6.5240 Sublinear Time Algorithms October 15, 2026

Lecture 10

Lecturer: Ronitt Rubinfeld *Scribe: Miranda Wang*

Contents

1 Partition Oracles **1**
1.1 Recap of Lecture 9 (Planarity Testing) 1
1.2 Constructing Partition Oracles 2
1.2.1 Global Partitioning Algorithm 2
1.2.2 Local Simulation of Partitioning Algorithm 3

2 Monotonicity Testing **4**
2.1 Monotonicity Testing of Lists 4
2.2 Monotonicity Testing on The Boolean Hypercube 5

1 Partition Oracles

1.1 Recap of Lecture 9 (Planarity Testing)

Definition 1 *A graph G is (ε, k)–*hyperfinite *if one can remove $\leq \varepsilon n$ edges such that all remaining connected components are of size $\leq k$.*

Theorem 2 *For all ε and Δ, there exists a constant c such that every planar graph of maximum degree Δ is $(\frac{\varepsilon\Delta}{2}, \frac{c}{\varepsilon^2})$–hyperfinite.*

Let G be the given graph which is $(\frac{\varepsilon\Delta}{2}, \frac{c}{\varepsilon^2})$–hyperfinite, and $G'$the subgraph obtained from removing at most $\frac{\varepsilon\Delta}{2}n$ edges so that the size of each connected component is at most $\frac{c}{\varepsilon^2}$.

- If G is planar, then G' is planar.
- If G is ε–far from planar, then G' is $\frac{\varepsilon}{2}$–far from planar.

Assuming access to a partition oracle, we thus have an algorithm for testing whether a given graph is planar or not. Denote our given graph by $G = (V(G), E(G))$. Let $k = O\left(\frac{1}{\varepsilon^2}\right)$, and give fixed ε, k to the oracle.

Algorithm 1 The partition oracle.

Require: Fixed ε, k, with $k = O\left(\frac{1}{\varepsilon^2}\right)$.
Input: $v \in V(G)$
Output: $P[v] \leftarrow$ The name of the partition containing v, $P[v]$, such that $\forall v \in V(G)$:

1. $|P[v]| \leq k$,
2. $P[v]$ is connected.

If G is planar, then (with probability $\geq \frac{9}{10}$)

$$\left|\{(u,v) \in E : P[u] \neq P[v]\}\right| \leq \frac{\varepsilon\Delta n}{4}.$$

The oracle does not need to compute the entire partition for $V(G)$ per call, but its outputs should be consistent with the same partition across different calls.

1.2 Constructing Partition Oracles

Partition oracle plan:

1. First, consider a global partitioning algorithm (not sublinear time).
2. Next, we simulate the global partitioning algorithm locally to obtain sublinear time.

Definition 3 *A subset of vertices $S \subseteq V(G)$ is a* (δ, k)–isolated neighborhood of v *if:*

1. $v \in S$,
2. *S is connected,*
3. $|S| \leq k$,
4. *The number of edges connecting S and $V(G) \backslash S$ (the number of edges leaving S) is $\leq \delta |S|$, ie.*

$$\left| \{(u, v) \in E(G) : u \in S, v \notin S\} \right| \leq \delta |S|.$$

For the analysis below, take $\delta = \varepsilon\Delta$ and $|S| = \frac{c}{\varepsilon^2}$. Observe that for any valid partition of a planar graph, most nodes are in such a (δ, k)–isolated neighborhood. This is because by $(\frac{\varepsilon\Delta}{2}, \frac{c}{\varepsilon^2})$–hyperfiniteness,

$$\begin{aligned} E[\#\text{edges leaving } P[v]] &\leq \frac{\varepsilon\Delta/2}{\#\text{partitions}} \cdot n \cdot 2 \\ &\leq \frac{\varepsilon\Delta}{k}, \end{aligned}$$

where we multiply by 2 to count both endpoints of each edge. Thus, we can apply Markov's inequality to show that the number of edges leaving $P[v]$ for most $v \in V(G)$ is small.

1.2.1 Global Partitioning Algorithm

We now describe the global partitioning algorithm to determine partitions, which does not run in sublinear time.
Input: $G = (V(G), E(G))$,
Output: Partition P of vertices.

Algorithm 2 Global partitioning algorithm.

```
Let π_1, ··· , π_n be nodes in a random order.
P ← ∅.
for i = 1 to n do
    if π_i still in graph then
        if ∃ (δ, k)–isolated neighborhood of π_i in the remaining graph then
            S ← lexicographically first such neighborhood (type 1)
        end if
    else S ← {π_i} (type 2)
    end if
    P ← P ∪ {S}
    remove S and its adjacent edges from G
end for
```

Proof The type 1 sets S that form (δ, k)–isolated neighborhoods are good. This is because they contribute $\leq \delta|S|$ edges each, so this case contributes overall

$$\sum_{s_i \in S} \delta|s_i| \leq \delta n.$$

For type 2 $S = \{\pi_i\}$, each vertex could contribute Δ edges, leading to overall Δn, which is too many. Our goal is to show that there exists only a small number of such nodes, and in particular if there are $\leq \varepsilon' n$ such nodes, then they contribute $\leq \varepsilon' n \Delta$ overall. We will need the following lemma.

