

INDUSTRIAL AND SYSTEMS ENGINEERING



The Scientific Approach to Algorithm Design and Evaluation

ANDREW V. GOLDBERG¹

¹Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, USA

ISE Technical Report 26T-017-R1



The Scientific Approach to Algorithm Design and Evaluation

Andrew V. Goldberg

ISE Department, Lehigh University, Bethlehem, PA, USA

ag0282@lehigh.edu

October 10, 2026

Abstract

We discuss how to apply the scientific method to algorithm design to develop practical algorithms with a solid theoretical understanding of their performance. The process forms a feedback loop that combines theoretical design and analysis, implementation, experimentation, and analysis of experimental results. This analysis can provide insights for further improving algorithm performance, both in theory and in practice. The scientific method helps mitigate the shortcomings of theoretical analysis and experimental evaluation when conducted in isolation. We focus on closing the feedback loop by giving a theoretical explanation for the experimental results. We also discuss how to keep the research relevant to practice by using real-life instances.

1 Introduction

In this paper we discuss *algorithm science*, by which we mean the use of the scientific method in developing practical algorithms whose behavior we understand. The scientific method is widely used in many disciplines, in particular in the natural sciences. Before becoming a central part of computer science, algorithms were formalized by mathematicians, implemented on hardware designed by engineers, and used by practitioners with mathematical, scientific, engineering, and operations research backgrounds. These communities brought somewhat different perspectives on algorithms.

Hoare [34] gives a mathematician’s view of algorithms (programs):

Computer programs are mathematical expressions. They describe, with unprecedented precision and in the most minute detail, the behavior, intended or unintended, of the computer on which they are executed.

We take the computer science view of algorithms, where the relationship between programs and the application domain is preserved, as observed by Colburn [16]:

In each of these accounts of mathematics, all concern for the content or subject matter of specific terms is abandoned in favor of the *form* of the deductive system. So the abstraction involved results in essentially the *elimination* of content. In computer science, the content is not totally abstracted away in this sense. Rather, abstraction in computer science consists in the *enlargement* of content. For computer scientists,

this allows programs and machines to be reasoned about, analyzed, and ultimately efficiently implemented in physical systems.

Preserving the real-world content implies that, in addition to the formal analysis, we must evaluate algorithm performance in the real world. This includes implementing the algorithms and running them on real-life instances.

Theoretical design and analysis of algorithms is a well-established discipline. One develops algorithms and analyzes them in a formal model of computation using mathematical tools to obtain upper and lower bounds on the complexity of the resulting algorithms. See, e.g., [17]. We express the bounds using the “ O ” or “ Ω ” notation. The bounds are usually worst-case. In some cases, we have average-case bounds for simple input distributions. However, we usually do not know what the real input distribution is. Theoretical analysis is machine-independent, but ignores constant factors.

To have real-world impact, algorithms need to be implemented and evaluated on real-life instances. For a given application, we also need to model the corresponding instances. The goal of this evaluation and modeling is to predict algorithm performance.

One caveat is that many problems arising in computer science and operations research have a human-defined component. In addition to fundamental problems, real-life problems may have human-defined features. Examples of the former are the halting problem and sorting. Definitions of these problems do not depend on the application domain. The vehicle routing problem [24] is an example of the latter. For example, government regulations and operational requirements define driver constraints. Different regions may have different regulations (e.g., EU vs. North America), and these regulations change over time. If autonomous trucks are deployed in the future, they will operate under different constraints. The underlying problem will change and may require substantially different algorithms. Problems related to data center optimization are another example. Data centers are human-made, and their architecture is under our control. Another issue related to algorithm implementation is computer hardware. It also evolves, and new architectures may require different algorithms (e.g., taking advantage of GPUs).

To future-proof algorithms, we need to understand their performance, including its dependency on features such as problem size and other parameters, and problem structure. We also need to anticipate future problems and enable their algorithmic solutions. Therefore, we need to predict future problems, in particular, to identify future problem features and parameters.

Much of the experimental work on algorithms falls into two categories. One is to run one or more algorithms on a set of benchmarks and report the results. If the set of algorithms used includes the state-of-the-art code, the results are useful for future applications that involve similar problem instances. Such an experimental evaluation may also be part of a broader study that includes an analysis of the results and a better understanding of the problem complexity. Note, however, that a study that involves running a bad algorithm on arbitrary instances is of questionable value.

Another approach to experimental evaluation [35, 47, 48] is to measure algorithm dependency on the size and the parameters of the problem. This approach takes a mathematical view of algorithms. In particular, to speed up the process of algorithm implementation, the authors of [35, 47] advocate using non-optimized implementations or simulations in combination with operation counts. The goal is to model algorithm performance as a function of the parameters. However, such studies often do not include real-life instances and do not produce efficient implementations. Hooker [35] criticizes developing optimized implementations and comparing them on real-life

instances. The results involving slow implementations cannot be applied to practice without producing efficient implementations, which involves significant work. Furthermore, due to the limitations of theory, in real life the optimized implementations may not perform as predicted.

These experimental evaluation approaches evolved into the discipline of *algorithm engineering* [55], which focuses on developing efficient algorithm implementations and testing them on both real and synthetic instances. In this paper, we review some of the algorithm engineering practices and discuss how these can be modified to better fit the scientific algorithm development approach.

