Cartography and Geographic Information Science Manuscript-Nr.
(will be inserted by hand later)

Complexity and Intractability: Limitations to Implementation In Analytical Cartography

Alan Saalfeld[1]

Department of Civil and Environmental Engineering & Geodetic Science, The Ohio State University, 2070 Neil Avenue, Hitchcock Hall 470, Columbus, OH 43210-1275, USA

Submitted December 6, 1999, Revised March 8, 2000

Abstract. The computational complexity of algorithms is an important consideration for all computer systems, including geographic information systems and mapping systems. Mathematical cartographers and GIS professionals need to understand and to take into account the limitations imposed on problem-solving by the very nature of computation itself. We look at three active research sub-areas of analytical cartography to highlight the differences between traditional mathematical solutions and solutions with computationally tractable algorithms. The three sub-areas are map projections, map feature labeling, and map generalization.

Key words: Algorithms, Computational Complexity, Tractability, Projections, Map Labeling, Generalization

Introduction

Computer mathematics is finite mathematics with a special twist to the word "finite". The number and the size of objects that can be handled by even the world's fastest computer at any given moment in history are not only finite, but they are also bounded. The number of operations that any given machine can perform in a reasonable (or even unreasonable) period of time is also bounded. Computer scientists realized quickly that many mathematical solutions even to very simply formulated problems could not be implemented. The solution methods were computationally intractable—any computer algorithm to implement the mathematical solution would have to execute so many instructions that it would take too long to run. In this paper we identify some hard problems in analytical cartography that might be re-examined with an eye for "other options"—non-traditional solution approaches. We also note characteristics of classes of tractable problems that do not require non-traditional and possibly sub-optimal solution approaches.

Correspondence to: saalfeld.1@osu.edu

There are many meanings for the phrase "complexity of a problem." A *hard problem* may refer to a problem that students complain about taking up too much of their time, or a problem that you personally have worked on unsuccessfully for long periods, or even a problem like proving Fermat's Last Theorem that had remained unsolved for centuries. One way or another, time often plays a role in the measure of problem complexity. A problem that takes only two seconds to solve fails to meet anyone's definition of "hard."

The main objective of the theoretical computer science discipline of *complexity analysis* is to place problems and algorithms that solve those problems on a scale of difficulty. The scale of difficulty is often a proxy measure for the amount of computer time (or other resources) needed to solve the problem or to run an algorithm that generates a solution to the problem. Under this how-much-time-does-it-take measure, algorithmic solutions to several interesting problems wind up with a "difficulty value" so large that the corresponding computer running time would be outrageously long—years, or lifetimes, or eons.

When complexity analysis shows that a program to run an algorithm would take too long, the reasonable scientist stops coding the algorithm and looks for other options. The first option to look for is an alternative mathematical formulation that leads to a more efficient algorithm for solving the problem. We have all seen hard problems solved by "looking at them the right way." My favorite example is the domino/checkerboard problem: 32 dominoes cover the 64 squares on a checkerboard (each domino covers two adjacent squares). Cut out two squares from opposite corners of the checkerboard. How many ways can you place 31 dominoes on the doubly notched checkerboard so that all 62 remaining squares are covered? One might try to systematically construct and enumerate all possible placements (a so-called "brute force" solution), but a more efficient and insightful way to arrive at the correct answer, "none", is to note that opposite corners on a checkerboard have the same color; and each domino must cover one black square

and one red square. The cut checkerboard no longer has the same number of squares of each color.

Insight into mathematical structure may reveal an algorithm that exploits that structure very efficiently. One may, after all, have simply tried to solve the problem with a very inefficient algorithm when better algorithms are available. If a trial algorithm runs too slowly, one of the first options is to question your choice of algorithm and try to do better.

If this approach fails, the mathematician/computer scientist may suspect that no efficient algorithm exists and may alternatively try to prove that the problem in question is *intrinsically hard* by showing that it is equivalent to some well known hard problem, such as the Traveling Salesperson Problem. The Traveling Salesperson Problem can be loosely stated as follows: given n cities (n point locations in a plane or on a surface), find a closed path that visits all of the cities and has the shortest total length among all closed paths that visit all of the cities. Theoretical computer scientists have classified a substantial number of intrinsically hard problems and have developed the mathematical machinery for proving equivalence to these hard problems [Garey 79]. One such class of hard problems, called NP-complete problems ("NP" stands for "Nondeterministic Polynomial"), has a special air of mystery to it because of the following gap in the theory. Computer science theoreticians and mathematicians have tried, but so far have been unable to prove that the problems in the class NP are actually harder than the problems in the class P (those that have polynomial-time solutions). Nevertheless, the NP-complete problems seem to be much harder in the following sense: every algorithm known so far for solving any NP-complete problem requires running times that are exponential, not polynomial, functions of the problem size. The Traveling Salesperson Problem is an NP-complete problem for which the only known exact algorithms are exponential functions of the problem size n, and implementing exponential algorithms is not practical for anything but very small n (see Table 1). The fact that no one has been able to find any polynomial time solution to any problem in NP only makes the search for a proof more compelling. Answering the question "Does $\mathsf{P} = \mathsf{NP}$?" is the ultimate challenge for computer science theoreticians today. The person that answers that question will become famous—at least among other computer scientists and mathematicians!

