ECS 120 Lesson 28 – Approximation Algorithms, Randomized Algorithms

Oliver Kreylos

Wednesday, June 6th, 2001

In our discussion of time complexity, we have seen that there are problems that are solvable in principle, but require so much time and/or space to run, that they cannot in practice be solved for any but the smallest input sizes. But what should one do if facing such a problem in practice? Often, there are ways to get around the prohibitive time requirement by either making the problem slightly easier, or by accepting a solution that might be wrong in very few cases. Today we will address both these approaches.

1 Approximation Algorithms

As an early example of an $\mathcal{NP}$-complete problem, we have seen the VERTEX-COVER problem: Given a graph $G = (V, E)$, is there a vertex cover, i. e., a subset of nodes, that touches all edges in E and contains not more than k vertices, where k is a given constant? Posed as a language, this problem becomes

$$\text{VERTEX-COVER} = \left\{ \langle G, k \rangle \;\middle|\; G \text{ has a vertex cover of at most } k \text{ vertices} \right\}$$

Typically, this problem is asked in another form: Instead of asking whether some vertex cover exists, the task is to find the smallest possible vertex cover: Given a graph $G = (V, E)$, find a vertex cover of G having the smallest possible number of vertices. Typically, optimization problems like this one are even more difficult to decide than the related yes/no problems: VERTEX-COVER is $\mathcal{NP}$-complete, but MIN-VERTEX-COVER is $\mathcal{NP}$-hard, i. e., it is not even in $\mathcal{NP}$ itself.

But the problem can be solved quite easily if the requirement of finding the *optimal* solution is relaxed: In practice, it is often sufficient to find a solution that is not much worse than the theoretical best solution. We can formalize this notion by the following definition:

Definition 1 (k-optimal Approximation) *If P is a minimization problem, and A is a polynomial-time approximation algorithm for P that always returns a solution that is not more than k times optimal, then A is a* k-optimal *approximation algorithm for P.*

The following algorithm is 2-optimal for MIN-VERTEX-COVER:

2-MIN-VERTEX-COVER On input $\langle G \rangle$, where $G = (V, E)$ is a graph,

1. Find an unmarked edge $e \in E$ that does not touch, i. e., does not share a vertex with, any marked edge in E. If there is no such edge, go to step 4.
2. Mark the found edge e.
3. Go to step 1.
4. Output all vertices touching marked edges.

This algorithm definitely runs in polynomial time, and it also produces a correct vertex cover: If there were any edge in E that is not covered by a selected vertex, it cannot share any vertices with any marked edge. But then it would have been marked by step 2 of the algorithm, and both vertices touched by it would have been included in the vertex cover. That is a contradiction.

Let us now look at how many more vertices than necessary this algorithm selects: Let us denote the set of marked edges as M, the set of selected vertices as X, and some minimal vertex cover as C. Then X contains exactly twice as many vertices as M: For every edge in M, the algorithm selects two vertices. The selected vertices for two different edges must be different, because by step 1 no two marked edges share vertices. Thus, $|X| = 2 \cdot |M|$. Furthermore, the number of vertices in C must be at least as big as the number of edges in M: C is a vertex cover, so its vertices must cover all edges in E, and especially all edges in M. But since edges in M do not share vertices, one vertex in C can only cover one edge in M; therefore, $|C| \geq |M|$. Altogether, this means that $|X| \leq 2 \cdot |C|$, which means that the given algorithm is 2-optimal.

2 Probabilistic Algorithms

Another technique to cope with intractable problems is to construct an algorithm that is allowed to give a wrong answer, but only in very few cases. One way to formalize this idea is to construct a probabilistic Turing Machine.

A probabilistic Turing Machine (PTM) is very similar to a nondeterministic Turing Machine, in that its transition function δ allows it several choices to move on some combinations of state and tape character. A probabilistic Turing Machine has a computation tree exactly like an NTM, but instead of splitting computation on each branch, and executing all branches in parallel, the PTM "flips a coin" at each branching point and determines randomly which branch to execute. If the PTM reaches an accepting configuration, it accepts; if it reaches a rejecting configuration, it rejects.

