More Sub-linear Algorithms
+
Compression Algorithms

Lecture 21
Recall: Property Tester for decision problems

Let P be a Decision problem with
- $distance(x, y)$ function between any two inputs
- parameter $\varepsilon > 0$

**Property Tester for P with parameter $\varepsilon$:**

for every input $x$
- if $P(x) = \text{yes}$, then $\text{Prob}[\text{Tester accepts } x] = 1$
- if $\forall x' \text{ such that } P(x') = \text{yes}$, $\text{distance}(x, x') > \varepsilon$, then $\text{Prob}[\text{Tester reject } x] > 2/3$
Testing Properties of graphs

- Given graph $G = (V, E)$, $|V| = n$, $|E| = m$
- Decision problems regarding $G$
  - Connected?
  - bipartite?
  - $k$-colorable?
  - $K$-clique?

Goal: Design Property Testers with sub-linear query complexity
Example: Testing Connectivity in Sparse Graphs

• Let G be a graph with n vertices, m edges and average constant degree \( d = \frac{m}{n} \)

• Graph Representation
  - Adjacency Lists: each of the n vertices points to a list of adjacent vertices of average size \( d \)

• Query: what is the i-th neighbor of \( u \)?
We say that graph $G$ is $\varepsilon$-far from connected if we need to add to it more than $\varepsilon m$ edges to $G$ to make it connected, and $\varepsilon$-close otherwise.

**Property Tester for Connectivity with parameter $\varepsilon$:**

- Accept $G$, if $G$ is connected.
- Reject $G$ with probability $> 2/3$, if $G$ is $\varepsilon$-far from connected.
- Don’t care otherwise.
The idea for Connectivity Tester

Intuition:
If Graph is far from being connected ⇒
Graph has many connected components ⇒
many small connected components ⇒
many vertices in small connected components

Idea for Tester: pick vertices at random and reject if they are in a small component
Choose a vertex $s$ at random
   Ran BFS ($s$), stop after you visit $> \text{small (}=2/\varepsilon d)$ vertices
   if found $\leq \text{small}$ number of vertices during BFS, reject
   (the graph is not connected)
Repeat for $4/\varepsilon d$ repetitions,
accept if never found small component

**Theorem:** If $G$ is connected, Tester always accept
   If $G$ is $\varepsilon$-far from connected then,
   Prob $[\text{Tester Rejects}] > 2/3$
   Query Complexity: $O(1/(\varepsilon d)^3) = O(\text{poly}(m/n)) =$
   $O(\text{poly}(\varepsilon, d)) = \text{SUBLINEAR}$
Far from Connected ⇒ Many Small Connected Components

Let $CC(G) = \# \text{ connected components in G}$

**Simple Observation:**
If graph is not connected, then the minimum number of edges to add to become connected is $CC(G) - 1$

**Lemma:** Graph is $\varepsilon$-far from connected ⇒ more than half of the connected components are small (<$2/\varepsilon d$ nodes in a component)

**Proof:** If G is $\varepsilon$-far, then there are more than $\varepsilon m$ components. Let us partition these into small and big components where big component has > $2/\varepsilon d$ vertices & small has < $2/\varepsilon d$ vertices. Clearly, #big components is at most $n/\text{size-of-big} < \varepsilon m/2$

#small components + #big components = $CC(G)$ ⇒
#small components = $GG(G) - #\text{big components} > CC(G) - \varepsilon m/2 > \varepsilon m/2$
Property Tester for Connectivity

Theorem:
If G is connected, Tester always accept
If G is $\varepsilon$-far from connected then, Tester Rejects with probability $> 2/3$

Proof:

⇒ Obviously, if G is connected will accept.

⇐ the number of nodes in small components $>$
    the number of small components $>$ (by lemma)
    half of the components $> \varepsilon m/2$

Prob( algorithm accept if G is $\varepsilon$-far) $<$

$(1 - \text{prob(pick vertex in small component)})^{4/\varepsilon d} <$

$(1 - \varepsilon m/2n)^{4/\varepsilon d} = (1 - \varepsilon d/2)^{4/\varepsilon d} < 1/e^2 < 1/3$
Problem: Given $m \times m$ matrix of distances between $m$ points in the plane (more generally satisfying symmetry and triangle inequality)
compute the diameter $d = \max_{u,v} \text{distance} (u,v)$
Approximation Algorithm

- Pick an arbitrary point \( u \)
- Find the point \( v \) furthest from \( u \) by looking at the column headed by \( u \)
- Output \( z = \text{distance} (u, v) \)

**Claim:** Algorithm is a 2-approximation for the diameter problem with time complexity is \( O(m)=\sqrt{\text{input size}} \)

**Proof:** Need to show \( z > \text{diameter}/2 \). Fix any 2 pts \( x \) & \( y \) s.t. \( \text{Diameter} = d(x,y) := \text{distance} (x,y) \). Then,

\[
\text{diameter} = d(x,y) \leq d(x,u) + d(u,y) \leq d(v,u) + d(u,v) \leq 2d(u,v) = 2z
\]
Is Randomness always necessary for sub-linear algorithms?

