1 Sub-linear time algorithms

In last lecture, we looked at some of the problems that can be solved (or approximated) using sub-linear time algorithms:

- Diameter of a metric space
- Searching in sorted linked-list
- Estimating the average degree of a graph (incomplete)

1.1 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 $G$.

The $O(\sqrt{n}/ε^{2.5})$ time algorithm presented in last lecture computes an estimate within a factor $\sim 2$ with sufficiently high probability. As the case with most sub-linear time algorithms, presentation of this algorithm is also simple but the analysis is not trivial.

Algorithm [1]

```plaintext
for i = 1,..., 8/ε do
    Pick a set $S_i$ of $s = \sqrt{n}/(ε^{2.5})$
    Compute $d_{S_i}$ = average degree of vertices in $|S_i| = s = \sqrt{n}/(ε^{2.5})$
end for
Output $\min_i d_{S_i}$
```

To prove the correctness of this algorithm we will prove the following claims:

Let $d$ be the true average degree and $S$ be one of these $S_i$

- **Claim 1:** $\text{Prob}[d > (1 + ε)d] \leq 1 - \frac{ε}{2}$ (proved in last lecture)
- **Claim 2:** $\text{Prob}[d < \frac{1}{2}(1 - ε)d] \leq \frac{ε}{84}$

**Theorem (Chernoff’s bound):** Let $Z_1, ..., Z_n$ independent “trials” of $Z$. Let $Z_i \in \{0, 1\}$ and $Z = \sum_{i=1}^{n} Z_i$ and $\mu = E[Z] = E[\sum_{i=1}^{n} Z_i]$. Then

$$\text{Prob}\left[\sum_{i=1}^{n} Z_i < (1 - ε)\mu\right] \leq e^{-\muε^2/4}$$  \hspace{1cm} (1)

**Proof of Claim 2:** Let $H$ be $\sqrt{εn}$ vertices of highest degree in the graph. Assume that the random selection of samples is done from $L$ where,

$$L = V - H$$  \hspace{1cm} (2)
By removing high degree vertices from random samples the probability of obtaining an average degree \( d_S < \frac{1}{2}(1 - \epsilon)d \) goes up. Now, the expected value of \( d_S \) when sampling from \( L \) is,

\[
E[d_S] \geq \frac{1}{2} \left( \frac{d_s |V| - \left( \frac{H}{2} \right)}{|L|} \right) = \frac{1}{2}(d - \epsilon)
\]  

(3)

Therefore,

\[
Prob \left[ d_S < \frac{1}{2}(1 - \epsilon)d \right] = Prob \left[ d_S < (1 - \epsilon)E[d_S] \right]
\]  

(4)

Let \( x_i \) be the degree of vertex chosen,

\[
Prob[d_S < (1 - \epsilon)E[d_S]] = Prob \left[ \frac{\sum x_i}{d_H} \leq (1 - \epsilon)E \left[ \frac{\sum x_i}{d_H} \right] \right] \leq e^{-\epsilon^2 s.E[x_i]/d_H} \quad \text{(Chernoff's bound)}
\]  

(6)

If \( s \geq \epsilon^{-2} \frac{d_H}{E[x_i]} \) we will be done; but we want our bound without knowing \( d_H \). There are two cases:

- **Case 1:** \( d_H \geq \frac{|H|}{\epsilon} \)

\[
E[x_i] = \sum_{v \in L} \frac{d(v)}{|L|} \geq \frac{|H|d_H - |H|^2}{|L|} \geq \frac{|H|(1 - \epsilon)d_H}{|L|}
\]  

\[
\Rightarrow \frac{d_H}{E[x_i]} \leq \frac{|V|}{|L|} \quad (|V| > |L|)
\]  

(10)

\[
= \frac{n}{\sqrt{en}}
\]  

(11)

Thus,

\[
s \geq \epsilon^{-2} \epsilon^{-1/2} \sqrt{n}
\]  

