The Mathematics of Algorithm Design

Jon Kleinberg
Cornell University, Ithaca NY USA.

1 The Goals of Algorithm Design

When computer science began to emerge as a subject at universities in the 1960s and 1970s, it drew some amount of puzzlement from the practitioners of more established fields. Indeed, it is not initially clear why computer science should be viewed as a distinct academic discipline. The world abounds with novel technologies, but we don't generally create a separate field around each one; rather, we tend to view them as by-products of existing branches of science and engineering. What is special about computers?

Viewed in retrospect, such debates highlighted an important issue: computer science is not so much about the computer as a specific piece of technology as it is about the more general phenomenon of computation itself, the design of processes that represent and manipulate information. Such processes turn out to obey their own inherent laws, and they are performed not only by computers but by people, by organizations, and by systems that arise in nature. We will refer to these computational processes as *algorithms*. For the purposes of our discussion in this article, one can think of an algorithm informally as a step-by-step sequence of instructions, expressed in a stylized language, for solving a problem.

This view of algorithms is general enough to capture both the way a computer processes data and the way a person performs calculations by hand. For example, the rules for adding and multiplying numbers that we learn as children are algorithms; the rules used by an airline company for scheduling flights constitute an algorithm; and the rules used by a search engine like Google for ranking Web pages constitute an algorithm. It is also fair to say that the rules used by the human brain to identify objects in the visual field constitute a kind of algorithm, though we are currently a long way from understanding what this algorithm looks like or how it is implemented on our neural hardware.

A common theme here is that one can reason about all these algorithms without recourse to specific computing devices or computer programming languages, instead expressing them using the language of mathematics. In fact, the notion of an algorithm as we now think of it was formalized in large part by the work of mathematical logicians in the 1930s, and algorithmic reasoning is implicit in the past several millenia of mathematical activity. (For example, equation-solving methods have always tended to have a strong algorithmic flavor; the geometric constructions of the ancient Greeks were inherently algorithmic as well.) Today, the mathematical analysis of algorithms occupies a central position in computer science; reasoning about algorithms independently of the specific devices on which they run can yield insight into general design principles and fundamental constraints on computation.

At the same time, computer science research struggles to keep two diverging views in focus: this more abstract view that formulates algorithms mathematically, and the more applied view that the public generally associates with the field, the one that seeks to develop applications such as Internet search engines, electronic banking systems, medical imaging software, and the host of other creations we have come to expect from computer technology. The tension between these two views means that the field's mathematical formulations are continually being tested against their implementation in practice; it provides novel avenues for mathematical notions to influence widely used applications; and it sometimes leads to new mathematical problems motivated by these applications.

The goal of this short article is to illustrate this balance between the mathematical formalism and the motivating applications of computing. We begin by building up to one of the most basic definitional questions in this vein: how should we formulate the notion of *efficient* computation?

2 Two Representative Problems

To make the discussion of efficiency more concrete, and to illustrate how one might think about an issue like this, we first discuss two representative problems — both fundamental in the study of algorithms — that are similar in their formulation but very different in their computational difficulty.

