  Unnoticed anomalies are the worst kind of anomalies because they leave the reader wondering whether they are true anomalies or a typographical error.

- Pet Peeve 17. The ex post facto stopping criterion:
  It concerns using the best optimal solution as a stopping criterion from the point of view of telling the full story. Many search heuristics continue searching for a solution instead of stopping after finding a local optimum. They proceed by reporting the run-time of finding the optimum solution rather than the overall run-time, which underestimates the total run-time. A better approach would be to also report the number of steps or iterations taken before the best solution was found.

- Pet Peeve 18. Failure to report overall running times:
  Papers fail to report running times for the following reasons: the main subject of their study is only one component of the running time or is a combinatorial count related to the algorithm's operation. However, readers may be interested to know the contribution of the component or the meaningfulness of the component to the overall run-time. Moreover, if your paper concerns approximation algorithms, you may either be concerned about the quality of solution produced or an estimate of the average-case solution quality. However, since approximation algorithms present a trade off between between quality of solution and run-time, readers will be interested in knowing the run-time.

Principle 9. Draw Well-Justified Conclusions and Look for Explanations

- Pet Peeve 19. Data without interpretation:
  In order to convince the reader that the data supports the conclusions and conjectures

stated, data must be summarized and interpreted.

- Pet Peeve 20. Conclusions without support:
  Data presented must support the conclusions drawn. Mistakes often made are ignoring exceptions or some trends in data.

- Pet Peeve 21. Myopic approaches to asymptopia:
  Experiments try to address asymptotic using small or relatively large instances. These instances do not always forecast behavior for very large instances and hence results in misleading run-time.

- Suggestion 7. Use profiling to understand running times:
  Provide the number of calls to the various subroutines and the time spent in them. Focus on measuring run-times and memory usage in order to evaluate the performance of the algorithm and the quality of the solution it generates in case of approximation algorithms.
  Provide why an algorithm is taking in practice less running time than what is predicted in theory. By profiling and instrumenting the code, we can find answers to these questions. Note that such measurements generally lead to the conclusion that the implementation was defective.

Principle 10. Present Your Data in Informative Ways

- Pet Peeve 22. Tables without pictures:
  Add graphical pictures to summarize and tell a story.

- Pet Peeve 23. Pictures without tables:
  Tables summarizing findings can be added to the appendix.

- Pet Peeve 24. Pictures yielding too little insights:
  It is not useful to present pictures that cannot add much to our understanding.

- Suggestion 8. Display normalized running times:
  If you are comparing the run-time of some algorithms, you will get a clearer idea about relative speeds between them if you normalize their running time.

- Pet Peeve 25. Inadequately or confusingly labeled pictures:
  Using different indistinguishable symbols to present data points.

- Pet Peeve 26. Pictures with too much information:
  Although pictures are informative, they can get unclear if they hold too much information. An alternative would be to have multiple figures representing the information.

- Pet Peeve 27. Confusing pictorial metaphors:
  Don't confuse the reader with too much creativity. Keep things as simple as possible.

- Pet Peeve 28. The spurious trend line:
  A misleading trend line that is an artifact of how you decided to present your data. For instance, such spurious trend line will not be reproduced if you had presented your data differently.

- Pet Peeve 29. Poorly structured tables:
  It is not a good idea to sort your tables by instance or algorithm name. A better practice is to sort tables by the size of instances. This will allow you to spot trends in time or solution quality.

- Pet Peeve 30. Making your readers do the arithmetics:
  Displaying a table of results with missing information about things that the reader wants to know.

- Pet Peeve 31. The undefined metric:
  Labeling a column/axis without explaining the meaning of the label.
  Including a run-time column without specifying whether it is an instance reading preprocessing time, time for a single run, or the total time for all runs.
  Stating the number of iterations or time without explaining their meaning or units used.

- Pet Peeve 32. Comparing apples with oranges:
  This includes comparing algorithms' run-times whose experiments were performed on different machines.

- Pet Peeve 33. Detailed statistics on unimportant questions:
  In order to get the full picture of an algorithm's performance, it is better to investigate its performance on many instances rather than determining statistics on just one instance.

- Pet Peeve 34. Comparing approximation algorithms as to how often they find optima:
  It is a metric that fails to make a distinction between algorithms as the instance size gets larger. In addition, it restricts attention to instances of the optimal solution and doesn't answer the question of how near the optimal does an algorithm gets when it does not find an optimum.

- Pet Peeve 35. Too much data:
  Do not overwhelm the reader with too much data. Instead, use summary statistics to present them.
  Dominated algorithms that are not interesting should be dropped.
  Omitted data can be added to a Web archive.