From this definition follows, that each branch of a PTM's computation tree has a certain probability of being executed, and that the PTM's answer is wrong with a certain probability. To see why, we have to recall the definition of acceptance of NTMs: An NTM accepts a word w, if at least one branch of computation reaches an accepting configuration. It rejects if there is no accepting branch. a PTM, on the other hand, accepts if its random choices led it to an accepting configuration, and rejects otherwise.

Now let us assume that the computation tree of some NTM N on some input word w has accepting and rejecting branches. This means that N will accept w. The equivalent PTM P, however, can accept or reject w, depending on which branch was randomly selected. If we denote the probability of P ending up in a certain branch b as $\Pr(b)$, then the total probability of P accepting w is

$$\Pr(P \text{ accepts } w) = \sum_{\substack{b \text{ is an} \\ \text{accepting branch}}} \Pr(b) \ ,$$

and the probability of P rejecting w is

$$\Pr(P \text{ rejects } w) = \sum_{\substack{b \text{ is a} \\ \text{rejecting branch}}} \Pr(b) = 1 - \Pr(P \text{ accepts } w) \ .$$

In other words, P gives a wrong answer with probability $\Pr(P \text{ rejects } w)$.

Probabilistic algorithms can still be good enough for practical purposes if we find a probabilistic polynomial-time algorithm P, where the probability that P answers wrongly can be made as small as necessary. It is often possible to construct algorithms like this; one common example is deciding whether

a given number p is a prime or not. The brute-force algorithm to decide this checks if any integer between 2 and $\sqrt{p}$ divides p; if this is the case, p cannot be prime; otherwise, it must be prime. Since the amount of integers in the given range is exponential in the length of the encoding of p, this algorithm requires exponential time.

To convert this into a probabilistic algorithm with a small probability of failure, we need some tools from number theory:

Theorem 1 (Fermat's Little Theorem) *If p is a prime number, then $a^{p-1} \equiv 1 \pmod{p}$ for any $a \in \{2, 3, \ldots, p-1\}$.*

Theorem 2 *If p is an integer, and $a^{p-1} \not\equiv 1 \pmod{p}$ for some integer $a \in \{2, 3, \ldots, p-1\}$, then $a^{p-1} \not\equiv 1 \pmod{p}$ for at least half of the integers $a \in \{2, 3, \ldots, p-1\}$.*

These two theorems together can be used to probabilistically decide if a number p is prime: An algorithm could pick one number $a \in \{2, 3, \ldots, p-1\}$ at random, and check if $a^{p-1} \equiv 1 \pmod{p}$. If this is not the case, p must be a composite number, and the algorithm can reject. If it is the case, there might be two reasons: First, the number p could be prime, or second, the equivalence just holds by coincidence. But the latter case can only occur with a probability of less than 1/2: The second theorem says, that if one number violates it, at least half the numbers must. But then randomly picking a number that satisfies the equivalence only has a probability of less than 1/2.

An error probability of 1/2 is way too large for any practical purpose, but the algorithm can easily reduce it: If the random test is repeated k times, and p is not a prime number, then the probability of picking a "lucky" satisfying number is the product of the probabilities of picking one in each step – the random experiments are independent of each other. Thus, the probability of picking wrong k times is only 2^{-k}. By making k large, the error probability becomes arbitrarily small, and it decreases quickly: By repeating the test only 1000 times, the error probability is less than 2^{-1000}. The time complexity of each step is polynomial, so repeating it k times for constant k yields total polynomial running-time.

This algorithm still has one serious flaw: There are some non-prime numbers that have the nasty property of passing any modulo test of this kind; they are called *Carmichael Numbers*. The algorithm as stated will accept all primes, and also all Carmichael Numbers, with a probability of 1. To

filter out the Carmichael Numbers, the algorithm must be changed by using another fact from number theory.