If a problem is demonstrably hard (i.e., shown to be NP-complete), the reasonable scientist will try to solve an easier related problem, perhaps by focusing on a special case of the problem, or by seeking only an approximate solution to the problem, or by seeking a feasible, but possibly suboptimal, solution to a relaxed version of the problem. When a problem has been determined to be NP-complete, the only alternative to this settle-for-less approach is to find a polynomial-time algorithm and thereby prove that $\mathsf{P} = \mathsf{NP}$.

In the final half of this paper, we will identify two NP-complete problems that cartographers should be aware of. The first is label placement for point labels, and the second is a decidability problem related to cartographic line simplification. We will also point out that many cartographic problems such as computing map projections and their inverses are quite tractable and may be solved systematically with efficient algorithms based on binary search techniques. Before we look at the cartographic examples, however, we want to examine the tools and terminology of complexity analysis.

Algorithm Complexity

We have mentioned polynomial-time algorithms, exponential-time algorithms, NP-complete problems, and equivalence of hardness of problems without providing any definitions or examples of these terms. In this section we will examine some of the basic notions that make up the core of computational complexity theory in the analysis of algorithms. Later, we will illustrate these notions with simple examples taken from computational problems that might arise in computer mapping and in GIS applications.

There have been many notions of complexity over the long history of problem solving, but computational complexity is a theory that is less than four decades old. Computational complexity theory has grown right alongside the explosive growth of computers and computer science. The theory provides a means of measuring and summarizing an algorithm's performance on different sized inputs. Complexity theoretical methods almost always deal with inequalities, not equalities; and the methods often give bounds, upper and lower, to describe limits on how slow or how fast an algorithm might run. The principal reason that the theory produces bounds and ranges instead of exact numbers is that algorithm performance does not depend only on the size n of the input data set, but it also depends to an even greater degree on the actual n values that the input data assumes.

An Example of Analyzing an Algorithm's Complexity

Consider the pseudo-coded Algorithm 1 for finding the closest pair of points taken from an input set of n points $\{p_1, p_2, p_3, \ldots, p_n\}$ by measuring and comparing distances between all pairs. We will try to determine how many times each of the 16 instructions gets executed.

```
(1)   Set d_min = distance{p_1, p_2};
(2)   Set i_min = 1;
(3)   Set j_min = 2;
(4)       For i = 1 to n − 1
(5)           For j = i + 1 to n
(6)               Compute d = distance{p_i, p_j};
(7)               If d < d_min, then do
(8)                   Set d_min = d;
(9)   .   .   .   .   Set i_min = i;
(10)                  Set j_min = j;
(11)              EndIf;
(12)          End For j;
(13)      End For i;
(14)  Print d_min;
(15)  Print p_{i_min};
(16)  Print p_{j_min};
```

Algorithm 1: Find the distance d_{min} between the nearest pair of points $p_{i_{min}}$ and $p_{j_{min}}$ in $\{p_1, p_2, p_3, \ldots, p_n\}$.

This algorithm systematically computes all interpoint distances between p_i and p_j for all i and j with $i < j$: if i is 1, then j successively takes on all $n-1$ values between 2 and n. When i is 2, then j successively takes on all $n-2$ values between 3 and n. The outer "For" loop holds i fixed while j runs through all possible values that are bigger than i. We can easily compute the number of times that the instruction (6) executes: $(n-1)+(n-2)+\cdots+3+2+1$, which is equal to $n(n-1)/2$. The test in instruction (7) also executes the same number, $n(n-1)/2$, of times. Depending on the results of that test, instructions (8), (9), and (10) may or may not execute. The first three instructions, (1), (2), and (3), each execute only once. They initialize the variables that keep track of the best answers so far, i.e., those variables get updated and changed each time a closer pair is found using the test in instruction (7). The last three instructions, (14), (15), and (16), also execute just once each to deliver the algorithm output. The two nested "For" loop instruction pairs, {(4), (13)}, and {(5), (12)}, each involve several more fundamental instructions, including initializing the counter (once each time the loop is entered), incrementing the counter (i gets incremented $n-1$ times), and testing the counter to see if it has exceeded the loop limit (the test on i is done n or $n-1$ times, depending on the language[1] used to implement this pseudo-code).

