CS 59000 CTT Current Topics in Theoretical CS	Aug 21, 2012
Lecture 1	
Lecturer: Elena Grigorescu	*Scribe: Josh Cain*

1 Sublinear models of computation

The use of randomness in designing algorithms that deal with large data sets can lead to vast improvements in performance compared to deterministic algorithms for the same problems. In this course we will be looking at a few computational models that use randomness as a means to finding efficient or super-efficient algorithms for various computational problems specific to graphs, error-correcting codes, strings or functions.

What do we mean by 'super-efficient algorithms'? An algorithm that takes inputs of size n and whose running time is polynomial in n is said to be efficient. Typically, algorithms required to always output the correct answer must at least read their entire input, so their running time is at least linear in the size of the input. In practice, when dealing with huge data, say graphs with millions of nodes, even linear time algorithms are considered too slow. Can we do anything in time better than linear, that is with partial (i.e. local) information about the input? The answer to this question is yes, as long we are willing to live with approximate solutions to our problems. What do we really mean by approximate solutions? Of course, the answer depends on the computational model we look at, and it is often the case that even defining the right computational model for the tasks and the data that one is interested in can be a challenging task.

We start by describing some models of sublinear computation that not only have been successful in practice but that have also proved to be instrumental in many fundamental developments across theoretical CS.

1. **Program checking/correcting** This model originated in the 80's with the works of Blum and Kannan [1] and Blum, Luby, Rubinfeldand [2]. In this model one wants to quickly check that a given piece of software does compute the function that it is supposed to compute. One might try and do so by using the software as a 'black-box' and verifying that the program is correct on some carefully chosen inputs (instead of checking that each line of code is correct). For software computing complicated functions one might instead try to show that the function computed by the program satisfies a certain property. This model is the precursor of the Property testing model.

2. **Property testing** In the Property testing model one is interested in computing an approximate answer to a decision problem: does a given object have a property or does it differ by a lot from any object that has that property? The property testing model was defined by Rubinfeld and Sudan [3] and it has led to widely recognised successes in computational complexity, cryptography, learning theory.

   In this course we will focus on this model of computation. We will show that properties such as connectedness and triangle-freeness in graphs are testable in constant time! We will also look at properties such as monotonicity of functions and membership in error-correcting codes.

3. **Approximation algorithms** In this model one would like to output a solution to an optimization problem, for example for the MAX-CUT problem we want to compute the maximum number of edges crossing a partition of the graph. This is an NP-complete problem, so unless P=NP we can't expect to be able to solve it in polynomial time. But it turns out that there is a simple polynomial time algorithm that gives an approximation within a factor of 2 of the optimal answer. Approximation algorithms are not only interesting for NP-complete problems but they can be formulated for efficient problems for which we might want sublinear time algorithms. In this course we'll see several approximation algorithms, and in this lecture we will see an example of a deterministic sublinear-time algorithm that approximates the diameter of a set of points.

4. **Streaming model** In the streaming model the focus is on the space complexity of a problem, when the data arrives sequentially in an online fashion. This model has many applications when dealing with large data sets where one has little available storage. In this lecture we'll see an example of a technique used in this model.

5. **Local algorithms for error-correcting codes** Error-correcting codes are ways to encode data (by introducing redundant information) so that it can all be recovered even after corruption. They are widely used in communication scenarios and storage devices (CDs, DVDs). A typical task is that of decoding a received message that has been encoded using an error-correcting code and sent over a noisy channel. In some settings we don't actually need the entire message but we'd like to quickly recover only a small portion of it (for eg. one bit). Whenever this is possible the code is called locally decodable. In this course we'll describe some other tasks that can be super-efficient when our objects are error-correcting codes, namely testing membership and correcting.

6. **Communication complexity** In this model two parties want to quickly compute a function of their inputs. They can communicate by sending messages to each other, but since communication is expensive, they want to minimize the number of bits sent. For example, if they wish to compute the parity of their joint inputs, 2 bits are enough. Can you see why? On the other hand, if they want to figure whether they have the same input, one can show that any deterministic protocol can succeed only when the parties share almost all of their input bits. This model provides techniques for proving lower bounds for streaming algorithms, data structures and property testing. In this course we will see an application to property testing.