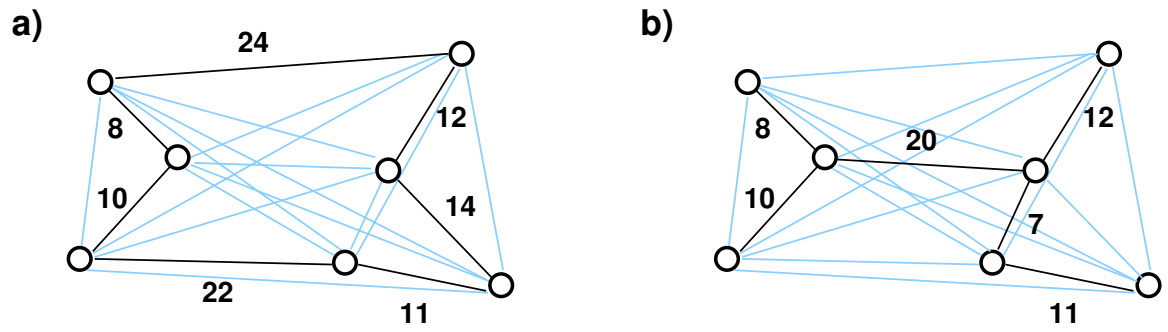

Figure 1: Solutions to instance of (a) the Traveling Salesman Problem and (b) the Minimum Spanning Tree Problem, on the same set of points. The dark lines indicate the pairs of cities that are connected by the respective optimal solutions, and the lighter lines indicate all pairs that are not connected.

The first in this pair is the *Traveling Salesman Problem* (TSP), and it is defined as follows. We imagine a salesman contemplating a map with n cities (he is currently located in one of them). The map gives the distance between each pair of cities, and the salesman wishes to plan the shortest possible tour that visits all n cities and returns to the starting point. In other words, we are seeking an algorithm that takes as input the set of all distances among pairs of cities, and produces a tour of minimum total length. Figure 1(a) depicts the optimal solution to a small instance of the TSP; the circles represent the cities, the dark line segments (with lengths labeling them) connect cities that the salesman visits consecutively on the tour, and the light line segments connect all the other pairs of cities, which are not visited consecutively.

A second problem is the *Minimum Spanning Tree Problem* (MST). Here we imagine a construction firm with access to the same map of n cities, but with a different goal in mind. They wish to build a set of roads connecting certain pairs of the cities on the map, so that after these roads are built there is a route from each of the n cities to each other. (A key point here is that each road must go directly from one city to another.) Their goal is to build such a road network as cheaply as possible — in other words, using as little total road material as possible. Figure 1(b) depicts the optimal solution to the instance of the MST defined by the same set of cities used for part (a).

Both of these problems have a wide range of practical applications. The TSP is a basic problem concerned with sequencing a given set of objects in a "good" order; it has been used for problems that run from planning the motion of robotic arms drilling holes on printed circuit boards (where the "cities" are the locations where the holes must be drilled) to ordering genetic markers on a chromosome in a linear sequence (with the markers constituting the cities, and the distances derived from probabilistic estimates of proximity). The MST is a basic issue in the design of efficient communication networks; this follows the motivation given above, with fiber-optic cable acting in the role of "roads." The MST also plays an important role in the problem of clustering data into natural groupings. Note for example how the points on the left side of Figure 1(b) are joined to the points on the right side by a relatively long link; in clustering applications, this can be taken as evidence that the left and right points form natural groupings.

It is not hard to come up with an algorithm for solving the TSP. We first list every possible way of ordering the cities (other than the starting city, which is fixed in advance). Each ordering defines a tour – the salesman could visit the cities in this order and then return to the start – and for each ordering we could compute the total length of the tour, by traversing the cities in this order and summing the distances from each city to the next. As we perform this calculation for all possible orders, we keep track of the order that yields the smallest total distance, and at the end of the process we return this tour as the optimal solution.

While this algorithm does solve the problem, it is extremely inefficient. There are $n-1$ cities other than the starting point, and any possible sequence

of them defines a tour, so we need to consider $(n-1)(n-2)(n-3)\cdots(3)(2)(1) = (n-1)!$ possible tours. Even for $n = 30$ cities, this is an astronomically large quantity; on the fastest computers we have today, running this algorithm to completion would take longer than the life expectancy of the Earth. The difficulty is that the algorithm we have just described is performing *brute-force search*: the "search space" of possible solutions to the TSP is very large, and the algorithm is doing nothing more than plowing its way through this entire space, considering every possible solution.

For most problems, there is a comparably inefficient algorithm that simply performs brute-force search. Things tend to get interesting when one finds a way to improve significantly over this brute-force approach.