While both the theoretical and the experimental approaches have their merits, they can be combined using the scientific method. The interaction of the two approaches allows one to mitigate their shortcomings. In addition to observing performance on real instances, one needs to collect additional data that can help in understanding algorithm performance, identifying performance bottlenecks, and determining the parameters affecting performance. One can use synthetic instances to better understand parameter dependence.

Even if the authors of the experimental study are only interested in performance, the additional data can be used by others to provide a theoretical explanation for the observed behavior, which may help improve the algorithms. In algorithm research, the methodology of reading experimental papers and analyzing the data to provide a theoretical understanding of the results is not as common as, for example, in physics. Wider use of this methodology will make theoretical algorithm research more relevant.

Many studies of algorithms use the scientific method implicitly. In this paper, we emphasize the need for experimental work to generate data that can be used to obtain theoretical results, and for subsequent data analysis to obtain such results. We focus on the interaction between experimentation and theoretical analysis. Most examples we use are from our work on shortest path and related problems, where we know the conjectures made and can make them explicit. We give examples of new theoretical results and practical algorithm speedups obtained from the analysis of the experimental data. This illustrates the main benefit of the scientific method.

2 Preliminaries

In this paper, we discuss algorithm performance and heuristics. By performance, we mean the running time, unless mentioned otherwise. By a heuristic we mean an algorithmic enhancement that has no theoretical justification but may lead to practical improvements. Under this definition, if the enhancement gets a theoretical justification, it ceases to be a heuristic.

We use examples from papers on the shortest path problem, the closely related negative cycle detection problem, and the special case of the shortest path problem on road networks. We assume the reader is familiar with the shortest path problem; Tarjan’s book [60] gives a good introduction.

Most of the previous work follows the scientific method implicitly, without explicitly stating conjectures to be verified. In this paper, we spell out the conjectures, emphasizing the scientific approach. To conduct an experimental study, one starts with a conjecture the study aims to verify. This determines the requirements for the experimental part of the study, including the benchmarks and the codes to be compared, and the level of code optimization. For example, one can conjecture that a new algorithm with a better theoretical time bound outperforms previous codes on a certain problem class. Then one needs the corresponding benchmark, the previous

state-of-the-art code, and an implementation of the new algorithm. The collection of benchmark instances and codes forms an experimental framework.

After setting up the experimental framework, one runs experiments, which should produce enough data for statistically valid conclusions. Note that initial experimental results may give feedback for improving the framework. For example, if algorithm running times have a large variance, one may need more instances.

The results of the experiments are analyzed to gain insight into algorithm performance. For example, one can identify the real-life bottleneck, e.g., using profiling. If it is different from the theoretical bottleneck, one may investigate the new bottleneck. A natural research direction would be to attempt to modify the algorithm to mitigate the bottleneck. Alternatively, one can attempt to prove that on the class of benchmark instances used, the observed bottleneck is also a theoretical bottleneck. Sometimes a deep dive into the experimental results may lead to a better understanding of algorithm performance, including theoretical or practical algorithm improvements. This is one of the main benefits of the scientific method.

3 Planning an Experimental Study

The goal of an experimental study is to test a conjecture and collect data on algorithm performance that may be used in a subsequent theoretical analysis of the results. A conjecture can be, for example, “a new algorithm implementation can outperform existing codes on a benchmark”, or “a heuristic improves algorithm performance”. Note that conjectures can be implicit. For example, a study may disprove a folklore belief that the authors had implicitly assumed to be true. The design of an experimental study needs to ensure well-justified results and facilitate subsequent analysis of the data.

The conjecture influences the selection of test instances. If a study focuses on a specific application, one needs a representative set of application-generated instances. To evaluate algorithm dependence on an input parameter, the test set needs instances with different values of the parameter. If no real-life instances with the desired properties exist, one may need to use synthetic instances.

The conjecture also determines which algorithms need to be implemented and how much to optimize the code. If one aims to outperform the current state-of-the-art code, significant optimization may be needed. One also has to include all competing methods that are not known to be dominated on the benchmark instances. If one only wants to show that a heuristic reduces the number of certain operations, there may be no need for optimization.

Note that an experimental study plan is not static. Experiments may show the need for additional experiments or suggest new ideas for further experiments or algorithmic improvements.

4 Benchmark Problem Selection

4.1 Real-Life Instances

Real-life instances are the key to keeping algorithmic research relevant in the real-world context. Selecting the set of test instances depends on the scope of the study. If the study is limited to an existing application domain, one may use existing benchmarks. If the application domain is evolving, one needs to make sure the benchmarks are up to date.

Companies often view their data as proprietary and limit or deny external access to it. In some cases this is justified, for example, by personal user data or trade secret protection. In other cases, the decision is due to the unwillingness to take the risk.

In many cases, production data can be modified to hide proprietary or user data. If algorithms behave similarly on the original and the modified instances, these instances can serve as proxies for production instances. For example, while employed by Microsoft and working on routing algorithms for road networks [20], we observed that turn penalties have a significant effect on algorithm performance, and different algorithms are affected by substantially different amounts. We used proprietary data that we could not share. However, we showed that adding a one-minute U-turn penalty to publicly available road network data reproduces the phenomena observed in the proprietary data. This allows other researchers to replicate our results and study the effect of turn penalties.

Data availability makes academic research more relevant to real-world applications. It also allows industry to easily evaluate academic results and build on the community’s work, since researchers will develop algorithmic techniques for problems that matter to industry. Making production data publicly available therefore benefits both industry and academia.

