1 Overview

Since many problems cannot be solved without inspecting all of the problem instance, sub-linear algorithms generally use sampling to investigate some parts of the problem instance in order to answer relaxed versions of these problems. This lecture covers more examples of sub-linear algorithms from the following categories:

Classical approximation problems

We aim to construct an approximate solution to classical problems. In many cases, we may not have enough time to construct the whole solution—instead, we approximate the size of the solution. This lecture covers the following problems.

- Approximate the diameter of a point set under metric space
- Approximate the size of minimum spanning tree

Property testing

A decision problem can generally be described as property testing: we accept instances that belong to a specified set $P$ (i.e., have property $P$), or reject otherwise. In our relaxed version, we accept all instances with property $P$, and reject those far from having the property with constant probability. For instances close to having property $P$, our algorithm may give an arbitrary answer. For these problems, we need to specify the representation of problem instances, the type of allowable queries, and the notion of being close to (far from) having property $P$. Common notions include the minimum number of bits/words in the problem instance that must be modified for property $P$ to hold, as well as edit distances. This lecture covers the following problems.

- Testing the sortedness of a sequence
- Testing the connectivity of a bounded-degree graph

2 Approximate the Diameter of a Point Set under Metric Space

In this problem, we are given a set of $m$ points via its $m \times m$ distance matrix $D$, where $D_{ij}$ denotes the distance between point $i$ and point $j$. We require that the distances satisfy both symmetry and triangle inequality. Let $x$ and $y$ be the points such that $D_{xy}$ is maximized—we call this distance the diameter of the point set. Our goal is to find points $k$ and $l$ such that $D_{kl} \geq D_{ij}/2$; this gives a 2-approximation of the diameter.

Our algorithm can be described as follows. Pick an arbitrary point $k$, and pick another point $l$ to maximize the distance $D_{kl}$. This guarantees that $D_{xk} = D_{kx} \leq D_{kl}$ and $D_{ky} \leq D_{kl}$ (via symmetry). Then, by triangle inequality,

$$D_{xy} \leq D_{xk} + D_{ky} \leq D_{kl} + D_{kl} = 2D_{kl}.$$ 

While this algorithm takes $O(m)$ to find $l$, the input is the matrix $D$ whose size is $n = m^2$. Thus the overall running time is $O(\sqrt{n})$, which is sub-linear. Unlike most sub-linear algorithms, this algorithm is deterministic and gives a 2-approximation guarantee.
3 Testing the Sortedness of a Sequence

3.1 Problem Statement

Recall that a sequence of \( n \) elements \( y_1, \ldots, y_n \) is sorted if \( y_1 \leq y_2 \leq \cdots \leq y_n \). We call a sequence \( \epsilon \)-close to sorted if we can delete \( \epsilon n \) elements to make the remaining sequence sorted; otherwise we call such sequence \( \epsilon \)-far from sorted. Since determining whether a sequence is sorted takes linear time, we consider a weaker, randomized problem. Specifically, we want to accept all sorted sequences, and reject sequences that are \( \epsilon \)-far from sorted with probability at least \( \frac{3}{4} \). Note that we may accept or reject sequences that are \( \epsilon \)-close to sorted. We aim to achieve the running time of \( O(\frac{1}{\epsilon} \log n) \), which matches the lower bound of this problem.

3.2 Failed Attempts

Consider first an algorithm that repeatedly picks a random index \( i \), checks whether \( y_i \leq y_{i+1} \), and rejects if this condition does not hold. This algorithm only detects drops between consecutive terms. Consider the following input:

\[
1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4
\]

There are only 3 breakpoints, so in expectation, we will need to repeat the procedure above \( \Omega(n) \) times to detect one. This failure suggests that we also need to compare distant elements in the sequence.

Now consider the algorithm that picks a random pair of indices \( i < j \) and checks whether \( y_i \leq y_j \). While we compare distant elements, this algorithm still fails on the following input:

\[
4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9, \ldots, 4k, 4k - 1, 4k - 2, 4k - 3, \ldots
\]

That is, we have \( n/4 \) groups of four consecutive elements that are decreasing within the group but increasing between different groups. Thus we need to pick \( i \) and \( j \) from the same group to find a decreasing pair. From the birthday problem, this takes \( \Omega(\sqrt{n}) \) samples.

From these two attempts, clearly we cannot rely on simply comparing elements at a fixed distance or at random distances apart. We now present the algorithm that uses powers of two as distances.

3.3 Algorithm

First, we ensure that all elements in the sequence are distinct by appending the index \( i \) to each element \( y_i \). For example, the sequence 1, 1, 2, 6, 6 becomes (1, 1), (1, 2), (2, 3), (6, 4), (6, 5). Note that we perform this appending process virtually—instead of modifying the whole sequence, we only append sampled elements for comparison purpose. This approach break ties without changing the order, so the property of being \( \epsilon \)-close/far are preserved.