The Minimum Spanning Tree Problem provides a nice example of how such an improvement can happen. Rather than considering all possible road networks on the given set of cities, suppose we try the following myopic, "greedy" approach to the MST. We sort all the pairs of cities in order of increasing distance, and then work through the pairs in this order. When we get to a pair of cities, say A and B, we test if there is already a way to travel from A to B in the collection of roads constructed thus far. If there is, then it would be superfluous to build a direct road from A to B — our goal, remember, is just to make sure every pair is connected by *some* sequence of roads, and A and B are already connected in this case. But if there is no way to get from A to B using what's already been built, then we construct the direct road from A to B. (As an example of this reasoning, note that the potential road of length 14 in Figure 1(a) would not get built by this MST algorithm; by the time this direct route is considered, its endpoints are already joined by the sequence of two shorter roads of length 7 and 11 as depicted in Figure 1(b).)

It is not at all obvious that the resulting road network should have the minimum possible cost, but in fact this is true. In other words, one can prove a theorem that says, essentially, "On every input, the algorithm just described produces an optimal solution." The payoff from this theorem is that we now have a way to compute an optimal road network by an algorithm that is much, much more efficient than brute-force search: it simply needs to sort the pairs of cities by their distances, and then make a single pass through this sorted list to decide which roads to build.

This discussion has provided us with a fair amount of insight into the nature of the TSP and MST problems. Rather than experimenting with actual computer programs, we described algorithms in words, and made claims about their performance that could be stated and proved as mathematical theorems. But what can we abstract from these examples, if we want to talk about computational efficiency in general?

3 Computational Efficiency

Most interesting computational problems share the following feature with the TSP and the MST: an input of size n implicitly defines a search space of possible solutions whose size grows exponentially with n. One can appreciate this explosive growth rate as follows: if we simply add one to the size of the input, the time required to search the entire space increases by a multiplicative factor. We'd prefer algorithms to scale more reasonably: their running times should only increase by a multiplicative factor when the input itself increases by a multiplicative factor. Running times that are bounded by a polynomial function of the input size — in other words, proportional to n raised to some fixed power — exhibit this property. For example, if an algorithm requires at most n^2 steps on an input of size n, then it requires at most $(2n)^2 = 4n^2$ steps on an input twice as large.

In part because of arguments like this, computer scientists in the 1960s adopted *polynomial time* as a working definition of efficiency: an algorithm is deemed to be efficient if the number of steps it requires on an input of size n grows like n raised to a fixed power. Using the concrete notion of polynomial time as a surrogate for the fuzzier concept of efficiency is the kind of modeling decision that ultimately succeeds or fails based on its utility in guiding the development of real algorithms. And in this regard, polynomial time has turned out to be a definition of surprising power in practice: problems for which one can develop a polynomial-time algorithm have turned out in general to be highly tractable, while those for which we lack polynomial-time algorithms tend to pose serious

challenges even for modest input sizes.

A concrete mathematical formulation of efficiency provides a further benefit: it becomes possible to pose, in a precise way, the conjecture that certain problems cannot be solved by efficient algorithms. The Traveling Salesman Problem is a natural candidate for such a conjecture; after decades of failed attempts to find an efficient algorithm for the TSP, one would like to be able to prove a theorem that says, "There is no polynomial-time algorithm that finds an optimal solution to every instance of the TSP." A theory known as *NP-completeness* provides a unifying framework for thinking about such questions; it shows that a large class of computational problems, containing literally thousands of naturally arising problems (including the TSP), are equivalent with respect to polynomial-time solvability. There is an efficient algorithm for one if and only if there is an efficient algorithm for all. It is a major open problem to decide whether or not these problems have efficient algorithms; the deeply held sense that that they do not has become the "$P \neq NP$" conjecture, which has begun to appear on lists of the most prominent problems in mathematics.

