CSC2420: Algorithm Design, Analysis and Theory Nov 3, 2010

Lecture Notes 8: Sublinear Algorithms

Professor: Allan Borodin *Scribe: Dai Tri Man Lê*

Before we start with sublinear algorithms, we will finish the proof that we promised in the previous lecture.

1 Proof of $(k+1)$-approximation for maximizing a submodular set function on k-independence set system

Theorem 1. *The natural greedy algorithm achieves $(k+1)$-approximation for the problem of maximizing a submodular set function $f(S)$ subject to S being independent in a k-independence set system (k-matroid).* ♦

This result was originally attributed to Nemhauser, Wolsey, and Fisher [4], but they only claimed this result without a proof. Jenkyns was the first to give a full proof of this result. We will discuss a simplified proof due to Calinescu et al [1].

Proof. If we have $(\alpha+1)$-approximation for incremental oracle then approximation bound is $\alpha k+1$. Let $\beta \leq 1$ and $\beta = 1/\alpha$. Then we want

$$f(Greedy) \geq \frac{\beta}{k+\beta} f(OPT).$$

Assume the sequence of sets produced as each iteration of the greedy algorithm is

$$S_1, S_2, \ldots, S_t,$$

where $S_i = S_{i-1} \cup \{e_i\}$ and $S_t = Greedy$. We let

$$\delta_i := p_{e_i}(S_{i-1}) = f(S_i) - f(S_{i-1})$$

denote the improvement in the solution value when e_i is added into S_{i-1}.

The goal for us now is to construct a partition of $OPT = O_1 \cup O_2 \cup \ldots \cup O_t$ so that we can charge profit in O_i to δ_i. For this purpose, we need the following important *Charging Lemma*.

Lemma 1 (Charging Lemma). *We can construct a partion of $OPT = O_1 \cup O_2 \cup \ldots \cup O_t$ such that*

$$\delta_i \geq \frac{\beta}{k} f_{O_i}(S_t)$$

for every $1 \leq i \leq t$. ♦

Sketch of proof (of Charging Lemma). Then we can construct the partition of O using the following algorithm.

Algorithm:

$T_t \leftarrow 0$
for $i = t$ to 2 **do**
  $B_i \leftarrow \{e \in T_i \mid S_{i-1} \cup \{e\}$ is independent$\}$
  if $|B_i| \leq k$ **then**
    $O_i \leftarrow B_i$
  else
    O_i is any k-subset of B_i.
  end if
  $T_{i-1} = T_i \setminus B_i$
end for
$O_1 = T_1$

Clearly, $|O_i| \leq k$ for $i \geq 2$. Thus it only remains to show $|O_1| \leq k$. For this purpose, they showed by induction on $i = t, \ldots, 1$ that $|T_i| \leq i \cdot k$. The proof needs the k-independence property even in the base case. For a more detailed proof of this lemma, the reader is referred to [1, page 28]. □

Using the Charging Lemma, we can complete the proof of the theorem as follows.

$$\begin{aligned} f(Greedy) = f(S_t) &= \sum_{i=1}^{t} \delta_i \\ &\geq \frac{\beta}{k} \left(\sum_{i=1}^{t} f_{O_i}(S_t) \right) \\ &\geq \frac{\beta}{k} f_{OPT}(S_t) && \text{(by submodularity)} \\ &\geq \frac{\beta}{k} \left[f(OPT) - f(Greedy) \right] && \text{(by submodularity)} \end{aligned}$$

□

2 An overview of computational complexity

The content of this "bonus" section is adapted from a distinguished Fields lecture given by Avi Wigderson in Sept 14th, 2010. Since we use randomness extensively when designing sublinear algorithms, it would be useful to review the role of randomness in computational complexity theory. We also review briefly the P vs. NP question.

2.1 Hard vs. easy problems

Consider the problem of multiplying two integers. This is clearly an easy problem since the simple grade school multiplication algorithm can return the product of two two n digit numbers in $O(n^2)$ steps. Thus a problem is *easy* if it can be solved by a polynomial time algorithm, i.e., algorithm that

runs in polynomial steps with respect to the size of the input. The class of all problems that can be solved using a polynomial time algorithm is denoted by P.