Generalizations and Special Cases. Given a real-life problem instance, one can add, remove, or relax some constraints to obtain new instances. For example, one may study how a constraint affects the problem complexity.

Simplified instances may have a different complexity, and one needs to justify their use. Academic studies sometimes focus on a clean version of the problem, ignoring “ugly” or unavailable real-life constraints. Studying simplified problems can lead to insights into the original problem, but extending these insights to real problems and verifying their effectiveness may require significant work.

Derivative instances. Derivative instances modify data from one domain to obtain a problem in another domain. For example, one can take a graph of social network connections, add arc lengths and a source, and obtain a shortest path problem instance. If such instances give new insights into algorithm performance, their use is justified even though they are not real-life instances. Having the original instances available, and understanding their structure, makes the derivative instances easier to use.

Domain expertise. Maintaining and updating benchmark instances relevant to evolving applications requires domain expertise. An expert knows the current problem sizes and the appropriate modeling assumptions. An assumption that seems reasonable to a non-expert may not hold in practice. For evolving applications, benchmarks need to be kept up to date using expert knowledge.

4.2 Synthetic Instances

Synthetic instances can be used to supplement the real ones. In particular, they allow control over problem parameters, which is useful for studying parameter dependence. For example, we can increase the number of vertices in the graph while keeping the degree constant. We can also

adjust several parameters simultaneously; for instance, we can increase the number of vertices n while setting the degree to $\sqrt{n}$.

The choice of parameters to investigate is guided by an understanding of the algorithms for the problem. For example, the analysis of the Bellman-Ford shortest path algorithm [10, 43, 50] shows that the number of passes of the algorithm is bounded by the depth of the shortest path tree, and each pass takes linear time. This suggests that the tree depth is an interesting parameter.

One of the synthetic instance generators used in [13] is a grid generator. A grid graph has Y layers L_i of size X , and a source vertex s . The source is connected to every vertex in the first layer. Vertex i in layer $j < Y$ is connected to vertex i in layer $j + 1$. The vertices of each layer form a bidirectional cycle. Arc lengths are chosen independently and uniformly at random from the interval $[0, 10000]$. Note that the expected depth of the shortest path tree grows with Y . As the graph size grows, fixing Y gives wide grids, and fixing X gives long grids. The expected tree depth is small in wide grids and large in long grids. In Section 7, we show that these problem families yield interesting results.

When Dijkstra’s algorithm [21] is implemented with certain data structures, such as k -heaps [64], its time bound depends on the heap size during execution. The grid problem families are also useful for investigating this dependence. For wide grids, the heap is large during most of the execution; for long grids, it is small. We discuss this further in Section 7.

Other interesting classes of test instances are what we call *algorithm separators*: classes on which competing algorithms perform very differently. For example, [13] shows that acyclic graphs with negative arc lengths separate the Bellman-Ford algorithm from the algorithms of Pape [52] and Pallottino [51]. Note that acyclic graphs with negative arc lengths arise naturally in PERT [62, 63] applications.

Problem transformations are another source of test instances. Even simple transformations can have a significant effect on performance. For example, a common assumption for path-finding algorithms is that every vertex is reachable from the source. For shortest path problems, one can ensure this by adding a new *artificial* source vertex and connecting it to all original vertices by arcs. The arc to the original source has length zero, and the other new arcs have a large length: the sum of all original positive arc lengths. Although the two problems are equivalent, we will see that the transformation affects different algorithms differently.

Algorithm expertise. Determining which synthetic problems may give significant insight into algorithm performance requires a good understanding of the problem and the algorithms for it. Take, for example, the problem of factoring a product of two primes. This problem is believed to be computationally hard, and this assumption underlies the security of the RSA encryption scheme [53]. Now consider a problem family consisting of products of two *consecutive* primes. A non-expert may think that this problem is also hard. However, someone familiar with number theory and primality testing algorithms can easily see that the problem is solvable in polynomial time under standard number-theoretic conjectures.

5 New and Competing Codes

5.1 Competing Implementations

When prior experimental work on the problem exists, the authors should identify the best available codes and include them in their study, using the best parameter settings for the benchmark set. If no single previous code dominates, the authors should include all codes that are not dominated by another existing code.

Previous codes may be available in one of two forms. For a black-box code, only the executable is available. We can time it and examine its output, which may or may not include additional performance data. Alternatively, the source code may be available. In that case, we can add instrumentation for performance analysis (e.g., operation counts).

Some experimental work is done in industry using proprietary code that is not publicly available. Authors publishing such work should include results on a good publicly available benchmark dataset, along with machine calibration experiments. This allows follow-up studies to make indirect comparisons with the proprietary code.

5.2 New Algorithm Implementation

For many experimental studies, new algorithm implementations need to be optimized. This is especially important if the goal is a state-of-the-art code and technology transfer to a user community—the gold standard of successful algorithm research. In some sense, an algorithm is not fully specified without its implementation details. Poor implementations may lead to wrong conclusions about the performance of the underlying algorithm.