While the theme of the course is the use of randomness in computation mainly in order to build sublinear algorithms, along the way we will be introducing techniques and objects that are studied in many other contexts. In particular, expander graphs are fundamental combinatorial objects that are well-studied in mathematics, cryptography, complexity theory and coding theory. These are graphs that on one hand are sparse (have few edges), and on the other hand are very well connected. They have random-like graph properties which make them instrumental in the study of pseudorandomness in computation. In this course we'll see some of their properties and we'll show how they can be used to obtain codes with nice features.

2 A Review of Core Definitions

The following are some definitions that are fundamental to any discussion of algorithm complexity:

Definition 1 *Let $f, g : \mathbb{Z}^+ \to \mathbb{Z}^+$. We say $f = O(g)$ (f* is big-O of *g) if there exists $c > 0$ such that $f(n) \leq cg(n)$ for sufficiently large n. We also say that $f = \Omega(g)$ if $g = O(f)$ and that $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$. Additionally, we say that $f = o(g)$ if for all $c > 0$ we have $f(n) \leq cg(n)$ for sufficiently large n, and that $f = \omega(g)$ if $g = o(f)$.*

3 An Example of a Sublinear Time, Deterministic Algorithm

We shall construct an algorithm that approximates within a factor of 2 the diameter of a set of points, in sublinear time.

Definition 2 *Let S be a set of points. A function $D : S \times S \to \mathbb{R}$ is a* distance metric *if for all $x, y, z \in S$:*
1) $D(x, y) \geq 0$, and $D(x, y) = 0$ if and only if $x = y$
2) $D(x, y) = D(y, x)$, and
3) $D(x, y) \leq D(x, z) + D(z, y)$ (the triangle inequality)

Definition 3 *Given a set of points S with a metric D, the* diameter *of S is* $\text{diam}(S) = \max\{D(x, y) : x, y \in S\}$

Note that for a set S of size n, the metric D can be represented by an $n \times n$ matrix where $D(i, j)$ is the distance between points i, j. Thus to establish the diameter of the set exactly, it would take $N = n^2$ time, i.e. linear in the size of the input, which is N (the algorithm would simply examine each element of the matrix and output the largest.) The following result shows a method for finding an approximation to the diameter, which can be completed in sublinear time. Deteministic algorithms that run in sublinear time are very rare.

Theorem 4 (Indyk) *Given a set of n points S with a metric D, there exists a deterministic algorithm that outputs d s.t. $\frac{\text{diam}(S)}{2} \leq d \leq \text{diam}(S)$ in sublinear time (i.e. time $O(n) = o(|D|)$).*

Proof Suppose j and k are two points of S. We can find an upper bound on their distance by considering a third point $i \in S$ and applying the triangle inequality

$$\begin{aligned} D(j, k) &\leq D(i, j) + D(i, k) \\ &\leq 2\max\{D(i, j), D(i, k)\} \\ &\leq 2\max\{D(i, l) : l \in S\} \end{aligned}$$

Since j and k were arbitrary with respect to i, it follows that $D(j, k) \leq 2\max\{D(i, l) : l \in S\}$ for *any* $j, k \in S$; in particular; $diam(S) \leq 2\max\{D(i, l) : l \in S\}$. Thus, we need only select one arbitrary element $i \in S$ and check $D(i, l)$ for all other $l \in S$, setting d as the maximum of these distances. This ensures that $d \geq \frac{diam(S)}{2}$. Clearly $d \leq diam(S)$. ■

4 The Streaming Model

In the above result we showed a sublinear-time algorithm. We now turn to describing tasks that might require sublinear-space (memory) usage and we focus on the streaming model.

The motivation is as follows: Suppose we have a set S of n elements, and we are receiving a stream of these elements one at a time (in an online fashion.) We have a small number of bits available for storage, and we desire to construct algorithms that can get us as much information about the data stream as possible using only what little storage we have.