Next consider the FACTORING problem, where we want to split an integer into two smaller non-trivial divisors, which when multiplied together equal the original integer. The best known algorithm for factoring takes $O(exp(\sqrt{n}))$ steps on an n digit input. It is open whether factoring has a polynomial time algorithm or not. So we don't know if factoring is a hard problem. However, it is widely believe that factoring is hard since the theory of public key cryptography depends on the hardness of factoring. We call a problem *hard* if there is no polynomial time algorithm solving it.

2.2 The P vs. NP question

Consider the MAP COLORING problem, where we take input as a planar map M with n countries. We observe the following questions:

- 2-COLORING: Is M 2-colorable?
- 3-COLORING: Is M 3-colorable?
- 4-COLORING: Is M 4-colorable?

The 2-COLORING problem is clearly easy since we answer using a simple greedy algorithm. The 4-COLORING problem is extremely easy since the answer is always yes for every planar graph by the *four color theorem*. However, it remains unknown if the 3-COLORING problem is easy or not, but we know the following theorem:

Theorem 2 (Cook-Levin '71, Karp '72). 3-COLORING *is* NP*-complete.* ♦

Here NP stands for *nondeterministic polynomial time*. Intuitively, NP is the class of problems whose solutions can be verified easily. For example, for 3-COLORING, once given a coloring of the map M, we can easily check in polynomial time if that coloring is valid by checking if any two adjacent countries have different color.

A problem in NP is called NP-complete if and only if we can reduce any other NP problems to it by a polynomial time transformation of the inputs. In fact, many problems in all sciences and engineering are NP-complete.

Note that we know FACTORING is in NP since we can easily check if the product of two integers is equal to another integer. But it remains open whether or not FACTORING is NP-complete.

The most fundamental question, i.e., the P vs. NP question, in computational complexity can be stated as following: is the 3-COLORING problem (or any other NP-complete problem) easy? Most of complexity theorists believe that all NP-complete problems are hard, i.e., they believe that

Conjecture 1. $\mathsf{P} \neq \mathsf{NP}$. ♦

2.3 The power of randomness in saving time

The main idea is to introduce randomness into the polynomial time algorithm, and we only require good probabilistic algorithm to succeed with high probability, e.g., with 99.99% probability. But one might ask why we tolerate errors? The reasons are:

- We tolerate errors in life.

– We can make the probability of errors exponentially small by repetition.
– To compensate, we can achieve much more...

We will next see two famous problems, for which we have *probabilistic polynomial time* algorithms, but no deterministic polynomial time algorithms are known.

PRIMES. Consider the following problem asked by Gauss: given $x \in [2^n, 2^{n+1}]$, is x prime? Two simple and fast probabilistic algorithms were invented in 1975 by Solovay-Strassen and Rabin. In fact, Solovay-Strassen and Rabin are the first to introduce randomness into algorithms. A later breakthrough is the discovery of polynomial time deterministic algorithm, aka AKS algorithm, for primality testing in 2002 due to Agrawal, Kayal and Saxena, but the AKS algorithm is not as efficient, and thus rarely used in practice.

However, no deterministic polynomial time algorithm is known for the following closely related problem: given n, find a prime in $[2^n, 2^{n+1}]$. But we can solve the problem using the following simple probabilistic algorithm: pick at random a sequence of random numbers $x_1, x_2, \ldots, x_{100n}$ from $[2^n, 2^{n+1}]$, and for each x_i apply primality test. By the *prime number theorem*, we can easily show that that $\mathsf{Prob}[\exists i, x_i \text{ is a prime}]$ is very high.

POLYNOMIAL IDENTITIES. For example, we want to check if

$$det\left(\begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \ldots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix}\right) - \prod_{1 \le i < j \le n} (x_j - x_i) \equiv 0?$$

We know from a theorem by Vandermonde, the answer is yes. But assume that we don't know this theorem, how do we check if this identity is true?

In general, given (implicitly, e.g. as a formula) a polynomial $p(x_1, \ldots, x_n)$ of degree d, we want to known if $p(x_1, \ldots, x_n) \equiv 0$. The following probabilistic algorithm by Schwartz-Zippel from '80 solves the problem: pick r_i independently at random from $1, 2, , 100d$. Then

$$p \equiv 0 \Rightarrow \mathsf{Prob}[p(r_1, \ldots, r_n) = 0] = 1$$
$$p \not\equiv 0 \Rightarrow \mathsf{Prob}[p(r_1, \ldots, r_n) \neq 0] > .99$$

Again no polynomial time deterministic algorithm is known for this problem.

