What can we do in sublinear time?

Sublinear Algorithms
Lecture 11

Ronitt Rubinfeld
Too much input? Too little time?
Models:

- **Sublinear time**
  - Random access queries
  - Samples

- **Sublinear space:**
  - Data stream
  - Sketching
Sublinear time models:

• Random Access Queries
  – Can access any word of input in one step

• Samples
  – Can get sample of a distribution in one step,
  – Alternatively, can only get random word of input in one step
    • When computing functions depending on frequencies of data elements
    • When data in random order

\[ x_1, x_2, x_3, \ldots \]
What can we hope to do without viewing most of the data?

• Change our goals?
  – for most interesting problems: algorithm must give approximate answer
What types of approximation?

• “Classical” approximation for optimization problems: output is number that is close to value of the optimal solution for given input. (not enough time to construct a solution)

• Property testing for decision problems: output is correct answer for given input, or at least for some other input “close” to it.

• Both types of approximations are also useful for distributions
I. Classical Approximation Problems
First:

- A very simple example –
  - Deterministic
  - Approximate answer
  - And (of course)…. Sublinear time!
Approximate the diameter of a point set [Indyk]

• Given: \( m \) points, described by a distance matrix \( D \), s.t.
  – \( D_{ij} \) is the distance from \( i \) to \( j \).
  – \( D \) satisfies triangle inequality and symmetry.
    (note: input size \( n = m^2 \))
• Let \( i, j \) be indices that maximize \( D_{ij} \) then \( D_{ij} \) is the diameter.
• Output: \( k,l \) such that \( D_{kl} \geq D_{ij}/2 \)

weak approximation!
Algorithm

• Algorithm:
  – Pick $k$ arbitrarily
  – Pick $l$ to maximize $D_{kl}
  – Output $D_{kl}$

• Why does it work?
  \[ D_{ij} \leq D_{ik} + D_{kj} \text{ (triangle inequality)} \]
  \[ \leq D_{kl} + D_{kl} \text{ (choice of } l \text{ + symmetry of } D) \]
  \[ \leq 2D_{kl} \]

• Running time? $O(m) = O(n^{1/2})$
More classical approximations?

• Later on (today, next week…)
  – Number of connected components
  – Minimum Spanning Tree
  – Other combinatorial optimization problems
    • Vertex cover
    • Maximum matching
II. Property testing
Main Goal:

- Quickly distinguish inputs that have specific property from those that are far from having the property.

All inputs

- Inputs with the property
- Close to having property
Property Testing

• Properties of any object, e.g.,
  – Functions
  – Graphs
  – Strings
  – Matrices
  – Codewords

• Model must specify
  – representation of object and allowable queries
  – notion of close/far, e.g.,
    • number of bits/words that need to be changed
    • edit distance
A simple property tester
Monotonicity of a sequence

- Given: list \( y_1, y_2, \ldots, y_n \)
- Question: is the list sorted?

- Clearly requires \( n \) steps – must look at each \( y_i \)
Monotonicity of a sequence

- Given: list $y_1, y_2, \ldots, y_n$

- Question: can we quickly test if the list close to sorted?
What do we mean by ``quick''?

- **query complexity** measured in terms of list size $n$

- Our goal (if possible):
  - *Very small* compared to $n$, will go for $clog n$
What do we mean by “close”? 

Definition: a list of size $n$ is $\varepsilon$-close to sorted if can delete at most $\varepsilon n$ values to make it sorted. Otherwise, $\varepsilon$-far.