Like any attempt to make an intuitive notion mathematically precise, polynomial time as a definition of efficiency in practice begins to break down around its boundaries. There are algorithms for which one can prove a polynomial bound on the running time, but which are hopelessly inefficient in practice. Conversely, there are well-known algorithms (such as the standard simplex method for linear programming) that require exponential running time on certain pathological instances, but which run quickly on almost all inputs encountered in real life. And for computing applications that work with massive datasets, an algorithm with a polynomial running time may not be efficient enough; if the input is a trillion bytes long (as can easily occur when dealing with snapshots of the Web, for example) even an algorithm whose running time depends quadratically on the input would be unusable in practice. For such applications, one generally needs algorithms that scale linearly in the size of the input — or, more strongly, that operate by "streaming" through the input in one or two passes, solving the problem as they go. The theory of such streaming algorithms is an active topic of research, drawing on techniques from information theory, Fourier analysis, and other areas. None of this says that polynomial time is losing its relevance to algorithm design; it is still the standard benchmark for efficiency. But new computing applications tend to push the limits of current definitions, and in the process raise new mathematical problems.

4 Algorithms for Computationally Intractable Problems

In the previous section we discussed how researchers have identified a large class of natural problems, including the TSP, for which it is strongly believed that no efficient algorithm exists. While this explains our difficulties in solving these problems optimally, it leaves open a natural question: what should we do when actually confronted by such a problem in practice?

There are a number of different strategies for approaching such computationally intractable problems. One of these is *approximation*: for problems like the TSP that involve choosing an optimal solution from among many possibilities, we could try to formulate an efficient algorithm that is guaranteed to produce a solution almost as good as the optimal one. The design of such approximation algorithms is an active area of research; we can see a basic example of this process by considering the TSP. Suppose we are given an instance of the TSP, specified by a map with distances, and we set ourselves the task of constructing a tour whose total length is at most twice that of the shortest tour. At first this goal seems a bit daunting: since we don't know how to compute the optimal tour (or its length), how will we guarantee that the solution we produce is short enough? It turns out, however, that this can be done by exploiting an interesting connection between the TSP and MST problems, a relationship between the respective optimal solutions to each problem on the same set of cities.

Consider an optimal solution to the MST problem on the given set of cities, consisting of a network of roads; recall that this is something we can compute efficiently. Now, the salesman interested in finding a short tour for these cities can use this optimal road network to visit the cities as follows. Starting at one city, he follows roads until he hits

a dead end — a city with no new roads exiting it. He then backs up, re-tracing his steps until he gets to a junction with a road he hasn't yet taken, and he proceeds down this new road. For example, starting in the upper left corner of Figure 1(b), the salesman would follow the road of length 8 and then choose one of the roads of length 10 or 20; if he selects the former, then after reaching the dead-end he would back up to this junction again and continue the tour by following the road of length 20. A tour constructed this way traverses each road twice (once in each direction), so if we let m denote the total length of all roads in the optimal MST solution, we have found a tour of length $2m$.

How does this compare to t, the length of the best possible tour? Let's argue first that $t \geq m$. This is true because in the space of all possible solutions to the MST, one option is to build roads between cities that the salesman visits consecutively in the optimal TSP tour, for a total mileage of t; on the other hand, m is the total length of the *shortest possible* road network, and hence t cannot be smaller than m. So we've concluded that the optimal solution to the TSP has length at least m; combined with the previous conclusion, that our algorithm is producing a tour of length $2m$, we have an approximation to within a factor of two of optimal, as we had wanted.

People trying to solve large instances of computationally hard problems in practice frequently use algorithms that have been observed empirically to give nearly optimal solutions, even when no guarantees on their performance have been proved. *Local search* algorithms form one widely-used class of approaches like this. A local search algorithm starts with an initial solution and repeatedly modifies it by making some "local" change to its structure, looking for a way to improve its quality. In the case of the TSP, a local search algorithm would seek simple improving modifications to its current tour; for example, it might look at sets of cities that are visited consecutively and see if visiting them in the opposite order would shorten the tour. Researchers have drawn connections between local search algorithms and phenomena in nature; for example, just as a large molecule contorts itself in space trying to find a minimum-energy conformation, we can imagine the TSP tour in a local search algorithm modifying itself as it tries to reduce its length. Determining how deeply this analogy goes is an interesting research issue.