Hooker [35] claims that careful implementation “diverts time and resources from productive investigation.” One can sometimes rule out algorithm variants without implementing them carefully, for example by using operation counts [48]. However, optimizing sophisticated algorithms requires algorithm engineering skills, which are orthogonal to software engineering skills. Working with several product teams in different companies, the author has seen slowdowns of one to two orders of magnitude when software engineers productized algorithm prototypes, even when the prototype had been implemented in the same environment. Regaining the lost performance required a deep dive into the code by the prototype’s creators.

Modern languages come with extensive libraries implementing various primitives. Analyzing experimental results may require a good understanding of library implementation details, since library implementations can create performance bottlenecks or cause unexpected behavior. Another reason to limit library use is reproducibility: a change in a library’s implementation may change the algorithm’s performance.

6 Reducing Dependence on Implementation and Architecture Details

Algorithm performance depends on implementation details and computer architecture. Machine and implementation-independent performance measures help to separate algorithm behavior from these factors, which helps to analyse the data.

6.1 Operation Counts

Operation counts are one way to abstract away from implementation details. If one can identify operations whose count strongly correlates with algorithm performance, then these counts should be used to supplement running times.

Consider, for example, the generic scanning method for the single-source shortest path problem [60]. For algorithms that use a simple queue, stack, or deque to order the operations, vertex scans are the bottleneck, and the scan count is a very good measure of performance. Note that if one uses some other algorithm as a subroutine to find the shortest path tree and then scans vertices in topological order with respect to the tree, each vertex is scanned at most once. However, in this case the subroutine is the bottleneck, and the scan count is not a good performance measure.

6.2 The Use of Lower Bounds

Sometimes one can use upper or lower bounds to gauge the quality of algorithms. For example, for a non-exact minimization algorithm, one can compare its result with a lower bound to bound how far the value is from the optimum. If the optimal value is available, one can use it as the lower bound. For hard problems with no known optimal solution, one can use the optimal value of a relaxed problem. An example is the Held-Karp lower bound [32, 33] for the traveling salesman problem.

One can also use lower bounds on the running time. For example, the paper [1] studies distance oracles for road networks. Given a pair of vertices in the network, a distance oracle query returns the distance between them. One can argue that the time of a lookup in a precomputed table of all pairwise distances is a lower bound on the running time of any such algorithm. The fastest algorithm studied in [1] answers queries, on average, within a factor of about 5 of the table lookup time.

7 Analyzing the Results

In order to understand and explain the data, one needs to analyze the results. In this section, we discuss some of the common approaches to the analysis.

7.1 Regression Analysis

To predict how algorithm performance scales with the problem size, one can use regression analysis [49] to fit an algorithm running time on a problem family with a function. For example, Ahuja et al. [3] conclude that on several problem families, an implementation of the push-relabel algorithm [28] runs in $O(n^{1.5})$ time, where n is the number of vertices in the input graph. Regression analysis is useful, but one has to be aware of the potential pitfalls.

To predict how algorithm performance scales with problem size, one can use regression analysis [49] to fit a function to an algorithm's running time on a problem family. For example, Ahuja et al. [3] conclude that on several problem families, an implementation of the push-relabel algorithm [28] runs in $O(n^{1.5})$ time, where n is the number of vertices in the input graph. Regression analysis is useful, but one has to be aware of its potential pitfalls.

When evaluating polynomial-time algorithms, authors usually fit a simple function, e.g., an^b . The implicit assumption is that, over the range of problem sizes in the study, the highest-order term dominates. However, this assumption often remains unverified. If the resulting growth rate is lower than the worst-case one, it is possible that a lower-order term dominates for the problem sizes studied. To check whether this is the case, one needs to run the code on problems much larger than those used for the regression analysis. This may be impractical if larger instances cannot be solved in reasonable time. However, one can still use the regression to predict running times for the problem sizes that can be solved. If solving larger problems becomes feasible, one can redo the regression analysis.

Another issue is that architecture and implementation details can affect performance differently for different problem sizes. For example, small problems may fit in cache, while the memory footprint of large problems may greatly exceed the cache size. This can make the same operation much faster on a small problem than on a large one. Therefore, small instances should be excluded from the regression. If a machine-independent operation count closely correlates with the algorithm’s running time, one can fit the operation count instead of the running time. Then, to make practical running time predictions, one needs a way to translate operation counts into running times.

7.2 Conjecture Verification

After running experiments, one needs to check whether the conjecture has been verified. For a well-chosen conjecture, at least one of the outcomes has to be interesting a priori. We can always state the conjecture so that the positive outcome is interesting. The negative outcome may or may not be interesting. An example of an interesting negative outcome involves the ellipsoid method [41], the first polynomial-time algorithm for linear programming. The mathematical programming community expected it to outperform the simplex method [19]. Although this conjecture was disproved [11, 30], the outcome brought into the mainstream the fact that a better worst-case bound does not necessarily imply better practical performance.

The motivation for our study of shortest path algorithms [13] came from multiple papers claiming that several heuristics with poor theoretical time bounds (e.g., those of Pape [52] and Pallottino [51]) outperform theoretically more efficient algorithms. We conjectured that these heuristics were not robust, in particular compared to the Goldberg-Radzick algorithm [27]. We also conjectured that for non-negative arc lengths, a proper implementation of Dijkstra’s algorithm is more robust than these heuristics.