We might summarize the preceeding paragraph as follows: two instructions are executed $(n^2/2-n/2)$ times each, the outer "For" loop assigns and/or reassigns a value for i at least n times, some instructions are executed only once, some instructions may be executed not at all, but no instruction of the 16 is executed as many as n^2 times. If we use these facts in the preceeding statement to find upper and lower bounds on the total number of instructions executed (counting multiplicities) when the algorithm is run, we see that the number of instructions executed is greater than n^2, but less than $16n^2$. This may seem like a wide range, and, indeed, we can certainly reduce the coefficient of 16 if we try. However, the key relationship that we have here that is used extensively in computational complexity analysis is the following chain of inequalities, (which, for our Algorithm 1 is true when $\alpha = 1$ and $\beta = 16$):

$$\alpha n^2 \leq \text{Number of instructions executed} \leq \beta n^2, \quad \text{for some } \alpha, \beta > 0. \tag{1}$$

These inequalities state that the number of instructions executed for any input of n points is bounded above by a multiple (β) of n^2 and also bounded below by a (different) multiple (α) of n^2. Let's look at each of these inequalities separately. When we have the following upper bound inequality,

$$\text{Number of instructions executed} \leq \beta n^2, \quad \text{for some } \beta > 0, \text{ and for all } n > \text{some } n_0. \tag{2}$$

we may use a shorthand description to say that the algorithm is "big-oh of n^2", which we write $O(n^2)$. More generally, when we have the upper bound inequality for some function $f(n)$,

$$\text{Number of instructions executed} \leq \beta f(n), \quad \text{for some } \beta > 0, \text{ and for all } n > \text{some } n_0. \tag{3}$$

we use a shorthand description to say that the algorithm is "big-oh of f of n", which we write $O(f(n))$. When we have the lower bound inequality for some non-negative function $f(n)$,

$$\alpha f(n) \leq \text{Number of instructions executed}, \quad \text{for some } \alpha > 0, \text{ and for all } n > \text{some } n_0. \tag{4}$$

we say that the algorithm is "big-omega of f of n", which we write $\Omega(f(n))$. When an algorithm is both $O(f(n))$ and $\Omega(f(n))$ for the same function f simultaneously, we say that the algorithm is "big-theta of f of n", and we write $\Theta(f(n))$. We have shown by our chain of inequalities (1) that Algorithm 1 is $\Theta(n^2)$ because it is both $O(n^2)$ (bounded above by βn^2) and $\Omega(n^2)$ (bounded below by αn^2).

Asymptotics

What does it mean to be "bounded from above and below by a multiple of n^2"? An upper bound puts an upper limit on how fast the number of operations can grow as the input n increases. If the input n doubles in size, the upper bound will quadruple. If the input of size n has a ten-fold increase in size to $10n$, the upper bound βn^2 increases one-hundred times to $\beta(10n)^2 = 100\beta n^2$. If the lower bound increases quadratically with n, then we have a lower bound growing one hundredfold each time the input size increases ten-fold. The fact that the lower bound is growing so fast forces the number of instructions executed to grow somewhat apace to stay above it. Although the number of instructions is not growing exactly quadratically, it is trapped between two functions, each of which is growing quadratically.

The algorithm analyst is mostly concerned with the behavior of the algorithm for larger and larger input

[1] Most languages perform the test before the first execution. Fortran does not.

sizes. The analyst wants to study so-called asymptotic behavior that ignores setup and bookkeeping instructions of lower orders of magnitude and keeps track of those instructions that are executed most frequently when n is large. Consider, for instance, the example of a computer program coded with k instructions, each of which gets executed some number of times. Suppose that the time needed to execute the jth instruction a single time is $t_j > 0$, for $j = 1, 2, \ldots, k$. (Here we are even allowing each instruction to have a possibly different execution time, but each instruction has the same execution time every time it is executed). If one instruction, say the i_0^{th}, were to execute n^2 times and all other instructions executed only n times, then, for larger and larger n, the time needed to execute the i_0^{th} instruction n^2 times would use up a larger and larger fraction of the total time needed to run the entire algorithm. In fact, that fraction is just:

$$\frac{\text{Time for } i_0^{\text{th}} \text{ instruction}}{\text{Time for all instructions}} = \frac{n^2 t_{i_0}}{n^2 t_{i_0} + \sum_{j \neq i_0}^{j \leq k} n t_j},$$

$$\text{where } \lim_{n\to\infty} \frac{n^2 t_{i_0}}{n^2 t_{i_0} + \sum_{j \neq i_0}^{j \leq k} n t_j} = \lim_{n\to\infty} \frac{1}{1 + (\frac{1}{n})(\frac{1}{t_{i_0}}) \sum_{j \neq i_0}^{j \leq k} t_j} = 1.$$

Notice that for large n, the time spent on the i_0^{th} instruction dominates the time spent on everything else put together, even if the time t_{i_0} that it takes to execute the i_0^{th} instruction once is much, much shorter than all other single-instruction execution times t_j. The concept of asymptotic behavior, how the one instruction "takes over" in this example as n "goes to infinity," permits complexity analysis to focus on essential bottlenecks to algorithm speedup.