This leads us to the second most fundamental question of computational complexity. We want to know whether or not randomness helps in saving time. In other words, are there problems with probabilistic polynomial time algorithm but no deterministic one? One might conjecture that:

Conjecture 2. There exists a problem that can be solved with a probabilistic polynomial time algorithm but not with a deterministic polynomial time one. Formally, $\mathsf{BPP} \neq \mathsf{P}$. ♦

Surprisingly, recent progress in computational complexity suggests that Conjecture 2 might be false! There are, essentially, two general arguments to support this belief that $\mathsf{BPP} = \mathsf{P}$. The first

argument is empirical: a large number of randomized algorithms use existing random sources (with no guarantees as to their properties) and the results do not seem to be adversely impacted. It is also the case that many randomized algorithms have been dererandomized without access to any source of randomness. The second argument (and perhaps more compelling argument) is that, under some plausible (what many believe to be more plausible than $\mathsf{BPP} \neq \mathsf{P}$) complexity assumption (namely, that there exists a problem, say SAT, in exponential time that requires exponential size circuits), we can build devices, called *pseudorandom generators*, which can be used to derandomize any probabilistic polynomial time algorithm to get a deterministic polynomial time one. To learn more about the area of *derandomization*, the reader is referred to the surveys [3,2].

3 What are sublinear algorithms?

By sublinear algorithms, we mean *sublinear time algorithms* and *sublinear space algorithms*. It seems that what can be achieved must be simple since we can't see the whole input in sublinear time/space. To solve more nontrivial problems, we allow approximation and the use of randomness in the computation. Note that, in contrast to the complexity theory world, we seem to need randomness in the design of sublinear time/space algorithms.

Sublinear space algorithms. For sublinear space algorithms, we assume that the Turing machine has a read-only tape, a sublinear working space, and a write-only output tape. The "gold standard" is $O(\log n)$ working space.

Similar to time complexity, we can also define complexity classes in term of working space. We let L denote the class of problems that can be solved by a deterministic machines using $O(\log n)$ space. The class L also has complete problem with respect to log-space reducibility. An interesting complete problem for L is the following UNDIRECTED REACHABILITY problem: given a *undirected* graph and two vertices on the graph s and t, decide wether or not t is reachable from s?

We let NL denote the class of problems that can be solved by a *non-deterministic* machines using $O(\log n)$ space. The class NL also has complete problem with respect to log-space reducibility. The standard complete problem for NL is the DIRECTED REACHABILITY problem: given a *directed* graph and two vertices on the graph s and t, decide wether or not t is reachable from s?

The only connection we know between L and NL is the following theorem due to Savitch. Let $\mathsf{DSPACE}(\log^2 n)$ denote the class of problems that can be solved by deterministic machines using $O(\log^2 n)$ space. Then Savitch's theorem can be stated as following.

Theorem 3 (Savitch). $\mathsf{NL} \subseteq \mathsf{DSPACE}(\log^2 n)$. ♦

However, it remains open whether $\mathsf{NL} \neq \mathsf{L}$. The class NL and its complement coNL exhibit a very surprising relationship.

Theorem 4 (Immerman-Szelepcsényi). $\mathsf{NL} = \mathsf{coNL}$. ♦

This theorem is counter-intuitive since it seems that we have to check all possible paths of the non-deterministic computation to decide a coNL problem.

One important subclass of sublinear algorithms are *stream algorithms*. For stream algorithms, we look at input as a stream of items, and we can't backtrack on input, and we can use only a "small" working space, where "gold standard" is $O(\log n)$ space. Again to do more useful things with stream

algorithms, we usually allow randomness in our computation. The main goal is to study when we can and cannot get answers with stream algorithms, i.e., we want to prove positive (upper-bound) and negative (lower-bound) results about this model.

Sublinear time algorithms. An algorithm is *sublinear time* if its running time is $o(n)$. As such an algorithm must provide an answer without reading the entire input. Thus to achieve non-trivial tasks, we also allow randomness in sublinear time algorithms.