We confirmed the conjecture that Pape’s [52] and Pallottino’s [51] heuristics are not robust. For example, these heuristics perform poorly on acyclic graphs with negative arc lengths. Further evidence that these algorithms are not robust came from experiments in which an artificial source was added. This affected the performance of Pape’s and Pallottino’s heuristics much more than that of Dijkstra’s algorithm. These heuristics rely on the order in which vertices are discovered, and adding the artificial source changes that order. On grid graphs, their performance changes from linear to highly superlinear. The two heuristics were the fastest on the original grid problems and the slowest on the modified ones.

Next, we discuss the performance of Dijkstra’s algorithm. As we mentioned in Section 4, the benchmarks used in [13] include wide and long grids. Results for these families support the conjecture that the running time of the Bellman-Ford algorithm is strongly correlated with the

depth of the shortest path tree, and the running time of Dijkstra’s algorithm is correlated with the heap size. For wide grids, the expected tree depth is constant, and Bellman-Ford runs in expected linear time. The heap size on wide grids grows asymptotically, so Dijkstra’s algorithm runs in superlinear time and is asymptotically slower than Bellman-Ford, both in theory and in practice. For long grids, the tree depth is linear, and Bellman-Ford runs in superlinear time. The expected heap size is small, and Dijkstra’s algorithm asymptotically outperforms Bellman-Ford.

In search of a robust implementation of Dijkstra’s algorithm, [13] compares several implementations and shows that the R -heap and two-level bucket implementations are more robust than the k -heap implementation. The data structure operation bounds for the first two implementations depend on $|C|$, the largest absolute value of an arc length, but not on n . The two-level bucket implementation was, overall, the best code for problems with non-negative arc lengths. While the running time of the k -heap implementation is superlinear on many problem families, the bucket-based implementations exhibit near-linear performance. This motivated a subsequent study [14], which developed a bucket-based data structure with improved theoretical bounds and even more robust practical performance. A later theoretical result [26] shows that a variant of the multi-level bucket algorithm runs in expected linear time if arc lengths are drawn independently from a natural probability distribution.

We also compared the performance of Dijkstra’s algorithm on grid graphs with and without an artificial source. The conjecture was that adding the artificial source would slow down the k -heap implementation. This is because the initial scan of the artificial source adds every original vertex to the heap, which then contains all unscanned vertices at all times, leading to the worst-case heap size. Experiments confirmed the conjecture. On the original long grids, Dijkstra’s algorithm runs in linear time; adding the artificial source results in superlinear behavior.

7.3 Real-Life Bottlenecks

Most real-life instances are much easier than worst-case analysis suggests. An experiment should attempt to determine the real bottleneck of an implementation. Sometimes this bottleneck differs from the worst-case bottleneck. In such cases, one may improve practical performance by focusing on the real bottleneck.

For example, non-saturating pushes are the theoretical bottleneck of the push-relabel method for the maximum flow problem [29]. The dynamic tree data structure [58] improves the worst-case bound by performing multiple non-saturating pushes in a single tree operation. In practice, however, the number of non-saturating pushes is of the same order of magnitude as the number of saturating pushes, so they are not the real bottleneck. Moreover, a typical dynamic tree operation performs too few pushes to offset its overhead. As a result, dynamic trees hurt the algorithm’s practical performance [5].

7.4 Implicit Assumption Violations

When analyzing the data, one needs to check whether the results align with current folklore beliefs, which are usually implicitly assumed to be true. If there is a conflict, one has identified a violation of an implicit conjecture. Such findings improve the understanding of the problem.

Given a weighted graph and a source vertex s , the classical shortest path problem is to find either shortest paths from s to all other vertices or a negative-length cycle reachable from s . Until the mid-1990s, most experimental studies focused on instances with no negative cycles. While

some algorithms, e.g., [18, 27], have a built-in cycle-detection mechanism, the classical Bellman-Ford algorithm [10, 43, 50] needs to be augmented to find cycles. Multiple experimental studies on instances with no negative cycles concluded that Bellman-Ford performs poorly. Because augmenting the algorithm requires it to perform additional work, the folklore belief was that such augmentations only slow the algorithm down.

In [12], we conducted a study of negative cycle detection algorithms. Based on the folklore and the fact that Bellman-Ford performs poorly in the absence of negative cycles, we conjectured that the algorithm would also perform poorly in the context of negative cycle detection. In particular, we expected it to perform much worse than the Goldberg-Radzik algorithm. Despite this expectation, we included in our study all cycle-detection strategies for the Bellman-Ford algorithm that we were aware of, because no experimental study had confirmed their poor performance. All but one of these strategies performed poorly. However, Tarjan’s subtree-disassembly augmentation [61] greatly improved the practical performance of Bellman-Ford, even on instances with no negative cycles. On many problem families, the improvement was asymptotic, and the modified algorithm was competitive with the Goldberg-Radzik algorithm. This disproved our conjecture about the poor performance of the Bellman-Ford algorithm.

Our analysis explained the performance improvement. The variant of the Bellman-Ford algorithm we implemented [60] uses a FIFO queue to maintain the vertices that need to be scanned. During subtree disassembly, some vertices are removed from the queue. The removed vertices have the property that their current distance estimate is *incorrect*: it is greater than the true distance. Scanning a vertex with an incorrect distance estimate is wasteful, because the vertex will be scanned again once its estimate improves. Furthermore, scanning such a vertex may cause its neighbors to be added to the queue with incorrect estimates. Therefore, removing a vertex with an incorrect distance estimate can eliminate many wasteful scans. Scan counts confirmed that the augmented algorithm performs fewer scans.