Sorted: 1 2 4 5 7 11 14 19 20 21 23 38 39 45
Close: 1 4 2 5 7 11 14 19 20 39 23 21 38 45
Far: 45 39 23 1 38 4 5 21 20 19 2 7 11 14
Requirements for algorithm:

- Pass sorted lists
- Fail lists that are $\varepsilon$-far.
  - Equivalently: if list likely to pass test, can change at most $\varepsilon$ fraction of list to make it sorted

Probability of success $> \frac{3}{4}$
(can boost it arbitrarily high by repeating several times and outputting majority answer)

- Can test in $O(1/\varepsilon \log n)$ time
  - [Ergun, Kannan, Kumar, Rubinfeld, Viswanathan]
  - best possible [EKKRV] + [Fischer]
An attempt:

- Proposed algorithm:
  - Pick random $i$ and test that $y_i \leq y_{i+1}$

- Bad input type:
  - $1,2,3,4,5,...,n/4, 1,2,...,n/4, 1,2,...,n/4$
  - Difficult for this algorithm to find “breakpoint”
  - But other tests work well…
A second attempt:

- Proposed algorithm:
  - Pick random $i < j$ and test that $y_i \leq y_j$
- Bad input type:
  - $n/4$ groups of 4 decreasing elements
    - $4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9, ...$, $4k, 4k-1, 4k-2, 4k-3, ...$
  - Largest monotone sequence is $n/4$
  - must pick $i, j$ in same group to see problem
  - need $\Omega(n^{1/2})$ samples
A minor simplification:

• Assume list is distinct (i.e. $x_i \neq x_j$)

• Claim: this is not really easier
  – Why?
    Can “virtually” append $i$ to each $x_i$
    
    \[
    x_1, x_2, \ldots, x_n \rightarrow (x_1, 1), (x_2, 2), \ldots, (x_n, n)
    \]
    e.g., $1, 1, 2, 6, 6 \rightarrow (1, 1), (1, 2), (2, 3), (6, 4), (6, 5)$
    Breaks ties without changing order

(historical note: this is an old trick used in parallel algorithms to break ties)
A test that works

• The test:

Test $O(1/\varepsilon)$ times:
  • Pick random $i$
  • Look at value of $y_i$
  • Do binary search for $y_i$
  • Does the binary search find any inconsistencies? If yes, FAIL
  • Do we end up at location $i$? If not FAIL
  – Pass if never failed

• Running time: $O(\varepsilon^{-1} \log n)$ time
• Why does this work?
Behavior of the test:

• Define index $i$ to be good if binary search for $y_i$ successful

• $O(1/\varepsilon \log n)$ time test (restated):
  – pick $O(1/\varepsilon)$ $i$’s and pass if they are all good

• Correctness:
  – If list is sorted, then all $i$’s are good (uses distinctness)
    • So test always passes
  – If list likely to pass test,
    • Then at least $(1-\varepsilon)n$ $i$’s are good.
    • Main observation: good elements form increasing sequence
      – Proof: for $i<j$ both good need to show $x_i < x_j$
        • let $k =$ least common ancestor of $i,j$
        • Search for $i$ went left of $k$ and search for $j$ went right of $k$ →
          $x_i < x_k < x_j$
      • Thus list is $\varepsilon$-close to monotone (delete $< \varepsilon n$ bad elements)
III. Classical Approximation Problems (more examples)
Minimum spanning tree (MST)

• What is the cheapest way to connect all the dots?

• Best known:
  – Deterministic $O(m \alpha(m))$ time [Chazelle]
  – Randomized $O(m)$ time [Karger Klein Tarjan]
A sublinear time algorithm:
[Chazelle R. Trevisan]

Given input graph with
- weights in $[1..w]$
- average degree $d$
- adjacency list representation

outputs $(1+\varepsilon)$-approximation to MST in time
$O(dw\varepsilon^{-3} \log dw/\varepsilon)$

Remarks: (1) sublinear when $dw=o(m)$
         constant when $d,w$ bounded
(2) we also know that $\Omega(dw\varepsilon^{-2})$ required
(3) case of integral weights, max degree $d$ can be done in $O(dw \varepsilon^{-2} \log w/\varepsilon)$ time
Idea behind algorithm:

• characterize MST weight in terms of number of connected components in certain subgraphs of $G$

• show that number of connected components can be estimated quickly
MST and connected components

