Sublinear time algorithms based on simulating greedy algorithms

Lecture 18
Ronitt Rubinfeld
Sparse Graphs

• Representation:
  – Max degree d
  – Adjacency list format

• Problems of interest:
  – Vertex cover
  – Matching
  – Dominating set
  – Set cover (on sparse set systems)
Maximal Matching

- $M \subseteq E$ is a matching if $8 (u,v), (w,x) \in M$ \{u,v\} \cap \{w,x\} = \$

- $M$ is a maximal matching if adding any edge violates the matching property

- Note:
  - Size of any vertex cover $\geq$ size of any maximal matching
  - Size of vertex cover $\leq 2 \cdot$ size of any maximal matching
Vertex cover vs. maximal matching

• Size of any vertex cover \( \geq \) size of any maximal matching
  – every matching edge must be covered by VC, but endpoints of matching are disjoint!

• Size of vertex cover \( \leq 2 \cdot \) size of any maximal matching
  – let C be all nodes that are matched by maximal matching.
  – If C is not a vertex cover, then there is an edge that could have been added to the matching (contradicting maximality).
Greedy algorithm for maximal matching

• Algorithm:
  – \( M \lessgtr \); 
  – \( 8 \; e = (u,v) \in E \)
    • If neither of \( u,v \) matched
      – Add \( e \) to \( M \)
    – Output \( M \)

• Why is \( M \) maximal?
  – If \( e \) not in \( M \) then either \( u \) or \( v \) already matched

• How big is \( M \) for graphs with max degree \( d \)?
  – Any edge in \( M \) removes < 2d others from consideration
  – Still have possible edges to add for \( \frac{n}{2d} \) rounds
Suppose you have help....

• Assume there is an oracle:
  – on input \( e \) (an edge) tells you if \( e \) is in the matching

• Why should you have such an oracle?
  – we’ll see later...
Idea for sublinear time algorithm:

- Run [Parnas-Ron] reduction algorithm:
  - i.e.,
    - Sample $O(1/²²)$ nodes
    - For each sampled node, call "oracle" on neighboring edges to decide if it is in the matching
    - Output
      \[
      \left( \frac{\text{fraction of sampled nodes in matching}}{2} \right) \cdot n + \left( \frac{²}{2} \right)n
      \]

- How do you implement oracle?
  - Idea: figure out what greedy would do
Problems with greedy

• Can have long dependency chains!
  – (see board for example)

• How can you implement the oracle?
  – Must know if adjacent edges that come before in the ordering are in the matching
  – Do not need to know anything about edges coming after
Breaking long dependency chains

• Assign random ordering to edges
  – Greedy works under any ordering
  – To show: random order has short dependency chains
Implementing oracle $\mathcal{O}$

[Nguyen Onak]

- **Preprocessing:**
  - assign random number $r_e \in [0,1]$ to each $e \in E$

- **Oracle implementation:**
  - Input: edge $e \in E$,
  - Output: is $e$ in $M$?
  - Algorithm:
    - Find all the adjacent edges of $e$, $e' \in E$, such that $r_{e'} < r_q$
    - Recursively check if any in $M$
      - If any in the matching, output NO
      - If none are in the matching, output YES
Example Run Θ
Example Run $\emptyset$ (cont.)
Example Run $\emptyset$ (cont.)
Example Run $\Theta$ (cont.)
Example Run $\emptyset$ (cont.)
Example Run $\Theta$ (cont.)

[Diagram of network with numbers and question marks]
Example Run $\emptyset$ (cont.)
Example Run $\Phi$ (cont.)
Example Run $\Theta$ (cont.)
Example Run $\emptyset$ (cont.)
Example Run $\Theta$ (cont.)
Example Run $\emptyset$ (cont.)
Example Run $\emptyset$ (cont.)
Correctness

• This algorithm simulates run of classical greedy algorithm
  – Greedy works under any ordering of edges

• Outputs estimate $t$ such that
  $$\text{MM}(G) \cdot t \cdot \text{MM}(G) + ^2 n$$
  where $\text{MM}(G)$ is size of some maximal matching
Complexity

• Claim: Expected number queries to graph per oracle query is $2^{O(d)}$

  – so total complexity is $2^{O(d)}/2^2$

– Main idea:
  • Bound probability a path of length $k$ explored:
    – Ranks must decrease along the path
    – So probability $\cdot \frac{1}{(k)!}$
Complexity

• Claim: Expected number queries to graph per oracle query is $2^{O(d)}$

• Proof:
  – $\Pr[\text{given path of length } k \text{ explored}] \cdot \frac{1}{(k)!}$
  – Number of neighbors at distance $k \cdot (2d)^k$
  – $E[\text{Number of nbrs explored at dist } k] \cdot \frac{(2d)^k}{(k)!}$
  – $E[\text{number of explored nodes}] \cdot \sum_{k=0}^{1} \frac{(2d)^k}{(k)!} \cdot e^{2d}/2d$
  – $E[\text{query complexity}] = O(d) \cdot e^{2d/2d}$

  = $2^{O(d)}$
Further work

• Always recurse on least ranked edge first gives better runtime [Yoshida Yamamoto Ito]
• More complicated argument for Maximum matching, set cover,…
• Even better results for certain classes of graphs [Hassidim Kelner Nguyen Onak]
  – Minor-free (e.g., planar, constant tree-width, non-expanding graphs…)
  – see board for more on this!!!