When we worked on [12], the Bellman-Ford algorithm had been known for over four decades and was standard textbook material. The subtree-disassembly technique had been known for almost two decades but was considered to be of purely theoretical interest. Our work showed that the subtree-disassembly version of this classical algorithm is the one to implement and to teach, due to its improved performance and integrated negative cycle detection.

8 Example: Road Network Routing

The problem of finding shortest paths in road networks (*road network routing*) is a special case of the shortest path problem, motivated by navigation systems. Since road networks do not change often, in this application we can preprocess them to facilitate fast queries. This problem has received considerable attention; see [8] for a survey. Extensive research has reduced query times from seconds to microseconds while maintaining realistic preprocessing times. The work on this problem is a good example of the scientific method in action, with multiple contributions made by several groups of scientists over a long period of time. Initial experimental work led to increasingly fast algorithms and implementations. These algorithms take advantage of the hierarchical structure of road networks [31, 54, 7, 25] or the fact that road networks partition well [46, 20]. Some of the algorithms were adopted by industry. Then came theoretical work that explained many of the experimental observations. These theoretical results have since been used to develop efficient algorithms for other problems on road networks.

Another approach is based on the *hub labeling (HL)* algorithm of Cohen et al. [15]. Abraham et al. [1] gave an efficient implementation of HL for road networks. Subsequent work sped up the algorithm [37, 59] and produced theoretical results on the complexity of the underlying problem [45]. The work on efficient hub labeling algorithms had an impact beyond road network routing: later research showed that the algorithm also works well on “complex” networks. This line of research, which started with [4], includes multiple papers; see [44] for a survey.

While fast algorithms for road network routing had a major practical impact, there was no theoretical analysis of these algorithms, because there was no mathematical model of road networks. Abraham et al. [2] developed such a model and introduced the notion of *highway dimension (h)*. They conjectured that h is small for road networks. For the hub labeling [15], contraction hierarchies [25], and transit node routing [7] algorithms, they obtained query time bounds that are linear in h and logarithmic in other network parameters. Follow-up research uncovered a relationship to VC-dimension [38] and gave an alternative model of road networks [42]. Although it was developed in the context of road network routing, highway dimension has been used to design theoretically efficient algorithms for other problems on road networks, such as the traveling salesman, Steiner tree, vehicle routing, k -means, and facility location problems [9, 22, 23, 39]. This is an example of experimental results leading to a better understanding of a problem, which in turn leads to new theoretical results.

9 Concluding Remarks

Many modern algorithms and applications are quite sophisticated, and high-quality experimental studies may require significant resources. We discussed how making these studies part of the scientific method loop leads to better and more practical algorithms. We would like to highlight the following points:

- An experimental study should explicitly state the conjectures that it aims to verify.
- The experiments should be designed to produce data that may give insight into the problem.
- The resulting data should be used in subsequent theoretical analysis.
- Real-life instances should be used to keep research connected to practice.

The current rapid advances in AI can help both algorithm engineering and scientific evaluation. AI assistants can help with coding (e.g., [6]). Machine learning can automate parameter tuning (e.g., [36, 57]) and algorithm selection (e.g., [40]). Machine learning and AI agents are also being used to aid algorithm design (e.g., [56, 65]). From the late 1950s to the late 1970s, advances in compiler optimization led high-level programming languages to largely replace assembly language. Similarly, advances in AI may eliminate the need to code algorithms manually. An AI-based system would take a sufficiently detailed textual description of a new algorithm, along with previous implementations, and produce an efficient implementation. A companion system could then conduct an experimental evaluation of the new algorithm. This would greatly accelerate the algorithm design cycle.

References

- [1] Ittai Abraham et al. “A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks”. In: *Experimental Algorithms - 10th International Symposium, SEA 2011, Chania, Crete, Greece, May 5-7, 2011. Proceedings*. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 230–241. DOI: 10.1007/978-3-642-20662-7_20. URL: <https://doi.org>.
- [2] Ittai Abraham et al. “Highway Dimension and Provably Efficient Shortest Path Algorithms”. In: *Journal of the ACM* 63.5 (2016), p. 41. DOI: 10.1145/2985473.
- [3] Ravindra K. Ahuja et al. “Computational Investigations of Maximum Flow Algorithms”. In: *European Journal of Operational Research* 97.3 (1997), pp. 509–542. DOI: 10.1016/S0377-2217(96)00269-X.
- [4] T. Akiba, Y. Iwata, and Y. Yoshida. “Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD’13. ACM, 2013, pp. 349–360.
- [5] Tamás Badics and Endre Boros. “Implementing a maximum flow algorithm: experiments with dynamic trees”. In: *Network Flows and Matching: First DIMACS Implementation Challenge*. American Mathematical Society, 1993, pp. 43–63.
- [6] Shraddha Barke, Michael B. James, and Nadia Polikarpova. “Grounded Copilot: How Programmers Interact with Code-Generating Models”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (Apr. 2023), pp. 85–111. DOI: 10.1145/3586030.
- [7] H. Bast et al. “In Transit to Constant Time Shortest-Path Queries in Road Networks”. In: *Proc. 9th International Workshop on Algorithm Engineering and Experiments*. Available at <http://www.mpi-inf.mpg.de/bast/tmp/transit.pdf>. SIAM, 2006, pp. 46–59.
- [8] Hannah Bast et al. “Route Planning in Transportation Networks”. In: *Algorithm Engineering*. Lecture Notes in Computer Science 9220 (2016), pp. 19–80. DOI: 10.1007/978-3-319-49487-6_2.
- [9] Amariah Becker, Philip N. Klein, and David Saulpic. “Polynomial-Time Approximation Schemes for k -center, k -median, and Capacitated Vehicle Routing in Bounded Highway Dimension”. In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Vol. 112. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 8:1–8:15. DOI: 10.4230/LIPIcs.ESA.2018.8.
- [10] R. E. Bellman. “On a Routing Problem”. In: *Quart. Appl. Math.* 16 (1958), pp. 87–90.
- [11] Robert G. Bland, Donald Goldfarb, and Michael J. Todd. “The Ellipsoid Method: A Survey”. In: *Operations Research* 29.6 (1981), pp. 1039–1091. DOI: 10.1287/opre.29.6.1039.
- [12] B. V. Cherkassky and A. V. Goldberg. “Negative-Cycle Detection Algorithms”. In: *Math. Prog.* 85 (1999), pp. 277–311.
- [13] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. “Shortest Paths Algorithms: Theory and Experimental Evaluation”. In: *Math. Prog.* 73 (1996), pp. 129–174.
- [14] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. “Buckets, Heaps, Lists, and Monotone Priority Queues”. In: *SIAM J. Comput.* 28 (1999), pp. 1326–1346.