A simple example would be to find the maximum of a stream of numbers: we would store the first number in the stream, and at each step if the incoming number is larger than our stored number we would update the storage.

Some examples of the types of questions we could attempt to solve are: what is the average of the data, what is the median of the data, how long is the largest increasing sequence, and how many distinct elements are there in the stream.

For some of these more complicated questions (such as finding the median), the best we can do with our limited space is to store a random, uniform sample, then estimate a function on the sample and interpret the output as an estimate of the desired parameter. In certain cases it is possible to prove that this estimate is close to the true value of the parameter we seek to estimate.

Now, this leads to the question: how can one get a uniform sample from a stream of unknown length? We next describe a technique for solving this problem, known as *reservoir sampling* and due to Vitter, 1985.

Theorem 5 *There exists a probabilistic algorithm which, on input a stream* $\langle X_1, X_2, ..., X_t \rangle$ *outputs a value* S *such that for any* $1 \leq i \leq t$ *we have that* $Pr[S = X_i] = \frac{1}{t}$.

Proof We'll analyze the following algorithm:

1. Initialize $S \leftarrow X_1$
2. At time $t \geq 2$ replace S with X_t with probability $\frac{1}{t}$ (in other words, keep the previous value of S with probability $\frac{t-1}{t}$.)

To illustrate that this algorithm will produce the desired outcomes, first consider $Pr[S = X_1]$. At step 1, $Pr[S = X_1] = 1$ for obvious reasons. At step 2, we keep the previous value of S with probability $\frac{1}{2}$, so $Pr[S = X_1] = \frac{1}{2}$. At step 3, we keep the value of S from step 2 with probability $\frac{2}{3}$; since there is a $\frac{1}{2}$ probability that the old value was X_1, we now have $Pr[S = X_1] = \frac{1}{2}\frac{2}{3} = \frac{1}{3}$. Proceeding inductively, at step t we have $Pr[S = X_1] = \frac{1}{2}\frac{2}{3} \cdots \frac{t-1}{t} = \frac{1}{t}$ as desired. In the general, i.e. for an arbitrary $i \leq t$, we can disregard the action of the stream before the i^{th} step; hence $Pr[S = X_i] = \frac{1}{i}\frac{i}{i+1} \cdots \frac{t-1}{t} = \frac{1}{t}$.

To construct a sample of size k, this algorithm should be carried out k times and therefore it needs to store $k \log n$ bits (that is $\log n$ bits for one iteration.) ■

5 The Property Testing Model

As previously mentioned, in this model we would like to know whether an object has a property or is far from having it. Hence this is a relaxation of the typical decision problems where we need to decide if an object has a property or not. As previously argued, deterministic algorithms can be very demanding in terms of both time and space. In practice it might even be the case that data is changing fast, and by the time a deterministic algorithm has completed, the result is no longer relevant. The PT model is again a model where the use of randomness is crucial in producing fast algorithms, substantially better than their known deterministic analogues. For example, the problem of determining whether a graph can be colored with 3 colors is NP complete. However, there is a randomized algorithm that can decides if the graph is 3-colorable or is far from being 3-colorable that runs in constant time, thus independent of the size of the input!

A *property* P is a set of objects that share a common property. The objects that one could consider are graphs, functions, strings, distributions, error-correcting codes.

For example,
$P_n = \{$all graphs in n vertices that are colorable with 3 colors$\}$
$P_n = \{$all graphs in n vertices that are bipartite$\}$
$P_n = \{$all graphs in n vertices that are connected$\}$
$P_n = \{$all functions$\{f : \{0,1\}^n \to \{0,1\}$ that are linear.$\}$