Lemma 4 *If $G' = (V(G'), E(G'))$ is a subgraph of a planar graph G such that $|V(G')| \geq \delta n$, then $\leq \frac{\varepsilon}{30}|V(G')|$ nodes of G' do not have (δ, k)–isolated neighborhoods for $\delta = \frac{\varepsilon\Delta}{30}$ and $k = \Theta\left(\frac{1}{(\varepsilon\Delta)^2}\right)$*

Sketch of Proof WLOG we can prove this for $G' = G$, since G being planar implies G' is planar.
$\Rightarrow G$ planar.
$\Rightarrow G'$ planar and thus hyperfinite
$\Rightarrow \exists$ partition such that most nodes are in (k, δ)–isolated neighborhood (this is a property implied by graph theory and then Markov's inequality)
$\Rightarrow$ Since the π_i are randomly chosen, with high enough probability π_i is in a (k, δ)–isolated neighborhood. ■

Remark It is necessary for the lemma to hold for both G and any subgraph of G, so that it can be applied on the remaining subgraphs as we remove neighborhoods from G in the algorithm.

By the lemma, type 2 is unlikely to occur too often. We conclude that the partitioning algorithm outputs a partition P with few average crossing edges over all $S \in P$. ■

1.2.2 Local Simulation of Partitioning Algorithm

We now describe the local sublinear time simulation of the global partitioning algorithm.
Input: $v \in V(G)$,
Output: $P[v]$.

Algorithm 3 Local simulation algorithm.

Require: Assume access to the location $\pi(v)$ of v in the random ordering $\pi : V(G) \to [n]$.
First, recursively compute $P[w]$ for all $w \in V(G)$ such that

1. $\pi(w) < \pi(v)$,
2. The distance between w and v is $\leq k$.

Note there are $\Delta^{O(k)}$ possible w.
if $\exists w$ such that $v \in P[w]$ **(case A) then**
  $P[v] \leftarrow P[w]$
else (case B)
  Look for lexicographically first (δ, k)–isolated neighborhood of v (ignoring nodes in $P[w]$ for smaller w's). We can disregard the time complexity of this and just do a brute force search, since the size of the input is of size $\Delta^{O(k)}$ (no dependence on n).
  if a neighborhood exists **then**
    $P[v] \leftarrow$ neighborhood,
  else
    $P[v] \leftarrow \{v\}$.
  end if
end if

Runtime: We have $\Delta^{O(k)}$ computations in the recursive step, and the runtime of each one is $O(2^{\Delta^k})$ (go back in notes on local random greedy simulation to see why). The second step is either the same runtime if case A is true, or, if case B is true, a brute force search on a graph of size $\leq \Delta^{O(k)}$, so its runtime is dominated by the recursive step. Therefore, the total runtime of this implementation is $O(2^{\Delta^k})$. ■

Remark The algorithm can actually be implemented with runtime poly(Δ/ε), proven in a recent paper. In any case, the maximum degrees Δ of planar graphs are typically bounded, so this runtime is acceptable despite seeming inefficient.

2 Monotonicity Testing

2.1 Monotonicity Testing of Lists

Definition 5 *A list of size n is* ε–close to sorted *if one can delete at most εn entries to make it sorted. Otherwise, it is ε–far from sorted.*

Question: Given a list $y_1, y_2, \cdots, y_n$ (e.g. alphabetized books in a library), how can we quickly test if the list is close to sorted?

- We wish to always pass sorted lists and fail lists that are ε–far with probability $> \frac{3}{4}$.
- Pass with high probability $\rightarrow$ ε–close
  (contrapositive) $\Leftrightarrow$
  Not fail with high probability $\rightarrow$ not ε–far

First attempt: Pick a random entry i and test that $y_i \leq y_{i+1}$ (the entry and its right neighbor are sorted).

- Bad example: Two increasing lists concatenated together, e.g. $1, 2, 3, \cdots, i, 1, 2, \cdots, n-i$.

Second attempt: Pick many random entries and pass if all in right order.

- Bad example: Ordered list with consecutive pairs flipped, so that we must delete one out of each pair to obtain a sorted list,
  e.g. $2, 1, 5, 4, 11, 7, 19, 14, 21, 20, 38, 23, 45, 39, 50, 46, 60, 57, 80, 61$.
- We need to pick $\Omega(\sqrt{n})$ entries in this case (can also show that $O(\sqrt{n})$ is enough).

Third attempt (a test that works): Fast random binary search test.

Algorithm 4 Monotonicity testing of lists.

Require: List $(y_1, \ldots, y_n)$, assumed distinct ($y_i \neq y_j$ for $i \neq j$). Ties can be broken by appending i to each y_i, which does not change the order.
Ensure: PASS or FAIL
  for $t = 1$ **to** $O(1/\varepsilon)$ **do**
    Pick a random index i and read the value y_i.
    Run binary search for y_i.
    if binary search does not find y_i at location i **then**
      return FAIL
    end if
    if binary search encounters any inconsistency **then**
      return FAIL
    end if
  end for
  return PASS

Proof Say that index i is "good" if the binary search for y_i is successful.
1) If the list is sorted, then clearly all i's are good and the test always passes.
2) Next, we show that if the list is likely to pass the test, then at least $(1-\varepsilon)n$ of the i are good.

Lemma 6 *The good elements form an increasing sequence.*

Proof If $i < j$, then some point of divergence k where the search of i went left and the search of j went right gives us $y_k < y_k < y_j$. ■

Thus, the list is ε–close to monotone. ■

Runtime: The runtime for the binary search $O(\log n)$ and we repeat for $O(1/\varepsilon)$ rounds, so the total runtime is $O(1/\varepsilon \log n)$.

2.2 Monotonicity Testing on The Boolean Hypercube

Definition 7 *Label the nodes of the n–dimensional cube by n–bit strings in $\{0,1\}^n$ such that nodes connected by an edge differ by exactly one bit and we have a partial ordering*

$$x \prec y \Leftrightarrow x_i \leq y_i, \forall i.$$

Then, $f : \{0,1\}^n \to \{0,1\}$ is monotone *if*

$$x \prec y \Rightarrow f(x) \leq f(y),$$

or equivalently increasing a bit of x does not decrease $f(x)$.

To be continued in lecture 11.