Problem Complexity versus Algorithm Complexity

The complexity of a problem is the complexity of the best algorithm that solves it. In many important situations we do not know exactly what the best algorithm is. Complexity theoreticians use "big-oh", "big-omega", and "big-theta" to describe problems as well as algorithms. A problem is $O(f(n))$ if there exists some algorithm that is $O(f(n))$ that solves the problem. If a problem is $O(f(n))$, then $f(n)$ is a known upper bound to the problem's complexity. A problem is $\Omega(g(n))$ if every algorithm that solves the problem is $\Omega(g(n))$. If a problem is $\Omega(g(n))$, then $g(n)$ is a known lower bound to the problem's complexity. A problem is $\Theta(h(n))$ if the problem is both $O(h(n))$ and $\Omega(h(n))$. Showing that a problem is $O(f(n))$ is fairly straightforward—just produce any algorithm that solves the problem and that runs in $\beta f(n)$ time for some β and for all n greater than some n_0. Showing that every algorithm that solves a problem needs at least some multiple of $g(n)$ time for all sufficiently large n often demands a somewhat trickier approach. To show, for example, that every sorting algorithm based on pairwise comparison needs to make at least $\alpha n \log n$ comparisons to get the job done, one considers the totality of $n!$ unsorted situations that may present themselves to an algorithm and the differentiating power that an algorithm can have at each stage. For $n > 7$, a binary decision tree with $n!$ leaf nodes must have depth greater than $(n/2)\log n$; hence, to handle all possible inputs, the comparison test must execute at least $(n/2)\log n$ times on some inputs. The fact that there are comparison sort algorithms that are $O(n \log n)$ (heap sort, for example) tells us that the problem of sorting-by-comparisons is $\Theta(n \log n)$.

There is a large class of problems, the NP-complete problems, for which we can only compute unequal "big-oh" and "big-omega" values. We can show that every algorithm that solves an NP-complete problem must be at least polynomial in n. In some cases, we can show that the polynomial must be at least degree d for some d that depends on the particular NP-complete problem. For some NP-complete problems, a solution involves finding a suitable or an optimal subset of the n input values. Such problems are $O(2^n)$ simply because there is an $O(2^n)$ algorithm that checks all subsets. Checking all subsets quickly gets impractical as n increases (see Table 1).

The collection of NP-complete problems are all equivalently hard in the following sense: if we build an algorithm in $O(f(n))$ to solve any one of them, then given any other NP-complete problem, we may transform our algorithm into an algorithm to solve that other NP-complete problem. The transformed algorithm to solve the other NP-complete problem will be $O(p(f(n)))$, where p is a polynomial expression in the complexity $f(n)$ of the original algorithm. In particular, if we could build the original algorithm to run in polynomial time, then, since a polynomial of a polynomial is again a polynomial (of higher degree), our transformed algorithm would also run in polynomial time. The essence of the question "Does P = NP?" boils down to "Can we find just one polynomial-time algorithm to solve just any one NP-complete problem?"

There is also a larger class of problems called NP-hard problems that are problems that can be shown to be at least as hard as NP-complete problems (and maybe harder). A problem is NP-hard if any $O(g(n))$ algorithm to solve it can be polynomially-transformed into an $O(p(g(n)))$ algorithm to solve some NP-complete problem. If an algorithm can be polynomially-transformed into an algorithm to solve some NP-complete problem, it can be polynomially-transformed into algorithms to solve all NP-complete problems. The converse need not be true. It may not be possible to polynomially-transform a solution to an NP-complete problem into a solution to an NP-hard problem.

Intractability

Intractability refers to the difficulty class of a problem. A problem is *intractable* if no polynomial-time solution is

known. Intractable problems include NP-complete and NP-hard problems (at least until someone proves that P = NP). Many network optimization problems are intractable. Lots of geometric problems are intractable. Several authors have compiled lists and categories of intractable problems [Blum 98, Garey 79, Hochbaum 97, Iturriaga 97, Karp 74, Traub 76a, Traub 76b].

If we do not have a polynomial-time algorithm for solving an industrial strength problem, we cannot even start to generate a solution. For many real-world problems, having a polynomial-time solution may still not be enough if the polynomial has high degree. Tractability is necessary, but it is not always sufficient.

One measure of suitability of an algorithm is the size of a problem that the algorithm may handle in some reasonable amount of time. Since computers are always getting faster, the size of a manageable problem is getting bigger. Suppose an algorithm can handle a problem of size n now. If the computer gets 10,000 times faster, how much bigger a problem can it handle? Table 1 gives us an answer. (According to a poorly paraphrased Corollary to Moore's Law, computers double their speed every 18 months. At that rate it would only take 20 years to get 10,000 times faster.)

Reasonable Ways of Dealing With Provably Intractable Problems

If you prove that a problem is NP-hard (or if someone proves it for you), you have several options. If you must have an efficient exact algorithm and nothing less will satisfy your requirements, then you should give up and hope that someone proves that P = NP. If you have real work to get done and need some solution that may not be the exact best solution, then you may try one of the following methods.

1. Look for special cases of the general problem that apply to your particular situation: Do we really need to solve the problem in all its generality? For example, some problems involving graphs are NP-complete, but can be solved in the special case of trees in polynomial time.
2. Try changing the problem statement: Add constraints or make limiting assumptions.
3. Change the problem domain. For example, consider robot movement restricted to horizontal and vertical movement (rectilinear movement). Then a robot has four choices for direction instead of infinitely many. Another example of changing the problem domain: Some problems (e.g., many problems related to visualization and graphic display) only require an approximate solution. An "adequate" picture has a range of acceptable resolutions, not just one.
4. Change your expectation for a solution by accepting piecewise or sub-optimal solutions. Use backtracking heuristics for yes/no decision problems. Use branch-and-bound techniques for optimization problems. Use local improvement techniques like hill-climbing. Apply specialized tools like simulated annealing to deal with misleading local optima.
5. Accept approximations after studying the latest results in approximation theory.