Suppose all weights are 1 or 2. Then MST weight

\[ \text{weight} = \# \text{weight 1 edges} + 2 \cdot \# \text{weight 2 edges} \]

\[ = n - 1 + \# \text{of weight 2 edges} \]

\[ = n - 2 + \# \text{of conn. comp. induced by weight 1 edges} \]
Suppose all weights are 1 or 2. Then MST weight
\[
= \# \text{weight 1 edges} + 2 \cdot \# \text{weight 2 edges}
\]
\[
= n - 1 + \# \text{of weight 2 edges}
\]
\[
= n - 2 + \# \text{of conn. comp. induced by weight 1 edges}
\]
• For integer weights 1..w let
  \[ c(i) = \# \text{ of connected components induced by edges of weight at most } i \]
• Then MST weight is
  \[ n - w + \sum_{i=1,\ldots,w-1} c(i) \]
• additive approximation of \( c(i) \)'s to within \( \epsilon n/w \) gives additive approx of MST to within \( \epsilon n \)
  – Since MST > n-1, also gives multiplicative approximation of MST to within \( 1\pm\epsilon \)
Approximating number of connected components:

• Given input graph with
  – max degree $d$
  – adjacency list representation

• outputs additive approximation to within $\varepsilon_0n$ of the number of connected components in time $O(d \varepsilon_0^{-2} \log 1/\varepsilon_0)$

• Can show $\Omega(d \varepsilon_0^{-2})$ time is required
Approximating # of connected components

• Let \( c \) = number of components
• For every vertex \( u \), define
  \[ n_u := \frac{1}{\text{size of component of } u} \]
  for any connected component \( A \subseteq V \),
  \[ \sum_{u \in A} n_u = 1 \]
  \[ \sum_u n_u = c \]
Main idea

• Estimate sum of approximations of $n_u$’s via sampling

• To estimate $n_u \equiv 1 / \text{size of component of } u$ quickly:
  – If size of component is big, then $n_u$ is small so easy to estimate (similar to property tester for connectivity [Goldreich Ron])
  – Suffices to compute $n_u$ exactly only for small components
Some more details:

Estimating $n_u \equiv 1 / \text{size of u's component}$:

- let $\tilde{n}_u := \max \{n_u, \varepsilon_0/2\}$
  - When size of u’s component is $< 2/\varepsilon_0$, $\tilde{n}_u = n_u$
  - Else $\tilde{n}_u = \varepsilon_0/2$

- $|n_u - \tilde{n}_u| < \varepsilon_0/2$ so $c = \sum_u n_u = \sum_u \tilde{n}_u \pm \varepsilon_0 n / 2$

- can compute $\tilde{n}_u$ quickly
  - in time $O(d/\varepsilon_0)$ with BFS
Not quite optimal algorithm:

CC-APPROX($\varepsilon_0$):

Repeat $O(1/\varepsilon_0^3)$ times
  pick a random vertex $v$
  compute $\bar{n}_v$ via BFS from $v$, stopping after at most $2/\varepsilon_0$ new nodes
return (average of the values $\bar{n}_v$) $\cdot$ $n$

Run time: $O(d /\varepsilon_0^4)$
Correctness

• Chernoff bounds $\rightarrow$ algorithm gives good estimate of average/sum of the $\bar{n}_v$ values
• Previous arguments $\rightarrow$ good estimate of average of the $\bar{n}_v$ values gives (slightly less) good estimate of average/sum of the $n_v$ values
• Estimate of sum of the $n_v$ values gives estimate of number of connected components via $\sum_u n_u = c$
Improvement for MST:

This gives MST algorithm with runtime $O(dw^2 \varepsilon^{-4})$

Can do better:

$\quad O(dw\varepsilon^{-3} \log dw/\varepsilon)$ algorithm
Further work:

• Euclidean MST approximation algorithm
  [Czumaj Ergun Fortnow Newman Magen Rubinfeld Sohler]
  – Given access to certain data structures can do better

• Metric MST [Czumaj Sohler]
  – \((1+\varepsilon)\)-approximation in time \(\tilde{O}(n)\)