4 Sublinear time algorithm

For the rest of this lecture, we will focus only on sublinear time algorithms. We will discuss some interesting problems that can be solved (or approximated) using sublinear time algorithms

4.1 Diameter of a metric space

Problem: Given points $x_1, \ldots, x_n$ in an arbitrarily metric space M and distances $d(x_i, x_j)$ satisfying the following condition:

$$\begin{aligned} d(x,x) &= 0 \\ d(x,y) &= d(y,x) \\ d(x,z) &\leq d(z,y) + d(y,z) \end{aligned}$$

Goal: compute $\mathsf{diameter}(M) = \max_{x,y \in M} d(x,y)$. ♦

Note that since we have $O(n^2)$ distances, the input "genuinely" has $O(n^2)$ items. We claim that the following algorithm $O(n)$-time algorithm guarantees a 2-approximation for this problem: choose $x \in M$ arbitrarily (i.e., $x = x_1$), and then let $d' = \max_y d(x,y)$. We next prove this claim.

Proof. We recall that $\mathsf{diameter}(M) = d(u,v)$ for some points u and v. And the property of distances, we have

$$d(u,v) \leq d(u,x) + d(x,v).$$

Thus, we have $d' \leq \mathsf{diameter}(M) \leq 2d'$ as desired. □

4.2 Searching in a sorted linked-list

Problem: Assume that n elements are stored in a linked-list, where each list element has access to the next element of the list, and the linked-list is sorted (i.e, if x follows y in the linked-list, then $y < x$). We also assume that all n linked-list elements are stored in an array A (but the array is not sorted and we do not impose any order for the array elements).
Goal: given x, determine if $x \in A$, and if so output i such that $x = A(i)$. ♦

Algorithm: If we know where list starts, says at $A(i)$, then we search $A(i), \ldots, A(i + 2\sqrt{n})$. If x is found, then report. Else, choose $\sqrt{n}$ random j's and search $A(j_1), \ldots, A(j_{\sqrt{n}})$ to get an i such that

$$A(i) = \max_{\substack{\ell \in \{j_1, \ldots j_{\sqrt{n}}\} \\ A(i) \leq x}} A(i).$$

Then, starting from the linked list node stored at $A(i)$, search forward on the linked list for $2\sqrt{n}$ steps. If found, then report success, else report failure. ♦

To show that the algorithm is a correct randomized algorithm with one-sided error for our problem, we want to show the following proposition.

Proposition 1. *For all x, we have*

1. *if $x \in A$, then with probability at least $6/7$ the algorithm will find i such that $A(i) = x$.*
2. *if $x \notin A$, then the algorithm will always report failure.*

Proof. The condition (2) is clear since if x is not on the list, then the algorithm will never find any i such that $A(i) = x$, and the algorithm will always fail to find x. It remains to show condition (1). Assume $x \in A$. Since the algorithm basically samples intervals,

$$\mathsf{Prob}[x \text{ is not in any of the randomly chosen intervals}] \leq (1 - t/n)^{\sqrt{n}},$$

where t is the size of each chosen interval and for our algorithm $t = 2\sqrt{n}$. Before proceeding, we need the following facts that we use very often when analyzing randomized algorithms.

Fact 1.

$$\left(1 - \frac{1}{m}\right)^m < \frac{1}{e} \qquad \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m = \frac{1}{e}$$

Thus, we have

$$\begin{aligned} \mathsf{Prob}[x \text{ is not in any of the randomly chosen intervals}] &\leq \left(1 - \frac{2\sqrt{n}}{n}\right)^{\sqrt{n}} = \left(1 - \frac{2}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{2} \cdot 2} \\ &< \left(\frac{1}{e}\right)^2 < \frac{1}{7}. \end{aligned}$$

Thus the probability that the algorithm will find x is at least $6/7$. □

4.3 Estimating the average degree of a graph

Problem: Given a graph $G = (V, E)$ and $|V| = n$, we want to estimate the average degree d of all vertices of G. ♦

We want to construct an algorithm that approximates the average degree with approximation ratio less than $(2 + \epsilon)$ with probability at least $3/4$ in time $O(\sqrt{n}/\epsilon)$.

Let d_i denote the degree of vertex v_i and we know that $d_i \in \{1, \ldots, n-1\}$. We assume that we have an oracle access to d_i. We will consider a weaker result when the algorithm will take $O\big(\sqrt{n}/(\epsilon^{2.5})\big)$. We claim the following algorithm will achieve the approximation ratio less than $(2+\epsilon)$ with probability at least $3/4$.