5 Mathematics and Algorithm Design: Reciprocal Influences

Many branches of mathematics have contributed to aspects of algorithm design, and the issues raised by the analysis of new algorithmic problems have in a number of cases suggested novel mathematical questions.

Combinatorics and graph theory have been qualitatively transformed by the growth of computer science, to the extent that algorithmic questions have become thoroughly intertwined with the mainstream of research in these areas. Techniques from probability have also become fundamental to many areas of computer science: probabilistic algorithms draw power from the ability to make random choices while they are executing, and probabilistic models of the input to an algorithm allow one to try capturing more accurately the family of problem instances that arise in practice. This style of analysis provides a steady source of new questions in discrete probability.

A computational perspective is often useful in thinking about "characterization" problems in mathematics. For example, the general issue of characterizing prime numbers has an obvious algorithmic component: given a number n as input, how efficiently can we determine whether it is prime? (There exist algorithms that are exponentially better than the approach of dividing n by all numbers up to $\sqrt{n}$.) Problems in knot theory such as the characterization of unknotted loops have a similar algorithmic side. Suppose we are given a circular loop of string in three dimensions (described as a jointed chain of line segments), and it wraps around itself in complicated ways. How efficiently can we determine whether it is truly knotted, or whether by moving it around we can fully untangle it? One can engage in this activity with many similar mathematical issues; it is clear that the corresponding algorithmic versions are extremely concrete as problems, though they may lose part of the original intent of the mathematicians who posed the questions more generally.

Rather than attempting to enumerate the intersection of algorithmic ideas with all the different

branches of mathematics, we conclude this article with two case studies that involve the design of algorithms for particular applications, and the ways in which mathematical ideas arise in each instance.

6 Web Search and Eigenvectors

As the World Wide Web grew in popularity throughout the 1990s, computer science researchers grappled with a difficult problem: the Web contains a vast amount of useful information, but its anarchic structure makes it very hard for users, unassisted, to find the specific information they're looking for. Thus, early in the Web's history, people began to develop *search engines* that would index the information on the Web, and produce relevant Web pages in response to user queries. But of the thousands or millions of pages relevant to a topic on the Web, which few should the search engine present to a user? This is the *ranking* problem: how to determine the "best" resources on a given topic. Note the contrast with concrete problems like the Traveling Salesman. There, the goal (the shortest tour) was not in doubt; the difficulty was simply in computing an optimal solution efficiently. For the search engine ranking problem, on the other hand, formalizing the goal is a large part of the challenge — what do we mean by the "best" page on a topic? In other words, an algorithm to rank Web pages is really providing a *definition* of Web page quality as well as the means to evaluate this definition.

The first search engines ranked each Web page based purely on the text it contained. These approaches began to break down as the Web grew, because they didn't take into account the quality judgments encoded in the Web's hyperlinks: in browsing the Web, we often discover high-quality resources because they are "endorsed" through the links they receive from other pages. This insight led to a second generation of search engines that determined rankings using *link analysis*.

The simplest such analysis would just count the number of links to a page: in response to the query "newspapers," for example, one could rank pages by the number of incoming links they receive from other pages containing the term — in effect, allowing pages containing the term "newspapers" to vote on the result. Such a scheme will generally do well for the top few items, placing prominent news sites like *The New York Times* and *The Financial Times* at the head of the list, but will quickly become noisy after this.

It is possible to make much more effective use of the latent information in the links. Consider pages that link to many of the sites ranked highly by this simple voting scheme; it is natural to expect that these are authored by people with a good sense for where the interesting newspapers are, and so we could run the voting again, this time giving more voting power to these pages that selected many of the highly-ranked sites. This revote may elevate certain lesser-known newspapers, favored by the Web pages authors who are more knowledgeable on the topic; in response to the results of this revote, we could further sharpen our weighting of the voters. This "principle of repeated improvement" uses the information contained in a set of page quality estimates to produce a more refined set of estimates. If we perform these refinements repeatedly, will they converge to a stable solution?