(12)

- **Case 2:** \( d_H < \frac{|H|}{\epsilon} \)

\[
\epsilon^{-2} \frac{d_H}{E[x_i]} \leq \frac{\epsilon^{-2}}{\epsilon} |H| \leq \epsilon^{-3} \sqrt{en} \quad \text{(14)}
\]  

\[
= \epsilon^{-2.5} \sqrt{n}
\]  

(15)

1.2 **Property testing**

**Definition:** “Given the ability to perform (local) queries concerning a particular object (e.g., a function, or a graph), the task is to determine whether the object has a predetermined (global) property (e.g., linearity or bipartiteness), or is far from having the property. The task should be performed by inspecting only a small (possibly randomly selected) part of the whole object, where a small probability of failure is allowed [2].”

**Property testing** grew out of **program testing**. In **program testing** the goal is to check whether the program computes a specified function. One can test whether a program satisfies a certain property before checking whether the program computes a specified
function. This paradigm has been followed both in theory of program testing and in practice through debugging. Different types of problems are studied in the context of property testing: graph properties, algebraic properties of functions, string properties, clustering, properties of boolean functions and more [2].

1.2.1 Testing an array for monotonicity

Goal: Given an array of length \( n \) with distinct values, test whether it is monotone or \( \epsilon \)-far away from monotone [3].

Algorithm

\[
\begin{align*}
\text{for } O(1/\epsilon) \text{ trials do} & \\
& \quad \text{Randomly choose } j \text{ where } 1 \leq j \leq n \text{ and let } v_j = A[j] \\
& \quad \text{Perform a binary search to determine whether } v_j \text{ is in } A \\
& \quad \text{if not found report } A \text{ is not monotone} \\
\text{end for} & \\
& \text{report } A \text{ is monotone}
\end{align*}
\]

The complexity of algorithm is \( O((1/\epsilon) \log n) \).

Let \( S \) be a set of successful searches.

Lemma: \( S \) is a monotone sub-sequence.

Proof: Given, \( i < j \) and \( i, j \in S \), at some point the binary search for \( v_i \) must diverge from the binary search for \( v_j \). Let \( k \) be that point then at \( k \),

\[
\begin{align*}
A(i) & \leq A(k) \\
A(k) & \leq A(j)
\end{align*}
\]

This implies that,

\[
A(i) \leq A(j)
\]

Therefore, \( S \) is an increasing sub-sequence.

Claim: If \( A \) is monotone the algorithm reports it with sufficiently high probability and if \( A \) is \( \epsilon \)-far from monotone the algorithm rejects with sufficiently high probability.

Proof: If \( A \) is monotone then all the binary searches will succeed and the algorithm always reports that \( A \) is monotone. Suppose \( A \) is \( \epsilon \)-far away from monotone. This implies \( |S| < (1 - \epsilon)n \) since \( S \) is a monotone sub-sequence and if \( |S| \geq (1 - \epsilon)n \), then changing at most \( n \epsilon \) coordinates \( j \notin S \) would make the input monotone. That would make \( A \) \( \epsilon \)-close to monotone. Hence the probability with which the algorithm reports \( A \) as monotone is,

\[
Prob[\text{ALG accepts}] < \frac{1 - \epsilon}{c}
\]

\[
= \left(1 - \frac{1}{\delta}\right)^{\delta}, \delta = \frac{1}{\epsilon}
\]

\[
\Rightarrow e^{-1}
\]

Thus if \( A \) is \( \epsilon \)-far from monotone, the algorithm rejects with probability \( 1 - e^{-1} \).

1.2.2 Testing for element distinctness

Goal: Given unsorted array \( A \) of length \( n \), test if all \( A(i) \) are distinct.