- [15] E. Cohen et al. “Reachability and Distance Queries via 2-hop Labels”. In: *SIAM Journal on Computing* 32 (2003).
- [16] Timothy Colburn. “Methodology of Computer Science”. In: *The Blackwell Guide to the Philosophy of Computing and Information*. Ed. by Luciano Floridi. Malden, MA: Blackwell, 2004, pp. 318–326. ISBN: 978-0-631-22942-1.
- [17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [18] G. B. Dantzig. “Application of the Simplex Method to a Transportation Problem”. In: *Activity Analysis and Production and Allocation*. Ed. by T. C. Koopmans. New York: Wiley, 1951, pp. 359–373.
- [19] George B. Dantzig. *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press, 1963.
- [20] Daniel Delling et al. “Customizable Route Planning in Road Networks”. In: *Transportation Science* 51.2 (2015), pp. 566–591. DOI: 10.1287/trsc.2014.0579.
- [21] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. In: *Numer. Math.* 1 (1959), pp. 269–271.
- [22] Yann Disser et al. “Travelling on Graphs with Small Highway Dimension”. In: *Algorithmica* 83.5 (2021), pp. 1352–1370. DOI: 10.1007/s00453-020-00785-5.
- [23] Andreas Emil Feldmann and David Saulpic. “Polynomial Time Approximation Schemes for Clustering in Low Highway Dimension Graphs”. In: *Journal of Computer and System Sciences* 122 (2021), pp. 72–93. DOI: 10.1016/j.jcss.2021.06.002.
- [24] J.H. Ramser G.B. Dantzig. “The Truck Dispatching Problem”. In: *Management Science* 6.1 (1959), pp. 80–91.
- [25] R. Geisberger et al. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *WEA*. 2008, pp. 319–333.
- [26] A. V. Goldberg. “A Practical Shortest Path Algorithm with Linear Expected Time”. In: *SIAM Journal on Computing* 37 (2008), pp. 1637–1655.
- [27] A. V. Goldberg and T. Radzik. “A Heuristic Improvement of the Bellman-Ford Algorithm”. In: *Applied Math. Let.* 6 (1993), pp. 3–6.
- [28] A. V. Goldberg and R. E. Tarjan. “A New Approach to the Maximum Flow Problem”. In: *J. Assoc. Comput. Mach.* 35 (1988), pp. 921–940.
- [29] A. V. Goldberg and R. E. Tarjan. “Finding Minimum-Cost Circulations by Canceling Negative Cycles”. In: *Proc. 20th Annual ACM Symposium on Theory of Computing*. 1988, pp. 388–397.
- [30] Donald Goldfarb and Michael J. Todd. “Modifications and Implementation of the Ellipsoid Algorithm for Linear Programming”. In: *Mathematical Programming* 23.1 (1982), pp. 1–19.
- [31] R. Gutman. “Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks”. In: *Proc. 6th International Workshop on Algorithm Engineering and Experiments*. 2004, pp. 100–111.