Now that all elements are distinct, we apply the algorithm given below. Note that the value \( t \) is to be defined later. Since each binary search takes \( O(\log n) \) time and queries, the both the running time and query complexity of this algorithm is \( O(\frac{1}{\epsilon} \log n) \).

\[
\text{for } i = 1 \text{ to } t \in O(1/\epsilon) \text{ do}
\]
\[
\begin{align*}
&\text{pick a random } i \in [n] \\
&\text{perform a binary search for } y_i \\
&\text{if the binary search finds any inconsistencies or does not end at location } i \text{ then}
&\quad \text{REJECT}
&\text{end if}
\end{align*}
\]
\[
\text{end for}
\]
\[
\text{ACCEPT}
\]
3.4 Correctness

Let us call an index $i$ good if the binary search on $y_i$ is successful. That is, (1) when we recurse to the left (right), the new middle element is less than (more than) the previous middle element; and (2) the search ends up finding $y_i$ at position $i$. By distinctness, all indices are good when the sequence is sorted, and thus our algorithm always accepts sorted sequences. We now need to show that when the input is not $\epsilon$-close, we reject it with probability at least $3/4$. Equivalently, we will show the contrapositive: if our algorithm accepts a sequence, then with probability higher than $1/4$, it is $\epsilon$-close.

First, consider two good indices $i < j$. Let $k$ be the last index that the binary searches for $y_i$ and $y_j$ share. If $k = i$ or $k = j$, then $y_i < y_j$ holds. Otherwise, after querying for $y_k$, we decide to go left for $y_i$, but go right for $y_j$. Since these searches are good, $y_i < y_k < y_j$ as well. As this applies to all pairs of good indices, then the sequence of elements with good indices must be sorted.

Suppose that our algorithm accepts a sequence with probability higher than $1/4$. Then all $t$ sampled indices are good. If more than $\epsilon n$ indices are bad, then the probability of choosing no bad indices is at most $(1 - \epsilon)^t$. For sufficiently large $t = 1/\epsilon = O(1/\epsilon)$, this probability is at most $(1 - \epsilon)^{\epsilon/\epsilon} \leq e^{-\epsilon} \leq 1/4$, which creates a contradiction. Thus, at most $\epsilon n$ indices are bad. Since removing these $\epsilon n$ elements results in a sorted sequence, then this sequence is $\epsilon$-close to sorted, as needed.

4 Testing Connectedness of a Bounded-Degree Graph

4.1 Problem Statement

Let $G$ be a graph with $n$ vertices and maximum degree at most $d$. Given its adjacency list representation, we wish to check whether the graph is connected. For an exact algorithm, this clearly requires $O(n + m)$ time, which can be achieved via depth-first search. We now restate this problem as a property testing problem. As usual, we allow degree queries and neighbor queries. Let $P$ be the set of connected graphs with maximum degree at most $d$. We call a graph $G$ $\epsilon$-close to connected if $G$ also has maximum degree also bounded by $d$, and there exists some graph $G'$ in $P$ such that $G$ and $G'$ differs on at most $\epsilon dn$ edges. Our goal is to accept all connected graphs, and reject graphs that are $\epsilon$-far from connected with probability at least $3/4$.

For this lecture, we make the following simplification. While we restrict that $G$ has maximum degree at most $d$, we do not restrict the degree of graphs in $P$. Therefore, a graph $G$ is considered $\epsilon$-close to connected as long as adding $\epsilon dn$ edges makes $G$ connected.

4.2 Algorithm

Our algorithm is based on the following observation: if $G$ is far from connected, then many components are small, and many vertices belong to small components. We assume that $n \geq 2/\epsilon d$ (otherwise we can check for connectedness by a simple search in $O(1/\epsilon)$. Our algorithm is given below. Again, the value $t$ is to be defined later. Each step of the breadth-first search (BFS) takes $O(d)$ time, and we run up to $O(1/\epsilon d)$ steps for each loop. As we repeat the loop $O(1/\epsilon d)$ times, the overall running time is $O(1/\epsilon^2 d)$.

```plaintext
for i = 1 to $t \in O(1/\epsilon d)$ do
    pick a random vertex $s$
    run a breadth-first search starting at $s$ up to $2/\epsilon d$ steps
    if the breadth-search search visited less than $2/\epsilon d$ distinct vertices then
        REJECT
    end if
end for
ACCEPT
```
4.3 Correctness

Clearly, our algorithm always accepts if $G$ is connected, since it has a single component of size $n \geq 2/ed$. Now consider the case where $G$ is $\epsilon$-far from connected. By way of contradiction, if we have less than $\epsilon d n$ components, we can connect all of them by adding at most $\epsilon d n$ edges between these components, making $G$ $\epsilon$-close to sorted. (This may violate the bounded-degree requirement, which we ignore in this lecture.) Therefore, $G$ must have at least $\epsilon d n$ components.

Let us call a component small if its size is less than $2/ed$, and call it large otherwise. Since we have $n$ vertices in total, there can be at most $n/(2/ed) = edn/2$ large components. Now that $G$ has at least $\epsilon d n$ components, then there are at least $\epsilon d n - edn/2 = \epsilon d n/2$ small components. Each small component has at least 1 vertex, and thus at least $\epsilon d n/2$ vertices are in small components.