• No, depends on the problem and the guarantee of the algorithm

• Example: Deterministic approximation algorithm for the diameter distance.
Basic Data Streaming Model

- Single pass over the data: $i_1, i_2, \ldots, i_n$
- Typically, we assume $n$ is known

- Bounded storage (typically $n^\alpha$ or $\log^c n$)
  - Units of storage: bits, words or „elements”
  - (e.g., points, nodes/edges)

- Fast processing time per element

- Randomness: almost always necessary
Compression Algorithms
Representing Data

Course so far: our goal was to represent data so that it is easiest to manipulate
- Search, edit: add, delete, cut, paste.

• However, there are actually competing interests
  - easily manipulated/processed
  - short for storage and transmission

• Today our goal: represent data in the shortest possible way

• Look for algorithms to achieve this
Today: Lossless Compression

\[ \text{D} = \text{D'} \]

Lossy Compression: \( \text{D'} \) close-enough to \( \text{D} \) (application dependent)
Data Compression: Lossy or Lossless

• Lossless:
  – Huffman Coding
  -- Lempel-Ziv (repeating patterns)
  – .gif

• Lossy:
  – .mp3
  – .jpg
Non Adaptive and Adaptive Algorithms

- **Non-Adaptive:** assume we know something about the data source, say the frequency of different characters
- **Adaptive:** do not know anything, and the algorithms has to build the knowledge up by itself.

- TODAY + Section: An example of non-adaptive algorithms
Set Up

- **Input:** a sequence of characters from a fixed and known alphabet
  - e.g. Alphabet = English characters a, b, c...
  - e.g. Input: “I am a human being”

- Look for **Binary codes** which encode each character in the alphabet as a binary string, also called a **code word**.
- **Code = {code words}**

- **Output:** a concatenation of code words each corresponding to the characters in the input.
Example of fixed length code

- **Fixed length code**: each character is assigned a code word of the same length
  - Suppose we have a file of 100k characters and each character is one of 8 letters [a...h]
  - Need 3 bits to represent each character: a:000, b:001, c:010,..., h:111
  - To represent the File requires: 300k bits

- Easy to encode and decode:
  - e.g. baba ≡ 001000001000
Variable Length Code

**Variable Length:** Code words for different characters may have different length

Say that you knew something about the document to be compressed: the *frequency* with which each character appears.

**Big-Idea:** use shorter code words to represent frequent characters with higher frequency & longer code words to represent rarer characters
Example of variable length code

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

- Frequency
- VarLenCode: 0 101 100 111 1101 1100
- Variable Length Code Size for example:
  \[45k*1 + 13k*3 + 12k*3 + 9k*4 + 5k*4 = 224k\], a 25% savings on the 300k of the fixed length code

- But how do we decode?
Use Prefix (Free) Coding

- **Prefix coding**: Codes where no codeword is a prefix of another codeword.
- **Ex**: code = \{0, 101, 100, 111, 1101, 1100\}

- **Easy to Decode back to the original file**: can scan from left to right, as soon as recognize a code word in the file, peel it off and continue decoding.
- **01101100 = 0101011110011011101**
Given alphabet \{a_i\} and frequencies f(a_i)

**Codes:** find binary code words c(a_1) \ldots c(a_n) that minimize number of bits to represent your data

\[ B(C) = \sum_i f(a_i) |c(a_i)| \]

**Trees:** Let d(a_i) be the depth of leaf a_i. Find binary tree T with n leaves labeled with a_1\ldots a_n that minimizes

\[ B(T) = \sum_i f(a_i) d(a_i) \]

Equivalent problems since \[ d(a_i) = |c(a_i)| \]
Search for Optimal Coding

• Shannon: invented information theory
• Fano and Shannon worked together to find minimal size codes, found good heuristics
• Fano assigned the problem to his class
• Huffman solved it: not knowing that his teacher had struggled with it !!!!

• A lesson to us all.
Huffman Code: a bottom-up algorithm to construct an optimal prefix code/tree

- Let A be the alphabet and \( f() \) the function of frequencies
- For all \( x \) in A, insert \((x, f(x))\) in queue Q
- Call Huffman \((A, f, Q)\)

Huffman Pseudo Code \((A, f, Q)\)

1. Left-child = \( x \) s.t. \( f(x) = \min_a \{ f(a) \text{ in } Q \} \), delete \((x, f(x))\) from Q
2. Right-child = \( y \) s.t. \( f(y) = \min_a \{ f(a) \text{ in } Q \} \), delete \((y, f(y))\) from Q
3. Allocate a tree node \( z \)
   - Make left-child and right-child the children of node \( z \)
   - Let \( A = A - \{x, y\} + \{z\} \) and \( f(z) = f(\text{left-child}) + f(\text{right-child}) \)
   - Insert \((z, f(z))\) in queue Q
4. if \( A \neq \emptyset \), call Huffman \((A, f, Q)\) else output root of the tree
Runtime: Let n be the size of the alphabet. Then, to create the tree \( O(n \log n) \), since each priority queue operation takes \( O(\log n) \) and we have n operations.
1. Take the characters and their frequencies, and sort this list by increasing frequency