Theorem 3 (Roots of Unity) *If a, b and p are integers, then a is said to be a* square root of b modulo p, *if and only if* $(a \cdot a) \equiv b \pmod{p}$.

If p is a prime number, then the number one has exactly two square roots modulo p, namely -1 and 1.

For many composite numbers, and especially all Carmichael Numbers, one has four or more square roots. To fix the given algorithm, we perform the "Fermat test" first, and if p passes it, we search roots of unity modulo p. This can be easily done: If $a^{p-1} \equiv 1 \pmod{p}$, then $p-1$ is an even number (all primes except 2 are odd), and $a^{(p-1)/2}$ must be a square root of 1: $(a^{(p-1)/2} \cdot a^{(p-1)/2}) \equiv a^{p-1} \equiv 1 \pmod{p}$. If $a^{(p-1)/2}$ is itself equal to 1, and $(p-1)/2$ is still even, we can create another root of unity by halving it again, repeating this step until the root is no longer one or the exponent becomes odd. Then the number $p-1$ can be written as $p = s \cdot 2^h$, where s is odd and $h \geq 1$. We can then compute a sequence of $h+1$ numbers by evaluating $a^{s \cdot 2^i}$ for $i = 0, 1, \ldots, h$. Every number in this sequence after the last number not equal to one is a root of unity, and, if p is prime, the last number not equal to one must be -1. All numbers before the last occurrence of -1 are not roots of unity, but roots of negative unity and must not be considered. The fixed algorithm using roots of unity works as follows:

Algorithm PRIMES On input $\langle p \rangle$, where $p \geq 2$ is an integer,

1. If p is even, accept if $p = 2$; otherwise, reject.
2. Repeat the following steps k times:
   (a) Randomly select $a \in \{2, 3, \ldots, p-1\}$.
   (b) Calculate $a^{p-1} \bmod p$. If this is not equal to 1, reject.
   (c) Find an odd s and $h \geq 1$ such that $p - 1 = s \cdot 2^h$.
   (d) Calculate the sequence of numbers $a^{s \cdot 2^i}$ for $i = 0, 1, \ldots, h$.
   (e) If some element of this sequence is not 1, find the last entry unequal to 1 and reject if that entry is not equal to -1.
3. Accept.

If p is a prime number, the algorithm will never reject, because p will pass all tests. On the other hand, if p is not a prime number, the algorithm will wrongly accept with a probability of less than 2^{-k}. By making k large, we can push the probability of wrongly accepting a composite number below any positive error tolerance $\varepsilon > 0$.

3 Probabilistic Optimization

One final technique that sometimes works very well for $\mathcal{NP}$-hard optimization problems is related to thermodynamics, of all fields. Let us examine a sample application of this method, related to data visualization. The task at hand is to approximate a function $f\colon [a,b] \to \mathbf{R}$ from an interval into the set of real numbers, see Figure 1, part (a). Often, functions are not given by a formula or arithmetic expression, but by a set of measured or otherwise computed function values, called *samples*. In this representation, a function is a finite set of (ordinate, function value) pairs:

$$f = \left\{ \big(x_i, f(x_i)\big) \,\middle|\, x_i \in [a,b] \text{ for } i = 1, 2, \ldots, N \right\}$$

If the ordinates x_i are randomly distributed over the interval $[a,b]$, this representation of functions is also called *scattered data*. An image of such a function representation is shown in Figure 1, part (b).

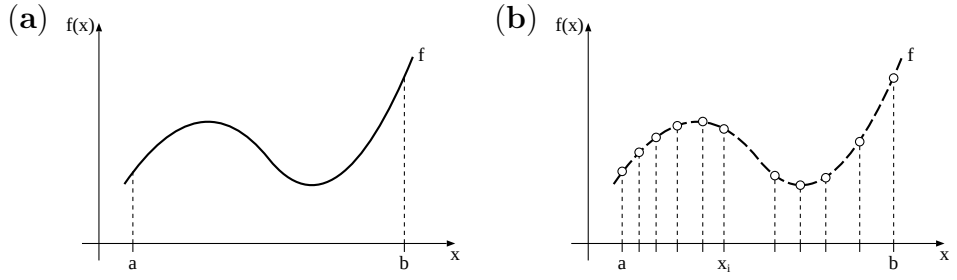

Figure 1: A function $f\colon [a,b] \to \mathbf{R}$ and its representation as scattered data.