Algorithm
Randomly choose set $X$ with $\sqrt{n}/\epsilon$ elements
   **if** $X$ has a repeated element **report** failure
   **else** **report** success

The complexity of algorithm is $O((\sqrt{n}/\epsilon) \log n)$. If we use hashing the we can get rid of the log $n$ factor. Proof of correctness is based on “birthday paradox”.

### 1.2.3 Graph property testing

There are several models for testing properties of graphs. Let $G = (V, E)$, $n = |V|$, and $m = |E|$,

1. Dense model: These graphs are represented by its $n \times n$ adjacency matrix. We say that a graph is $\epsilon$–far from having a property in this model if more than an $\epsilon$–fraction $(\epsilon n^2)$ of its adjacency matrix need to be modified in order to obtain the property.

2. Sparse/bounded degree model: In this model there is an upper bound $d$ (some constant) on the degree of vertices. The graph is represented by an $n \times d$ matrix. We say that a graph is $\epsilon$–far from having a property in this model if more than an $\epsilon$–fraction $(\epsilon dn)$ of its adjacency matrix should be modified in order to obtain the property.

**Testing $K$-colorability**

Given a dense graph $G = (V, E)$ test,

- $G$ is $k$–colorable.
- $G$ is $\epsilon$–far from $k$–colorable, i.e. need to remove at least $\epsilon n^2$ edges to make it $k$–colorable.

For $k = 2$, the problem reduces to testing the bipartiteness of graph. Given a dense graph $G = (V, E)$, determine with high probability if it is bipartite or $\epsilon$–far from it.

**Algorithm**

- Randomly selects $\Theta\left(\frac{\log(1/\epsilon)}{\epsilon^2}\right)$ vertices
- **Accept** if the sub-graph induced on them is bipartite

In dense model $\exists$ constant time algorithm (with the constant $C_{k, \epsilon}$ depending on $k$ and $\epsilon$) such that the algorithm tests for $k$–colorability (i.e. whether the graph is bipartite or $\epsilon$ far from being bipartite).

In sparse model, for constant degree $d$ and $\epsilon$, testing bipartiteness requires $\Omega(\sqrt{n})$ queries of the “incidence vector”.

**Algorithm**

- **for** $\Theta\left(\frac{1}{\epsilon^2}\right)$ times
  - Select a vertex $v \in V$
    - **if** $\exists$ odd length cycle of $v$, **report** graph is not bipartite
  **end for**

4
2 Sub-linear space (streaming) algorithms

In streaming model input is a sequence of data \( A(1), A(2), \ldots, A(m) \ldots \) which is too large to be stored in memory. The space available is less than linear space \(< < m \). Common types of problems analyzed by streaming algorithms are:

1. Computing frequency (moments) statistics [4]: Let \( A = (a_1, a_2, \ldots, a_n) \) be a sequence of elements, where each \( a_i \) is a member of \( N = \{1, 2, 3, \ldots, n\} \). Let \( m_i \) denote the number of occurrences of \( a_i \) in the sequence \( A \), then,

\[
F_k = \sum_{i=1}^{n} m_i^k
\]  

(22)

\( F_k \) are called the frequency moments of \( A \) and provide useful statistics on the sequence. \( F_0 \) is the number of distinct elements appearing in the sequence, \( F_1 \) is the length of the sequence, and \( F_2 \) is the repeat rate or Gini’s index of homogeneity needed in order to compute the surprise index of the sequence. Surprise index for event \((i)\),

\[
S_i = \frac{\sum_{j} p_j^2}{p_j}
\]  

(23)

where, \( p_j = \frac{m_j}{n} \). Alon, Matias, and Szegedy [4] showed that for every \( k > 0 \), \( F_k \) can be approximated randomly using at most \( O(n^{1-1/k}\log n) \) memory bits.

2. Finding \( k \) “heavy hitters": Heavy hitters are the items occurring with frequency above a given threshold. E.g. those \( a_i : a_i \) occurs at least \( m/k \) times in the stream.

3. Finding rare or unique values.

References