In just the past decade, theoretical computer scientists have broken the NP-complete algorithms into subcategories according to how well an optimal solution can be approximated [Baker 94, Hochbaum 97, Lewis 98]. They have identified at least three large categories of ϵ-approximation algorithms, where $[\mathrm{opt}(x) - A(x)]/\mathrm{opt}(x) \leq \epsilon$. The three categories are fully approximable (for ϵ, however small), partly approximable (for some range of ϵs), and inapproximable (for no ϵ, however large). Learn which category your problem lies in.

Existence proofs, a staple of mathematics, are not enough in the world of computation! Knowing a solution exists, but not being able to get to it, can be terribly frustrating. Knowing exactly when to settle for something other than optimal removes some frustration. Intractable algorithms are doomed to only run on very small sets. Some mathematical solutions are like a bridge to the moon. Conceptually, we can mentally manipulate the characteristics of such a bridge–its length, the number of bricks we would need, etc. Realistically, we have no hope whatsoever of building such a structure, even if adequate materials existed!!

Complexity of Some Mapping and GIS Problems

In this section we will examine some of the computational problems of current interest to the mapping and GIS communities. We will look at map projections and their inverses, common map transformations used for graphic display, several versions of map labeling problems, and some generalization problems and algorithms that have been proposed to solve those problems. First we examine issues of map projections.

Complexity of Map Projections and their Inverses

Map projections are transformations from all or part of the sphere or ellipsoid model of the Earth to the plane. A map projection is entirely specified by the action of x and y coordinate functions on sphere or ellipsoid coordinates, usually given as latitude (ϕ) and longitude (λ). The x and y coordinate functions may be given explicitly as forward equations for map projections, $x = x((\phi, \lambda))$, $y = y((\phi, \lambda))$. If the x and y coordinate functions are explicitly presented as a sequence of products and sums of trigonometric, polynomial, and other readily computable common functions, then they can be evaluated at any one point on a sphere or ellipsoid, (lat-long equals (ϕ, λ)), in constant time, $\Theta(1)$. Computing n points of a projection from explicit forward equations can be done in $\Theta(n)$ time. An example of an easy-to-compute pair of forward equations for an equal-area cylindrical projection from the sphere is:

$$x = R\lambda$$
$$y = R\cos(\phi)$$

Example	Algorithm Complexity	Size handled in 10^8 ops Approx. time 1 sec	in 10^{10} ops 1.7 min	in 10^{12} ops 2.7 hr	in 10^{16} ops 3 yr	in 10^{20} ops 300 centuries
(Examine n objects)	n	10^8	10^{10}	10^{12}	10^{16}	10^{20}
(Sort n objects)	$n \log n$	4×10^6	3.2×10^8	2.5×10^{10}	2×10^{14}	1.5×10^{18}
(Enumerate pairs taken from $\{1, \ldots, n\}$)	n^2	10^4	10^5	10^6	10^8	10^{10}
(Enumerate triples taken from $\{1, \ldots, n\}$)	n^3	460	2200	10,000	2.2×10^5	4.6×10^6
(Enumerate 4-sequences of $\{1, \ldots, n\}$)	n^4	100	320	1,000	10,000	10^5
(Examine all subsets of $\{1, \ldots, n\}$)	2^n	26	32	39	52	65
(Generate all n digit decimal numbers)	10^n	8	10	12	16	20

Table 1. Size of Problem Handled by Algorithms of Different Complexity

For this particular pair of forward equations, it is straightforward to compute the inverse equations that express λ as a function of x and y and also express ϕ as a function of x and y:

$$\lambda = x/R$$
$$\phi = \arccos(y/R)$$

Since we may evaluate this pair of equations for any value of x or y in constant time, we may evaluate the inverse functions for λ and ϕ at any n points in $\Theta(n)$ time. To draw our maps or inverse functions, we may want to select a number of regularly spaced grid points in our domain to map and connect in our range space. The domain for the projection maps is the (ϕ, λ) grid. The domain for the inverse function is a rectangular, regularly spaced (x, y) grid. Ideally, we would like to develop a closed-form formula for projections and their inverse functions. If we are able to do this, then we may evaluate the functions directly at every point with a fixed number of arithmetic, trigonometric, and other operations. If we have a closed-form formula for the forward equations, but we do not have a closed expression for the inverse function, we can, nevertheless, still use binary search and the forward functions to find a regular grid of $n = m \times m$ points in the x-y-plane and the (ϕ, λ) pairs that they come from. Since a dense regular grid is more than enough to generate the necessary coordinates of an inverse projection, and since each binary search has complexity $\Theta(\log n)$, we can populate a dense grid of x-y values in $\Theta(n \log n)$ time

