# Sorting, Maximal Independent Set

CS255
Chris Pollett
Mar. 8, 2006.

# Outline

- Finish Sorting
- Maximal Independent Set

# More on Sorting

- We finish the analysis of the BoxSort algorithm that we began presenting last day.
- To do this, we represent the execution of the algorithm as a tree.
  - The root is a node with all of the elements to be sorted in it
  - Internal nodes consist of sublists of nodes to be sorted. We will call these boxes.
  - Children of a given box represent sublists that need to be sorted after partitioning according to the splitter.
  - Each leaf has at most log n elements.
- We want to analyze root-leaf paths in this tree and bound the number of PRAMS steps along such a path.
- Recall based on the HW problem, doing splitting should take O(log the size of box we need to split).
- In a perfect world, at each level the expected size of the box goes down by a square root, so we get the sum $O(\log n + \log n^{1/2} + \log n^{1/4} + \ldots) = O(\log n + 1/2 \log n + 1/4 \log n + \ldots) = O((\log n)*(1/2+1/4+ \ldots)) = O(\log n)$
- We will argue even in a non perfect world that with high probability the sum of the log of the sizes of the boxes along any path is O(log n), so the runtime will be O(log n).

# Yet More on Sorting

- To see this partition the interval [1,n] into sub-intervals $I_0$, $I_1$, … We will then bound the probability that a box whose size is in $I_k$ has a child whose size is also in $I_k$.
- Fix $\gamma$ and d such that $1/2 < \gamma < 1$ and $1 < d < 1/\gamma$. For positive integers k, let $\tau_k = d^k$, $\rho_k = n^{\gamma\wedge k}$. Define $I_k = [\rho_{k+1}, \rho_k]$.
- n = (log n)^{log n $\log_{\log n}2$}. So if $\gamma^k > 1/\log n \log_{\log n}2$, then $\rho_k = n^{\gamma\wedge k} <$ log n. This will happen for some k< c log log n.
- So we will only be interested in O(log log n) many intervals $I_k$.
- For a box B in the tree, we let $\alpha(B) = k$ if $|B|$ is in $I_k$.
- In terms of our notation, the time to split Box B is $O(\log \rho_{\alpha(B)})$ .
- For a root-leaf path P = ($B_1$, …, $B_t$), the runtime is given by $\sum_{j=1}^{t} log \rho_{\alpha(B_j)}$
- The total runtime of the algorithm will be O of this plus log n (to sort the leaves).
- Define the event $E_P$ to be that the sequence $\alpha(B_1)$, … $\alpha(B_t)$ does not contain the value k more than $\tau_k$ times for 1<= k <= c log log n.
- If $E_P$ holds then the number of PRAM steps on path P will be

$$O(\log n + \sum_{j=1}^{\infty} \tau_k \gamma^k \log n)$$

# The End of Sorting

- Since $\tau_k = d^k$ and $d\gamma < 1$, this sums to O(log n).
- So it suffices to show $E_P$ happens with high probability.

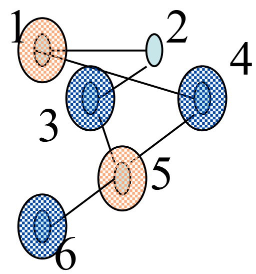
**Lemma** There is a constant b>1 such that $E_P$ holds with probability 1- $\exp(-\log^b n)$.

**Proof** uses Chernoff bounds and is omitted. (Chernoff Bounds: If X is the sum of independent random variables which outputs either 0 and 1, the latter with probability p, then for a $0 <= \theta <= 1$, prob$\{X >= (1+\theta)pn\} < e^{-\theta^2 pn/3}$)

- From this we can conclude:

**Theorem** There is a constant b >1such that probability at least 1-$\exp(-\log^b n)$, the algorithm BoxSort terminates in O(log n) steps.

- So the algorithm is an ZNC algorithm.




# Maximal Independent Set

- Let G(V,E) be an undirected graph with n vertices and $m = \Omega(n)$ edges. A subset I of V is said to be **independent** in G if no edge in E has both ends in I.
- Equivalently, if $\Gamma(v)$ is the set of vertices connected to v, then I is independent if for all v in I, $\Gamma(v) \cap I = \emptyset$.
- An independent set is **maximal** if it is not a proper subset of another independent set in G.
- The red nodes and the blues nodes in the graph above are two different maximal independent sets in the same graph. Notice the blue set has more nodes.
- The problem of finding a *maximum* independent set (the independent set with the most nodes) is NP-hard.
- In contrast the finding a maximal independent set is O(m) time:

Greedy MIS:

Input: Graph G(V,E) with V = {1,..,n}

Output A maximal I contained in V.

1. I <-- $\emptyset$
2. For v=1 to n do If $\Gamma(v) \cap I = \emptyset$ then I <-- $I \cup \{v\}$.

# More on Maximal Independent Set

- Greedy-MIS is very sequential in nature.
- For the graph on the last slide the algorithm outputs the Maximal Independent Set (MIS) {1,3,6}.
- Notice the two other independent we had previously drawn are {1,5} and {3,4, 6}. According to dictionary (lexicographical) order {1, 3, 6} is before {1,5} is before {3, 4, 6}.
- It turns out Greedy-MIS always outputs the lexicographically first MIS (LFMIS).
- LFMIS is a P-complete problem (with respect to log-time poly-porcessor PRAM reductions).
- So it is known that an NC algorithm for LFMIS would imply P=NC. (An open problem).
- We will describe an RNC algorithm for MIS and later show how to derandomize it to an NC algorithm.
- The maximal set we output won't be the lexicographically first one.

# Yet More on Greedy MIS

- Consider the following variant of Greedy MIS:
  1. I <-- $\emptyset$
  2. Pick any vertex v, add v to I, delete v and $\Gamma(v)$ from the graph.
- Choosing v to be the lowest numbered vertex present in the graph leads to the same outcome as Greedy MIS.
- The basic idea of our parallel algorithm is to general this to find an independent set S, add S to I and delete S and $\Gamma(S)$.
- We want to choose an independent set such that $S \cup \Gamma(S)$ is large to keep the number of iterations small.
- To do this we ensure the number of edges incident to $S \cup \Gamma(S)$ is a large fraction of the total remaining edges.
- To find such an S, we pick a large random set of vertices R contained in V. R won't usually be independent. If we bias the sampling in favor of vertices with low degree, we can hope that few will have both endpoints in R. For those edges which have both endpoints in R, we delete the one of lower degree. This gives an independent set.

# Parallel MIS

Input: G=(V,E)

Output: A maximal independent set I contained in V

1. I <-- $\varnothing$
2. Repeat
   a) For all v in V do in parallel
      If d(v) = 0 then add v to I and delete v from V.
      else mark v with probability 1/2d(v).
   b) For all (u,v) in E do in parallel
      if both u and v are marked
      then unmark the lower degree vertex.
   c) For all v in V do in parallel
      if v is marked then add v to S
   d) I <-- I $\cup$ S
3. Delete $S \cup \Gamma(S)$ from V and all incident edges from E
4. Until V is empty.

# Analysis

- Each iteration of the above takes O(log n ) time on an EREW PRAM with O(n+m) processors.
- We want to bound the number of iterations we do.
- Call a vertex v *good* if it has at least d(v)/3 neighbors of degree no more the d(v); otherwise, the vertex is bad. An edge is good if one of its endpoints is good and is bad otherwise.
- A good vertex is quite likely to have one of its lower degree neighbors in S and so is likely to be deleted from V.
- We will argue next day the number of good edges is large, and since good edges are likely to be deleted, a large number of edges will be deleted each iteration.