Algorithm:

for $i = 1$ to $8/\epsilon$ **do**
  Pick a set S_i of vertices at random such that $|S_i| = s = \sqrt{n}/(\epsilon^{2.5})$
  Compute average degree d_{S_i} of vertices in S_i
end for
Output $\min_i d_{S_i}$

To prove the correctness of this algorithm, we need the following lemma.

Lemma 2. *Let S be any S_i, then*

1. $\mathsf{Prob}\big[d_S > (1+\epsilon)d\big] < 1 - \dfrac{\epsilon}{2}$
2. $\mathsf{Prob}\left[d_S > \frac{(1+\epsilon)d}{2}\right] < \dfrac{\epsilon}{64}$

From this lemma, we can show the correctness of our algorithm (ALG) as follows.

Proof. From Part (1) of the lemma, it follows that

$$\begin{aligned}
\mathsf{Prob}\big[ALG > (1+\epsilon)d\big] &= \mathsf{Prob}\big[d_{S_i} > (1+\epsilon)d, \text{ for all } i\big] \\
&= \mathsf{Prob}\big[d_S > (1+\epsilon)d\big]^{8/\epsilon} \\
&< \left(1 - \frac{\epsilon}{2}\right)^{8/\epsilon} \leq e^{-4} && \text{(by Fact 1)} \\
&\leq 1/8
\end{aligned}$$

Also, we have

$$\begin{aligned}
\mathsf{Prob}\left[ALG < \frac{(1+\epsilon)d}{2}\right] &= \mathsf{Prob}\left[d_{S_i} < \frac{(1+\epsilon)d}{2}, \text{ for some } i\right] \\
&\leq \sum_i \mathsf{Prob}\left[d_{S_i} < \frac{(1+\epsilon)d}{2}\right] && \text{(by union bound)} \\
&\leq \frac{8}{\epsilon} \cdot \frac{\epsilon}{64} = \frac{1}{8}
\end{aligned}$$

We note that if the algorithm fails to get an estimation, then

$$ALG > (1+\epsilon)d \text{ or } ALG < \frac{(1+\epsilon)d}{2}.$$

By the union bound, the probability of at least one of these two events happens is at most $1/4$. Thus the probability the algorithm fails to get an estimation is bounded by $1/4$. □

Before proceeding with the proof of Lemma 2, we need some more important tools from probability theory.

Theorem 5 (Markov inequality). *Given a random variable $Z \geq 0$ and $E[X] = \mu$. Then*

$$\mathsf{Prob}[Z > b \cdot \mu] \leq 1/b$$

for every $b \geq 1$. ♦

Theorem 6 (Chernoff's bound). *Let $Z \in \{0, 1\}$ be a random variable such that $E[Z] = \mu$. Let $Z_1, \ldots, Z_T$ be independent "trials" of Z. Then*

$$\mathsf{Prob}\left[\sum_{i=1}^{T} Z_i \leq (1-\delta)\mu T\right] \leq e^{-\delta^2 T\mu/4}.$$

♦

Proof (Part (1) of Lemma 2). Define the random variable X_i to be the degree of vertex v_i that is chosen. Thus, we have $E[X_i] = d$. We also observe that

$$\begin{aligned} E[d_S] &= E\left[\frac{1}{|S|}\left(\sum_{i:v_i \in S} X_i\right)\right] \\ &= \frac{1}{|S|}\sum_{i:v_i \in S} E[X_i] && \text{(by linearity of expectation)} \\ &= d \end{aligned}$$

Thus, we can apply Markov inequality to get

$$\mathsf{Prob}\big[d_S > (1+\epsilon)d\big] \leq \frac{1}{1+\epsilon} < 1 - \frac{\epsilon}{2},$$

for sufficiently small ϵ. □

References

1. Gruia Calinescu, Chandra Chekuri, Martin Pal and Jan Vondrak. Maximizing a submodular set function subject to a matroid constraint. To appear in SIAM Journal on Computing, special issue for STOC 2008.
2. Oded Goldreich. Pseudorandom generator: a primer. University Lecture Series, vol. 55, AMS, 2010. Also available at: http://www.wisdom.weizmann.ac.il/ oded/prg-primer.html
3. V. Kabanets. Derandomization: a brief overview. BEATCS, Number 76, pages 88-103, 2002. Also available at: http://www.cs.sfu.ca/ kabanets/Research/beatcs.html
4. G. Nemhauser, L. Wolsey, and M. Fisher. An analysis of the approximations for maximizing submodular set functions. Mathematical Programming (14): 265-294, 1978.