Not all projection constructions possess closed-form formulae, especially when an ellipsoid is used for the datum surface. When we have no closed-form formula, we may compute and store the projection information by obtaining a complete set of (x, y) values evaluated on a dense regular grid of (ϕ, λ) pairs. Numerical solutions are frequently generated by iterative methods that converge after a small fixed number of iterations to a solution at each (ϕ, λ) pair. Projections defined implicitly by differential equations may be constructed using difference equation approximations and iterative improvement methods. Low-order terms of analytic infinite series expressions may be computed, and bounds may be found for error magnitudes of the truncated tails. Most of the aforementioned approximation methods evolved to reduce the number of arithmetic calculations made by hand in pre-computer eras. A number of still relevant pre-computer notions of *complexity* of functions defined by other than closed-form formulae are the following:

1. How many points need to be evaluated (and to what level of precision) to obtain a satisfactory pointwise approximation to a projection function or its inverse?
2. How fast do infinite series approximations converge to a limiting sum? (How many terms in the series must we evaluate?) Is the convergence uniform? (Can we use the same number of terms for all points, or do some evaluations require computing additional terms in the series?)
3. How fast do iterative approximation procedures (such as the Newton-Raphson method) converge to a solution?
4. How can seed values be chosen to guarantee faster convergence of iterative approximations?
5. How can convergence to local/non-global minima and maxima be handled properly?
6. How can arithmetic operations be re-organized and simplified to make them more robust (for example, using Bernstein polynomials as basis functions)?
7. What level of precision is required for numeric computations to limit or bound the accumulated detrimental effects of rounding errors and/or truncation errors?

Both explicit forward equations or implicit coordinate relationships given by differential properties can be used to compute x and y coordinate values for any (ϕ, λ) to a pre-specified precision by a fast $O(1)$ (constant-time for all inputs) algorithm. The complexity of computing the projection values on an entire grid of size n is, therefore, linear ($\Theta(n)$) in the size of the grid.

The inverse transformation is estimable on an entire regularly spaced grid of $n(= m\text{-by-}m)$ (x, y) points in $O(n \log n)$ time. Binary search suffices to obtain a suitable dense regular grid in the range space. Newton-Raphson methods may be used to speed up the rate of convergence, but the speedup will not improve the asymptotic complexity, but only the constant factor in the rate of convergence. The number of iterations needed to home in on a desired grid point in the range space with a pre-specified precision varies as the logarithm of the maximum local scale factor if simple binary search is used.

Complexity of Transformations for Graphic Representations

A number of data structures have been built to facilitate spatial queries and graphic representation of spatial data. Voronoï diagrams, for example, can efficiently

partition space to facilitate efficient nearest-neighbor searches. The Delaunay Triangulation, the planar dual to the Voronoï diagram, can be built simultaneously with the Voronoï diagram. Delaunay Triangulations provide an important structure for creating faceted continuous surfaces from irregularly spaced point elevation data. Delaunay Triangulations have also been used extensively to facilitate conflation and rubber-sheeting operations.

The history of algorithms for building Delaunay Triangulations is very interesting. The first algorithms used to construct Delaunay Triangulations were straightforward and easy to implement [Green 77]. They employed a method of iterated incremental insertion of a single vertex followed by a "fixing up" of the triangulation after insertion to make it a Delaunay Triangulation again. The "fixing up" step involved attaching some new edges to the newly inserted vertex and possibly removing some old edges from nearby the newly inserted vertex. A careful analysis of "worst-case" performance of the incremental algorithm revealed that the "fixing up" operation could require $O(n)$ edges to be removed or inserted at each vertex insertion, making the overall complexity of adding n vertices $O(n^2)$. An $O(n^2)$ algorithm was considered potentially dangerously inefficient, even though the incremental insertion algorithm had an excellent empirical performance history. The algorithm designers built a Delaunay Triangulation algorithm with guaranteed worst-case performance $O(n \log n)$ using a divide-and-conquer approach. A divide-and-conquer algorithm builds the Delaunay triangulation on a vertex set by recursively dividing the vertex set in half, building the Delaunay triangulation on each half, then stitching the two halves together with connecting edges that complete or correct the Delaunay triangulation of the whole vertex set. Although the divide-and-conquer approach was theoretically asymptotically better than the straightforward incremental algorithm, it never performed as well as the incremental algorithm and it proved very difficult to implement correctly [Kao 91]. The difficulty in implementation was due to the "divide" step in the algorithm. The "divide" step requires the splitting of the vertex set into two sets of nearly equal size lying on either side of some splitting line (see Figure 1). Every choice of orientation for the splitting line has potential drawbacks for different vertex sets. Since some splitting line must be chosen suitably at each recursive call, the procedure for choosing the splitting line becomes the most complex step in the divide-and-conquer algorithm.