In fact, this sequence of refinements can be viewed as an algorithm to compute the principal eigenvector of a particular matrix; this both establishes the convergence of the process and characterizes the end result. To establish this connection, we introduce some notation. Each Web page is assigned two scores: an *authority weight*, measuring its quality as a primary source on the topic; and a *hub weight*, measuring its power as a voter for the highest-quality content. Pages may score highly in one of these measures but not the other — one shouldn't expect a prominent newspaper to simultaneously serve as a good guide to other newspapers — but there is also nothing to prevent a page from scoring well in both. One round of voting can now be viewed as follows. We update the authority weight of each page by summing the hub weights of all pages that point to it (receiving links from highly-weighted voters makes you a better authority); we then re-weight all the voters, updating each page's hub weight by summing the authority weights of the pages it points to (linking to high-quality content makes you a better hub).

How do eigenvectors come into this? Suppose we define a matrix M with one row and one column for each page under consideration; the (i, j) entry equals 1 if page i links to page j, and

it equals 0 otherwise. We encode the authority weights in a vector a, where the coordinate a_i is the authority weight of page i. The hub weights can be similarly written as a vector h. Using the definition of matrix-vector multiplication, we can now check that the updating of hub weights in terms of authority weights is simply the act of setting h equal to Ma; correspondingly, setting a equal to $M^T h$ updates the authority weights. (Here M^T denotes the transpose of the matrix M.) Running these updates n times each from starting vectors a_0 and h_0, we obtain $a = (M^T(M(M^T(M \cdots (M^T(Ma_0)) \cdots))))$ $= (M^T M)^n a_0$. This is the power iteration method for computing the principal eigenvector of $M^T M$, in which we repeatedly multiply some fixed starting vector by larger and larger powers of $M^T M$. (As we do this, we also divide all coordinates of the vector by a scaling factor to prevent them from growing unboundedly.) Hence this eigenvector is the stable set of authority weights toward which our updates are converging. By completely symmetric reasoning, the hub weights are converging toward the principal eigenvector of MM^T.

A related link-based measure is *PageRank*, defined by a different procedure that is also based on repeated refinement. Instead of drawing a distinction between the voters and the voted-on, one posits a single kind of quality measure that assigns a *weight* to each page. A current set of page weights is then updated by having each page distribute its weight uniformly among the pages it links to. In other words, receiving links from high-quality pages raises one's own quality. This too can be written as multiplication by a matrix, obtained from M^T by dividing each row's entries by the number of outgoing links from the corresponding page; repeated updates again converge to an eigenvector. (There is a further wrinkle here: repeated updating in this case tends to cause all weight to pool at "dead-end" pages that have no outgoing links and hence nowhere to pass their weight. Thus, to obtain the PageRank measure used in applications, one adds a tiny quantity $\epsilon > 0$ in each iteration to the weight of each page; this is equivalent to using a slightly modified matrix.)

PageRank is one of the main ingredients in the search engine Google; hubs and authorities form the basis for Ask Jeeves' search engine Teoma, as well as a number of other Web search tools. In practice, current search engines (including Google and Teoma) use highly refined versions of these basic measures, often combining features of each; understanding how relevance and quality measures are related to large-scale eigenvector computations remains an active research topic.

7 Distributed Algorithms

Thus far we have been discussing algorithms that run on a single computer. As a concluding topic, we briefly touch on a broad area in computer science concerned with computations that are *distributed* over multiple communicating computers. Here the problem of efficiency is compounded by concerns over maintaining coordination and consistency among the communicating processes.

As a simple example illustrating these issues, consider a network of automatic bank teller machines (ATMs). When you withdraw an amount of money x at one of these ATMs, it must do two things: (1) notify a central bank computer to deduct x from your account, and (2) emit the correct amount of money in physical bills. Now, suppose that in between steps (1) and (2), the ATM crashes so that you don't get your money; you'd like it to be the case that the bank doesn't subtract x from your account anyway. Or suppose that the ATM executes both of steps (1) and (2), but its message to the bank is lost; the bank would like for x to be eventually subtracted from your account anyway. The field of distributed computing is concerned with designing algorithms that operate correctly in the presence of such difficulties.