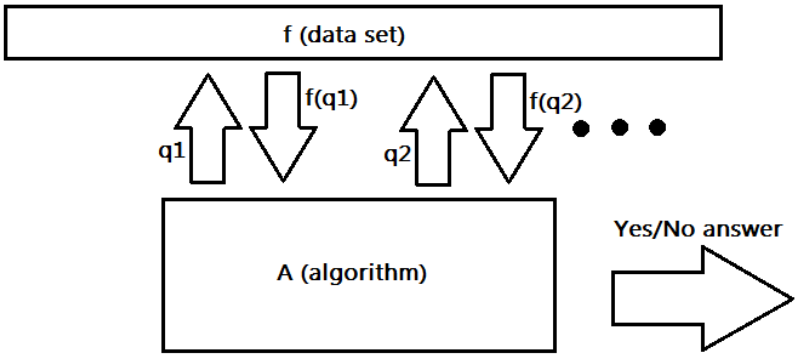
Notice that a graph $G(V, E(G))$ can be represented as a function over the domain $V \times V$ by $f(i,j) = 1$ if there is an edge between i and j, and $f(i,j) = 0$ otherwise. For this reason properties of graphs can also be expressed as collections of functions encoding the graphs that possess the property.

We will next formalize the Property testing model by referring to our data objects as functions over a finite domain.

Definition 6 *A property $P \subset \{h : D \to R\}$ (where D and R are the finite domain and range) is called* k-locally testable *if there exists a randomized algorithm A such that*

1. *on input $f : D \to R$ that A can access via queries*
2. *A makes k queries $q_1, q_2, \ldots, q_k$ to f*
3. *based on the answers it receives (namely the values $f(q_1), f(q_2), \ldots, f(q_k)$) A will output as follows:*
   (a) *If $f \in P_n$, then A says yes with probability $\frac{2}{3}$;*
   (b) *If f is far from P_n then A says no with probability $\frac{2}{3}$.*

See Figure 5 for an illustration of the interaction between the algorithm and the oracle f.



The running time of the algorithm is often about the same as its query complexity (i.e. $O(k)$), which is the main parameter in the above definition. We are interested in $k = o(n)$ or even better, we could ask for k that does not depend at all on n (that is k is a constant).

The notion of distance used in the previous definition depends on the objects and the tasks that we might be looking at.

A typical notion of distance between two functions (strings) is the (relative) Hamming distance:

Definition 7 *The* relative Hamming distance *between* $f, g : D \to R$ *is*

$$\delta(f, g) = \frac{|\{x : f(x) \neq g(x)\}|}{|D|}.$$

The distance from f *to a property* P *is then* $\delta(f, P) = \min_{g \in P} \delta(f, g)$

For graphs this notion of distance corresponds to the fraction of edges that one would need to add or delete in order to obtain a graph with the desired property.

As a simple illustration of a testable property, consider the property $P_n = \{f(x) = 1 : x \in [n]\}$. This can be equivalently expressed as $P_n = \{\langle 1^n \rangle\} = \{\langle 1, 1, \ldots, 1 \rangle\}$. We'll use the latter representation in what follows. For a given $\epsilon > 0$ and a bit sequence S of length n, we want to distinguish between the case where $S = 1^n$ and the case where S has more than an ϵ fraction of 0 entries. We claim that P_n is $\frac{2}{\epsilon}$-locally testable. Consider the following simple algorithm:

1. select $\frac{2}{\epsilon}$ entries at random (note that this is independent of n) and check their values
2. if any of the values is 0 output no (i.e. reject the input, meaning say $S \notin P_n$.) Otherwise, output yes (i.e. accept)

Let us analyse the running time. Clearly this is $O(2/\epsilon)$ which is independent of n (so it is a constant).

Now, if $S = 1^n$, the probability of accepting is clearly 1. If instead S is ϵ-far from 1^n, then $Pr[A \text{ finds a } 0 \text{ at any step } i] \geq \epsilon$.

Therefore, $Pr[\text{error at output}] = Pr[\text{no 0's were found in any of the } \frac{2}{\epsilon} \text{ steps}] \leq (1 - \epsilon)^{\frac{2}{\epsilon}} \leq e^{-2} < \frac{1}{3}$ (where we used that $1 - \epsilon < e^{-\epsilon}$).

References

[1] Manuel Blum and Sampath Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.

[2] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. Syst. Sci.*, 47(3):549–595, 1993.

[3] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM J. Comput.*, 25(2):252–271, 1996.