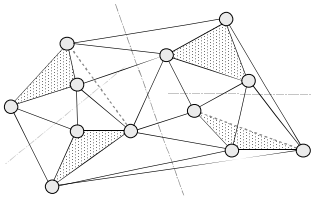

Fig. 1. Divide-and-Conquer Delaunay Triangulation Illustrated (Splitting lines shown and first 4 triangles shaded)

Finally, in 1990, some theoretical computer scientists [Knuth 90] introduced probabilistic analysis to the incremental algorithm and found that average case performance of the incremental algorithm was indeed better than that of the divide-and-conquer algorithm; and, moreover, randomizing the vertex insertion order assured that, *with high probability*, the incremental algorithm would perform better than the divide-and-conquer algorithm.

Many geometric problems still have not been solved. Some like the Traveling Salesperson Problem have at least been shown to be NP-complete. One of the more tantalizing open geometric problems whose complexity class is not even known is called Minimum Total Length Triangulation [MTLT] (or sometimes Minimum Weight Triangulation [MWT]). The Minimum Total Length Triangulation is the triangulation among all possible triangulations whose sum of edge lengths is smallest. If we were able to show that this problem were NP-hard, then we would at least feel justified to stop looking for tractable solutions to it. What keeps us from showing that MTLT is NP-hard is that we do not even know how to check in polynomial time that a candidate triangulation is really the triangulation with the shortest total edge length.

Label Placement Is NP-*Hard*

Label placement, the problem of placing text on a map to label the features, is really many different problems. Point, line and region labeling have different constraints. What constitute acceptable cartographic placement rules vary by location, by need, or by *fiat*. Permissible placements vary as do preferred placements. Some solutions address only feasibility: can you find a legitimate non-conflicted location assignment for every label? Other solutions look for optimality: can you find a placement of labels that is better in some measurable way than any other possible placement? Deciding feasibility (and finding ANY non-conflicting label placement) is the hard problem. Even when placement of labels is constrained to choosing from a finite configuration (thus making the problem a combinatorial one), the number of choice combinations increases geometrically or exponentially.

Point, line, and area labeling can be formulated as different problems. Of the three classes of problems, point labeling is probably the easiest. Nevertheless, NP-completeness for point labeling was stated explicitly and proved in 1991 [Marks 91], although earlier proofs in 1981 of a packing problem's complexity [Fowler 81] and in 1988 of a character placement problem [Kato 88] easily imply the NP-completeness result for point labeling. In spite of proofs that the problem is NP-complete, and, hence, "solving the point labeling problem efficiently will prove that P = NP", some people are still searching for the "optimal presumably polynomial solution" to point labeling. Those people are either very optimistic or very ignorant. Inasmuch as NP-hardness has now been proved, some people's approach to this problem reminds me of the people still trying to "square the circle" or to

trisect an arbitrary angle with ruler and compass. Mathematicians have proved something to be impossible, but because the layman cannot understand an impossibility proof, the layman continues to try to do the impossible! "NP-hard" does not mean "impossible", but it sure does mean that finding a polynomial solution is not going to be easy!

For real-world labeling algorithms (and, yes, somebody needs to put labels on maps), heuristics abound. Heuristics abound because anything that we can come up with right now is a heuristic and not a tractable exact solution. The map labeling business is a very active and profitable research area. Options for other-than-optimal solutions are many.

Map Generalization: Posing the Mathematical Problem

Just as map labeling proved to be many different problems, map generalization consists of many distinct and sometimes even conflicting problems [McMaster 92]. Sometimes the cartographer has to weigh tradeoffs against each other [Müller 95]. For example, polyline simplification versus "effective" shape representation are two competing interests in cartographic generalization. One key observation is that these two activities are not comparably quantifiable. As a consequence, one cannot even tell if a problem has been solved when a change is made to simplify a polyline that is part of a larger shape representation! Our mathematical tools often limit us to quantifying the relative goodness of a single characteristic (e.g., number of segments, closeness of approximating polyline). Even the very feasibility of a "good generalization" is not easily specified with quantifiable measures.

Much of the algorithmic effort in map generalization has consequently been directed at simpler isolated subproblems that can be evaluated in and of themselves. A typical example is the classic Douglas-Peucker Polyline Simplification Algorithm that has seen regular attempts to spruce it up and improve it over the years. What many of the theoretical efforts accomplish is to tweak the upper bound complexity down a notch by shaving off a log factor through the use of a complex data structure. In reality, the old $O(n^2)$ original algorithm ran just fine.

One recent complexity analysis of polyline simplification by shape-point removal has revealed an NP-complete problem at the core of topologically consistent holistic polyline simplification. Regina Estkowski recently proved [Estrowski 98] that "deciding whether a map having k polyline chains made up of n intermediate shape points can be simplified to have the same k chains using only m of the n intermediate shape points" is NP-complete. Her geometric constructs illustrate how removing shape points from one chain can alter the possibility of removing shape points from another chain. Although her proofs do not provide a strategy for accomplishing line generalization, they clearly illustrate the dependence of one feature's repositioning on another's potential repositioning. That dependence is well known in the problem of contour line simplification, as illustrated in Figure 2, in which no single contour vertex can be removed except along the innermost contour. Contour simplification requires simultaneous coherent adjustment (holistic approaches [Jones 95]) along all nearby contours.