- [32] Michael Held and Richard M. Karp. “The Traveling-Salesman Problem and Minimum Spanning Trees”. In: *Operations Research* 18.6 (1970), pp. 1138–1162. DOI: 10.1287/opre.18.6.1138.
- [33] Michael Held and Richard M. Karp. “The Traveling-Salesman Problem and Minimum Spanning Trees: Part II”. In: *Mathematical Programming* 1.1 (1971), pp. 6–25. DOI: 10.1007/BF01584070.
- [34] C. A. R. Hoare. “Mathematics of programming”. In: *BYTE* 11.8 (Aug. 1986), pp. 115–149.
- [35] John N. Hooker. “Testing Heuristics: We Have It All Wrong”. In: *Journal of Heuristics* 1.1 (1995), pp. 33–42. DOI: 10.1007/BF02430364.
- [36] Frank Hutter et al. “An Experimental Investigation of Model-Based Parameter Optimisation: The Algorithm Configuration Problem”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*. 2009, pp. 1152–1159.
- [37] I. Abraham et al. “Hierarchical Hub Labelings for Shortest Paths”. In: *Proc. 20th European Symposium on Algorithms (ESA 2012)*. Springer Verlag, 2012.
- [38] I. Abraham et al. “VC-Dimension and Shortest Path Algorithms”. In: *Proceedings of the 38th International Colloquium on Automata, Languages, and Programming (ICALP’11)*. Vol. 6755. Lecture Notes in Computer Science. Springer, 2011, pp. 690–699.
- [39] Aditya Jayaprakash and Mohammad R. Salavatipour. “Approximation Schemes for Capacitated Vehicle Routing on Graphs of Bounded Treewidth, Bounded Doubling, or Highway Dimension”. In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2022, pp. 877–893. DOI: 10.1137/1.9781611977073.37.
- [40] Pascal Kerschke et al. “Automated Algorithm Selection: Survey and Perspectives”. In: *Evolutionary Computation* 27.1 (2019), pp. 3–45. DOI: 10.1162/evco_a_00242.
- [41] L. G. Khachiyan. “A Polynomial Algorithm in Linear Programming”. In: *Doklady Akademii Nauk SSSR* 244.5 (1979), pp. 1093–1096.
- [42] Adrian Kosowski and Laurent Viennot. “Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2017, pp. 1462–1478. DOI: 10.1137/1.9781611974782.95.
- [43] Jr. L. R. Ford. *Network Flow Theory*. Tech. rep. P-932. The Rand Corporation, 1956.
- [44] Ye Li et al. “An Experimental Study on Hub Labeling Based Shortest Path Algorithms”. In: *Proceedings of the VLDB Endowment* 11.4 (2018), pp. 445–457. DOI: 10.1145/3164135.3164141.
- [45] M. Babenko et al. “Algorithms for Hub Label Optimization”. In: *Proc. 30th ICALP*. Ed. by F.V. Fomin et al. LNCS vol. 7965, Springer, 2013, pp. 69–80.
- [46] Rolf H. Möhring et al. “Partitioning Graphs to Speedup Dijkstra’s Algorithm”. In: *ACM Journal of Experimental Algorithmics* 11 (2006), p. 2.8. DOI: 10.1145/1187436.1216585.
- [47] C. McGeoch. “Towards an Experimental Method for Algorithm Simulation”. In: *INFORMS J. Comp.* 8 (1996), pp. 1–15.

- [48] Catherine C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012. ISBN: 978-1-107-00173-2. DOI: 10.1017/CB09780511843747.
- [49] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. *Introduction to Linear Regression Analysis*. 5th ed. John Wiley & Sons, 2012. ISBN: 978-0-470-54281-1.
- [50] E. F. Moore. “The Shortest Path Through a Maze”. In: *Proc. of the Int. Symp. on the Theory of Switching*. Harvard University Press, 1959, pp. 285–292.
- [51] S. Pallottino. “Shortest-Path Methods: Complexity, Interrelations and New Propositions”. In: *Networks* 14 (1984), pp. 257–267.
- [52] U. Pape. “Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem”. In: *Math. Prog.* 7 (1974), pp. 212–222.
- [53] R. L. Rivest, A. Shamir, and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342.
- [54] P. Sanders and D. Schultes. “Highway Hierarchies Hasten Exact Shortest Path Queries”. In: *Proc. 13th Annual European Symposium Algorithms*. 2005, pp. 568–579.
- [55] Peter Sanders. “Algorithm Engineering – An Attempt at a Definition Using Sorting as an Example”. In: *2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2010, pp. 55–61. DOI: 10.1137/1.9781611972900.6.
- [56] Christian Schulz. *Agentic Algorithm Engineering*. Version 1.0. 2026. URL: <https://github.com/CHSZLab/AgenticAlgorithmEngineering>.
- [57] Bobak Shahriari et al. “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.
- [58] D. D. Sleator and R. E. Tarjan. “A Data Structure for Dynamic Trees”. In: *J. Comput. System Sci.* 26 (1983), pp. 362–391.
- [59] T. Akiba et al. “Fast Shortest path Distance Queries on Road Networks by Pruned Highway Labeling”. In: *Proc. 16th International Workshop on Algorithm Engineering and Experiments*. 2014.
- [60] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
- [61] R. E. Tarjan. *Shortest Paths*. Tech. rep. AT&T Bell Laboratories, Murray Hill, NJ, 1981.
- [62] U.S. Navy Special Projects Office. *PERT (Program Evaluation Research Task): Summary Report, Phase 1*. Technical Report. Washington, DC: U.S. Department of the Navy, Bureau of Naval Weapons, Special Projects Office, 1958.
- [63] U.S. Navy Special Projects Office. *PERT (Program Evaluation Research Task): Summary Report, Phase 2*. Technical Report. Washington, DC: U.S. Department of the Navy, Bureau of Naval Weapons, Special Projects Office, 1958.
- [64] J. W. J. Williams. “Algorithm 232 (Heapsort)”. In: *Comm. ACM* 7 (1964), pp. 347–348.

- [65] He Yu and Jing Liu. “Exploring Automated Algorithm Design Synergizing Large Language Models and Evolutionary Algorithms: Survey and Insights”. In: *Evolutionary Computation* 34.2 (2026), pp. 167–194. doi: 10.1162/EVC0.a.370.