Typically, the number N of samples is very large – much larger than necessary to represent f. We thus want to construct an approximation to f by selecting only a few samples $\big(x_i, f(x_i)\big)$ and connecting them by straight line segments, generating a piecewise linear approximation function a, see Figure 2. The problem is, for a given M, to find the subset of M samples that optimally approximates the given function f.

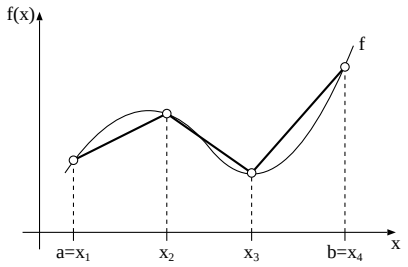

Figure 2: A piecewise linear approximation a to a function f over the interval $[a, b]$ using only four samples.

We judge the quality of an approximation by some error measure; a reasonable error measure would be the L^2-distance between a and f, calculated as $E(a) = \int_a^b \big(a(x) - f(x)\big)^2 \, dx$. This integral can be approximated by looking at the samples defining f only, without even knowing an analytical representation of f. Using these definitions, the problem becomes: Given a function f as a set of N samples $\big(x_i, f(x_i)\big)$, find a piecewise linear approximating function a using only M samples, such that the approximation error $E(a)$ is minimal. The brute-force approach to solve this problem would generate all possible subset of M samples, calculate the approximation error for each, and then select the best one found. Since there are $\binom{N}{M}$ different subsets of samples, and all have to be considered, this algorithm needs running-time exponential in N. With N typically being large, that algorithm is not practical. Instead, we use an iterative probabilistic optimization technique as follows:

Optimal Approximation On input $\langle f, M \rangle$, where f is a function defined by a set of samples and M is a positive integer,

1. Construct an initial approximation a_0 by selecting M samples from f randomly and connecting them to form a piecewise linear function.
2. Calculate the initial approximation error $E_0 = E(a_0)$.
3. For $i = 0, 1, \ldots, k$:
   (a) Create a new approximation function a_{i+1} by changing one of the samples selected for a_i: Remove one random sample, and insert another random sample from f.

(b) Calculate the approximation error $E_{i+1} = E(a_{i+1})$.

(c) If the difference $\Delta = E_{i+1} - E_i$ is negative, the change improved the approximation. Accept it.

(d) Otherwise, replace a_{i+1} by a_i and E_{i+1} by E_i.

This algorithm performs a *random walk* through the space of approximation functions: It starts with a random approximation, randomly changes the approximation in every step, and accepts a step if it improves the approximation. If this algorithm runs for a large number of steps, the final approximation will be a locally optimal one with a high probability. The behaviour of the algorithm can best be visualized in a one-dimensional optimization space: If the task is to find a minimum in a function $f: \mathbf{R} \to \mathbf{R}$, one could start searching at a random position x, move either left or right in each step, and accept the move if it decreased the function value $f(x)$. An illustration of such a walk is depicted in Figure 3.