So, $ed/2$ fraction of the vertices are in small components. Our algorithm correctly rejects $G$ if it finds one of them. For sufficiently large $t = c/ed \in O(1/ed)$, this occurs with probability at least $1 - (1 - ed/2)^{c/ed} \geq 1 - e^{-c/2} \geq 3/4$, as desired.

5 Approximate the Size of Minimum Spanning Tree

5.1 Problem Statement

In this problem, given a connected graph $G$, we want to compute the $(1 + \epsilon)$-approximation to the size of its minimum spanning tree (MST). That is, if $M^*$ is the size of the MST, we want to compute $M$ such that $(1 - \epsilon)M^* \leq M \leq (1 + \epsilon)M^*$. For this lecture, we assume that the maximum degree is $d$, and the weight of each edge is in $[w]$ for some integer $w$. We allow the same type of queries on the adjacency list representation.

The idea given below leads to an algorithm with running time $O(dw/\epsilon^2)$, and query complexity $\Omega(dw/c^2)$. This is sub-linear when $dw = o(m)$. It can be generalized to handle non-integral weights with running time $O(dw/\epsilon^2 \log dw/\epsilon)$. For this lecture, however, we present a simpler algorithm that gives $O(dw^5/\epsilon^4)$ running time.

5.2 Main Algorithm

We characterize the MST weight in terms of number of connected components in certain subgraphs of $G$, then compute the number of connected components quickly. Let $G_i$ be the graph $G$ containing only edges weighted at most $i$, and let $c_i$ be the number of connected components in $G_i$. Since the graph is connected, $c_w = 1$. For convenience, define $c_0 = n$.

Consider Kruskal’s algorithm. By the time we consider all edges in $G_i$, we must have connected the graph into $c_i$ components. Therefore, when we consider edges of weight $i$, $c_{i-1} - c_i$ edges are added, contributing $(c_{i-1} - c_i)$ to the total weight. Thus, the weight of our MST is

$$1(c_0 - c_1) + 2(c_1 - c_2) + \cdots + w(c_{w-1} - c_w) = c_0 + c_1 + \cdots + c_{w-1} - w \cdot c_w = n - w + \sum_{i=1}^{w-1} c_i.$$ 

Our algorithm approximates each $c_i$ with additive error $\epsilon n/w$, so that the total error is at most $\epsilon n$ once we apply this formula. Since the MST has weight at least $n - 1$, this gives an $(1 + \epsilon)$-approximation. In the next section, we present an algorithm that gives an approximation of $c_i$ with additive error $\epsilon_0 n$ with constant probability in $O(d/\epsilon_0^2)$ time. By setting $\epsilon_0 = \epsilon/w$ and repeat this algorithm to compute $c_1, \ldots, c_{w-1}$, the overall running time becomes $O(dw^5/\epsilon^4)$.

5.3 Approximating the Number of Connected Components

Our goal is to approximate the number of connected components to within an additive error of $\epsilon_0 n$. Let $c$ be the number of components. For each vertex $u$, let $C_u$ denotes the component containing $u$, and
let $|C_u|$ be its size. Define $n_u = 1/|C_u|$. Since the sum of $n_u$ for all vertices in a single component is $|C_u| \cdot (1/|C_u|) = 1$, then the sum of $n_u$ over all vertices in $G$ is $c$. Given $u$, we can compute $|C_u|$ exactly via BFS, but this is slow when $|C_u|$ is large. Instead, we will approximate $|C_u|$ when it is large–its error becomes small when we compute $n_u = 1/|C_u|$.

Specifically, we use $\tilde{n}_u = \max\{n_u, \epsilon_0/2\}$ as our approximation of $n_u$. To compute $\tilde{n}_u$, we run the BFS up to $2/\epsilon_0$ steps. If the BFS visited less than $2/\epsilon_0$ distinct vertices, we compute $n_u$ exactly. Otherwise, we use $\epsilon_0/2$ as our approximation. In this case, $\epsilon_0/2 = \tilde{n}_u \geq n_u > 0$, so the error $|n_u - \tilde{n}_u| < \epsilon_0/2$, and the total error of $c$ is at most $\epsilon_0 n/2$.

Our algorithm is given below. Since each BFS takes $O(d/\epsilon_0^4)$ time, this algorithm takes $O(d/\epsilon_0^4)$ time. With sufficiently large $t \in O(1/\epsilon_0^3)$, by Chernoff bound, this algorithm gives an additive error of at most $\epsilon_0 n/2$ with constant probability.

```latex
for i = 1 to $t \in O(1/\epsilon_0^3)$ do
    pick a random vertex $u_i$
    run a breadth-first search starting at $u_i$ up to $2/\epsilon$ steps
    compute $\tilde{n}_{u_i}$ from the number of nodes visited by the breadth-first search
end for
return $\frac{n}{t} \sum_{i=1}^{t} \tilde{n}_{u_i}$
```