E: 10, T: 7, O: 5, A: 3 →
A: 3, O: 5, T: 7, E: 10
Huffman Codes

2. Make the characters into vertices of a tree:

A: 3
O: 5
T: 7
E: 10
Huffman codes

3. Take the smallest (first) 2 vertices from the list and make them children of a new vertex having the sum of their frequencies

A: 3
O: 5
Z: 8

New vertex with Frequency 3 + 5 = 8
Huffman Codes

4. Insert the new vertex into the sorted list of vertices waiting to be put into the tree

List of remaining vertices:

- T: 7
- E: 10

New list, with the new vertex Inserted:

- T: 7
- Z: 8
- E: 10

New
5. Take the first 2 vertices from the list and make them children of a new vertex having the sum of their frequencies

New vertex with Frequency $7 + 8 = 15$
6. Insert the new vertex into the sorted list of vertices waiting to be put into the tree

List of remaining vertices:

New list, with the new vertex inserted:
7. Take the first 2 vertices from the list and make them children of a vertex having the sum of their frequencies.
Huffman Codes

- Left branch is 0
- Right branch is 1

Huffman code:
- E: 0
- T: 10
- A: 110
- O: 111
Optimality?
Claim: Huffman’s Algorithm produces a tree $T$ with $\min B(T)$

- **Proof** by induction on $|A|$.
  - **Base:** When $|A| = 2$, the optimal tree clearly has two leaves, with codes 0 and 1, which the algorithm constructs.
  - **Induction Step:** Suppose $|A| > 2$. The first greedy choice the algorithm makes is to make the two lowest-frequency characters ($x$ and $y$) into leaf nodes, create a new node $z$ with frequency $f(z) = f(x) + f(y)$ and applies the algorithm recursively to the smaller $A' = A \setminus \{x, y\} \cup \{z\}$. Since $A'$ is smaller, by induction the algorithm on $A'$ builds optimal $T'$.

We will show that adding $x$ and $y$ as children to $z$ in $T'$ yields an optimal tree $T$ for $A$. 
Proof (2/4)

Claim: Adding x and y as children to z in an optimal $T'$ for $A'$ yields optimal $T$ for $A = A' - \{z\} + \{x, y\}$

Proof: Suppose not. Say $\exists$ tree $BT$ s.t. $B(BT) < B(T)$. We will show a contradiction to $T'$ being optimal for $A'$ which holds by the induction hypothesis.

Case 1: $x$ and $y$ are siblings in $BT$. Remove them and make their parent a leaf labeled $z$ to get $BT'$ for $A'$. Then get $B(BT') < B(T')$ by the following calculation.

$B(BT') = B(BT) + f(z)d(z) - f(x)d(x) - f(y)d(y) = B(BT) + (f(x) + f(y))(d(x) - 1) - f(x)d(x) - f(y)d(x) = B(BT) - (f(x) + f(y))$. Similarly, $B(T') = B(T) - (f(x) + f(y))$

So, $B(BT) < B(T) \Rightarrow B(BT') < B(T')$ contradiction to $T$'s optimality.
Case 1: x and y are siblings in BT. Remove them and make their parent a leaf labeled z to get BT’ for A’ = A-\{x,y\} + \{z\}. Then get B(BT’) < B(T’) by the following calculation.

\[ B(BT) = B(BT’) - f(z)d(z) + f(x)d(x)-f(y)d(y) = \]
\[ B(BT’) - (f(x)+f(y)) (d(x)-1)+f(x)d(x)-f(y)d(x) = \]
\[ B(BT’) + (f(x)+f(y)) . \]

Similarly, B(T) = B(T’)+(f(x)+f(y))

So, B(BT) < B(T) \Rightarrow B(BT’) < B(T’) contradicting T’ being optimal.
• **Case 2:** $x$ and $y$ are not sibling leaves in the better tree $BT$. Then can show that there exists another tree $BT''$ where $B(BT) \geq B(BT'')$ in which $x$ and $y$ are sibling leaves, and apply the same argument of case 1 for $BT''$.

  - **Lemma:** Optimal Huffman Tree must be a Full Binary Tree (i.e. each internal node has 2 children)
  - **Proof:** if not, get rid of it and replace by child, that would decrease the total cost of encoding

• **Going from $BT$ to $BT''$:** For sibling leaves $a$ & $b$ at maximum depth of $BT$, replace $a$ with $x$ & $b$ with $y$. The weight of the new tree can not increase. (CLRS16.3)
Huffman Codes Summary

- Reduce size of data by 20%-90% in general

- If no characters occur more frequently than others, then no advantage over fixed length code

- **Encoding:** Given the characters & their frequencies, perform the algorithm and generate a code. Write the file using the code

- **Decoding:** Given the Huffman tree, figure out what each character is by traversing edges to character in leaf