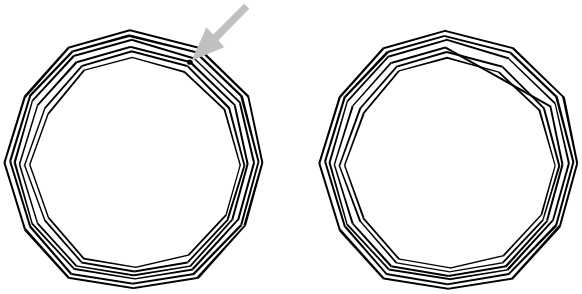

Fig. 2. No point or subset of points may be removed from a single outer or intermediate contour without upsetting topology

It is impossible to solve a poorly posed problem, but it is easy to argue that a proposed "solution" is not a solution at all. A better mathematical formulation of the generalization problem is required for an analysis of complexity to proceed. One mathematical model, for example, views generalization as either a homeomorphism or a continuous deformation retract of one topological cell structure onto a simpler cell structure. A "feasible" solution to this problem may turn out not to be a simplification at all!

Conclusions

Theoretical Computer Science has provided a new and useful framework in which to solve well-posed mathematical problems. Notions of being effectively computable (having an algorithm that terminates, and, upon terminating, delivers the correct answer) and efficiently computable (having an algorithm that runs in $O(n)$ or $O(n \log n)$ time-or maybe even sometimes in $O(n^2)$ time) are central to successful analysis of complexity. We still need to classify many analytical cartography problems according to their degree of difficulty. Knowing computational complexity of problems and their algorithms can be very useful in deciding how to attack a new problem. We must handle "hard" problems differently. A new field of approximation theory is developing, and we should learn to use tools in that field. We cannot build algorithms or analyze complexity until we can model the problems mathematically.

References

[Baker 94] Baker B, 1994, Approximation algorithms for *NP*-complete problems on planar graphs. *Journal of the ACM*, 41:153-180.

[Blum 98] Blum L, Cucker F, Shub M, Smale S, 1998, *Complexity and Real Computation*, Springer, NY.

[Bugayevskiy 97] Bugayevskiy L, Snyder J, 1997, *Map Projections: A Reference Manual*, Taylor & Francis, London.

[Estrowski 98] Estkowski R, 1998, No Steiner point subdivision simplification is NP-complete. *Proc. 10th Canadian Conference on Computational Geometry.*

[Fowler 81] Fowler R, Paterson M, Tanimoto S, 1981, Optimal packing and covering in the plane are NP-complete. *Inform. Process. Lett.*, 12(3): 133-137.

[Garey 79] Garey M, Johnson D, 1979, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, CA.

[Green 77] Green P, Sibson R, 1977, Computing Dirichlet Tessellation in the Plane, *Comput. J.*, v.21, pp 168–173.

[Hochbaum 97] Hochbaum D, ed., 1997, *Approximation Algorithms for NP-Hard Problems*, PWS Publishing, Boston, MA.

[Iturriaga 97] Iturriaga C, Lubiw A, 1997, NP-hardness of some map labeling problems. Technical Report CS-97-18, University of Waterloo, Canada.

[Jones 95] Jones C, Bundy G, Ware J, 1995, Map generalization with a triangulated data structure. *Cartography and Geographic Information Systems*, 22(4): 317-331.

[Karp 74] Karp R, 1974, *Complexity of Computation*, vol. 4, SIAM-AMS Proceedings, American Mathematical Society, Providence, RI.

[Kato 88] Kato T, Imai H, 1988, The NP-completeness of the character placement problem of 2 or 3 degrees of freedom, in *Record of Joint Conference of Electrical and Electronic Engineers* Kyushu, p 1138.

[Knuth 90] Knuth D, Guibas L, Sharir M, 1990, Randomized Incremental Construction of Delaunay Triangulations and Voronoï Diagrams, in *Proceedings of ICALP*.

[Kao 91] Kao T, Mount D, Saalfeld A, 1991, Dynamic Maintenance of Delaunay Triangulations, in *Proceedings of Auto-Carto 10*, vol. 6, published by ACSM/ASPRS, Bethesda, MD, 219 - 233.

[Lewis 98] Lewis H, Papadimitriou C, 1998, *Elements of the Theory of Computation*. Prentice Hall, NJ.

[Marks 91] Marks J, Shieber S, 1991, The computational complexity of cartographic label placement, Technical Report TR-05-91, Harvard CS.

[McMaster 92] McMaster R, Shea S, 1992, *Generalization in Digital Cartography*, Association of American Geographers, Washington, DC.

[Müller 95] Müller J-C, Lagrange J-P, Weibel R, eds., 1995, *GIS and Generalization: Methodology and Practice*, Taylor & Francis, London.

[Traub 76a] Traub J, ed., 1976, *Analytical Computational Complexity*, Academic Press, NY.

[Traub 76b] Traub J, ed., 1976, *Algorithms and Complexity*, Academic Press, NY.

[Wagner 94] Wagner F, 1994, Approximate map labeling is in $\Omega(n \log n)$. *Information Processing Letters*, 52(3):161-165.

This article was processed by the author using the LaTeX style file *cljour2* from Springer-Verlag.