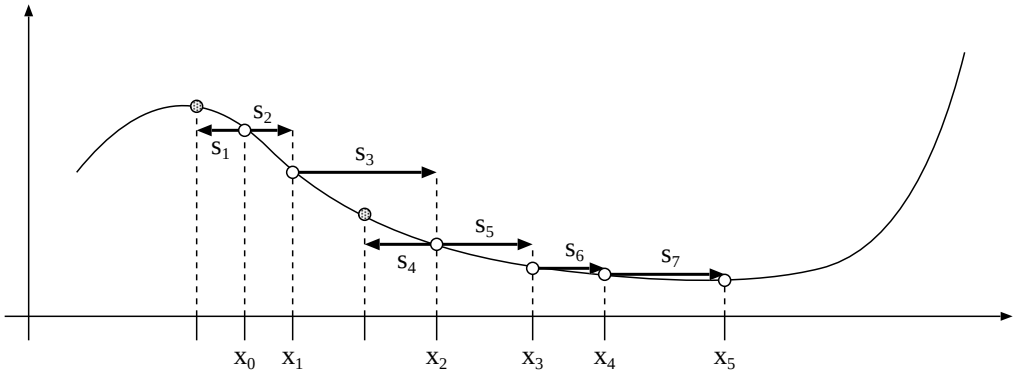

Figure 3: A random walk $x_0, x_1, \ldots, x_5$ to find the minimum of a function. The shaded circles denote rejected steps.

This algorithm has one major problem: It only accepts steps that decrease the error measure. This can lead to the random walk getting stuck in a local minimum that has a higher error measure than the global minimum. As it turns out, the shape of the error function $E(a)$ for function approximations is highly irregular: It typically has local minima in abundance – see Figure 4 for an illustration – and a good approximation algorithm has to move past them and find the true global minimum.

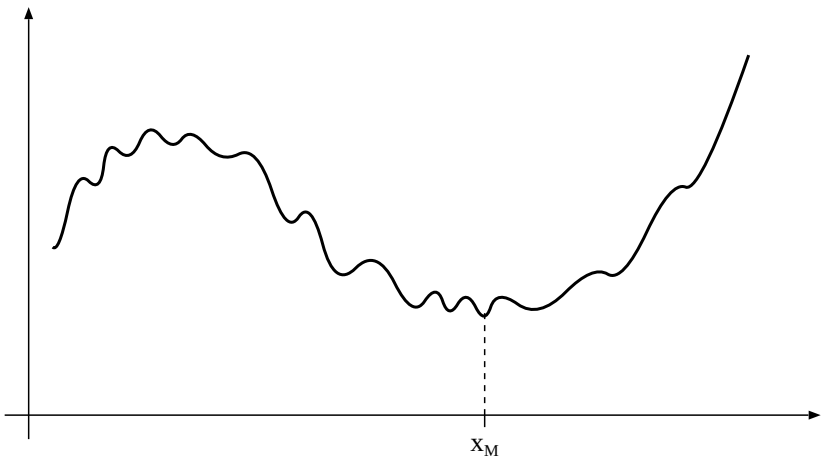

Figure 4: Typical shape of the error function for function approximation. The global minimum at x_M is surrounded by local minima in abundance.

The trick to overcome local minima is to allow the algorithm to accept a bad step *sometimes*. We perform this by accepting a step that increases the error measure by some positive Δ with a probability that approaches zero as Δ grows large: Pr(accepting an increase in error Δ) $= e^{-\Delta/kT}$, where kT is a constant called "temperature." This constant is initially set to some value, and then slowly decreased as the iteration runs. This allows the algorithm to accept some "bad" steps in the beginning, but in later stages of the iteration the probabiliy of accepting such steps approaches zero, and only good steps are taken, allowing the iteration to "converge" to a minimum.

This technique is called *simulated annealing*, and it has been used to great success in finding near-optimal solutions to $\mathcal{NP}$-hard optimization problems in reasonable amounts of time. The name is borrowed from the physical process of liquid metal solidifying to a "single crystal" when the temperature of the liquid is lowered slowly enough. In theory, this process can be viewed as an enormous optimization problem, where the energy optimum of the single crystal is hidden amongst many poorer crystallization structures. In physical annealing, the metal atoms somehow find the optimal structure by themselves; one reason for this is that a physical system can sometimes undergo a state change that increases its internal energy instead of decreasing it. The probability for this happening is $e^{-\Delta/kT}$, where Δ is the (positive) change in internal energy, T is the temperature of the liquid, and k is a physi-

cal constant called *Boltzmann's Constant*. This process works extremely well in nature, and its simulated version has been shown to work on optimization problems as well.