As a distributed system runs, certain processes may experience long delays, some of them may fail in mid-computation, and some of the messages between them may be lost. This leads to significant challenges in reasoning about distributed systems, because this pattern of failures can cause each process to have a slightly different *view* of the computation. It's easily possible for there to be two runs of the system, with different patterns of failure, that are "indistinguishable" from the point of view of some process P; in other words, P will have the same view of each, simply because the differences in the runs didn't affect any of the communications that it received. This can pose a problem

if P's final output is supposed to depend on its having noticed that the two runs were different.

A major advance in the study of such systems came about in the 1990s, when a connection was made to techniques from algebraic topology. Consider for simplicity a system with 3 processes, though everything we say generalizes to any number of processes. We consider the set of all possible runs of the system; each run defines a set of three views, one held by each process. We now imagine the views associated with a single run as the three corners of a triangle, and we glue these triangles together according to the following rule: for any two runs that are indistinguishable to some process P, we paste the two corresponding triangles together at their corners associated with P. This gives us a potentially very complicated geometric object, constructed by applying all these pasting operations to the triangles; we call this object the *complex* associated with the algorithm. (If there were more than three processes, we would have an object in a higher number of dimensions.) While it is far from obvious, researchers have been able to show that the correctness of distributed algorithms can be closely connected with the topological properties of the complexes that they define.

This is another powerful example of the way in which mathematical ideas can appear unexpectedly in the study of algorithms, and it has led to new insights into the limits of the distributed model of computation. Combining the analysis of algorithms and their complexes with classical results from algebraic topology has in some cases resolved tricky open problems in this area, establishing that certain tasks are provably impossible to solve in a distributed system.

8 For Further Reading

Algorithm design is a standard topic in the undergraduate computer science curriculum, and it is the subject of a number of textbooks including Cormen et al. [4] and a forthcoming book by the author and Éva Tardos [6]. The perspective of early computer scientists on how to formalize efficiency is discussed by Sipser [10]. The Traveling Salesman and Minimum Spanning Tree problems are fundamental to the field of combinatorial optimization; the TSP is used as a lens through which to survey this field in a book edited by Lawler et al. [7]. Approximation algorithms and local search algorithms for computationally intractable problems are discussed in books edited by Hochbaum [5] and by Aarts and Lenstra [1] respectively. Web search and the role of link analysis is covered in a book by Chakrabarti [2]; beyond Web applications, there are a number of other interesting connections between eigenvectors and network structures, as described by Chung [3]. Distributed algorithms are covered in a book by Lynch [8], and the topological approach to analyzing distributed algorithms is reviewed by Rajsbaum [9].

Bibliography

1. E. Aarts and J.K. Lenstra (eds.). *Local Search in Combinatorial Optimization.* Wiley, 1997.
2. S. Chakrabarti. *Mining the Web.* Morgan Kaufman, 2002.
3. F.R.K. Chung, *Spectral Graph Theory.* AMS Press, 1997.
4. T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press, 2001.
5. D.S. Hochbaum (ed.). *Approximation Algorithms for NP-hard Problems.* PWS Publishing, 1996.
6. J. Kleinberg, É. Tardos. *Algorithm Design.* Addison Wesley, 2005.
7. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, 1985.
8. N. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
9. S. Rajsbaum. Distributed Computing Column 15. *ACM SIGACT News* 35:3(2004).
10. M. Sipser. The history and status of the P versus NP question. *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

Biography of contributor

Jon Kleinberg received his A.B. from Cornell in 1993 and his Ph.D. in computer science from MIT in 1996. He subsequently spent a year as a Visiting Scientist at the IBM Almaden Research Center, and is now an Associate Professor in the Department of Computer Science at Cornell University. He has received research fellowships from the Packard and Sloan Foundations, and he received the 2001 U.S. National Academy of Sciences Award for Initiatives in Research for his work on algorithms in network analysis and Web search.