15.859-Z Algorithmic Superpower Randomization November 10, 2015

Lecture 19

Lecturer: Bernhard Haeupler *Scribe: Dabeen Lee*

1 Overview of Distributed Computing

In distributed computing, we have many machines in a network. One machine might not have all the data. Hence, machines need to communicate over the network to exchange data and coordinate together so that the whole network becomes able to compute something.

We introduce two models of distributed computing; LOCAL and CONGEST models. They are similar, but CONGEST model has a restriction on the number bits that one can send to another per round. LOCAL model has the following setting.

1) A network expressed as a graph $G = (V, E)$.
   n nodes of G correspond to n computers in a distributed setting, and an edge (u, v) implies that u can talk to v. However, we cannot use the global topology of the network. Initially, one node knows only the neighbors of it.

2) Synchronized communication model.
   We definitely need a communication model. The easiest thing we can think of is synchronized communication model. In this model, every node can send messages to all of its neighbors in each round of communication. Communication time between two computers sometimes varies over edges, so it is possible that the network gets to slow down because of some edges. To make communication synchronized, certain amount of overhead is sometimes necessary.

3) Amount of data that one computer send to another.
   A computer in LOCAL model can send as much as it wants, while the maximum number of bits that a computer in CONGEST model can send is $O(\log n)$.

4) Complexity measure.
   A typical complexity measure is the number of rounds needed to compute something.

2 Classical Examples

Distributed computing is plausible only when we do not need to know the topology of the whole network. There are many problems that we need to look at the whole network. For example, to solve SHORTEST-PATH, MST, and GRAPH-CONNECTIVITY, we need the topology of the whole network.

If we use LOCAL model to check whether a given graph is connected, we sometimes need $\Omega(n)$ rounds. That is because a vertex u can talk to its neighbor in one round, so another vertex w which is far away from u needs to wait for a long time until it gets the message sent from u. Essentially, the complexity of one problem depends on how far we need to look out from one node in order to solve the problem. As mentioned, we need to look at the entire network to check connectivity of a network.

In this lecture, we are interested in local problems that do not need large number of rounds to solve, so we do not have to look at the whole network in those problems. In general, poly-logarithmic number of rounds is desirable.

2.1 d-Clustering

We have a graph $G = (V, E)$. Now, we would like to find a subset C of V such that the distance between every two nodes in C is greater than d. We call each node in C a "center". Besides, assign each node to one center $c \in C$ if the distance between them is at most d. Then, in a d-clustering solution, each node is not too far from its center and no two centers are too close.

For each node, we need to check other nodes that have distance up to d from it. If d is small, the number of rounds should be also small.

2.2 $(\Delta + 1)$-Coloring

A *proper vertex coloring* is a coloring of nodes where two end nodes of each edge are colored with different colors. $(\Delta + 1)$-coloring is a proper vertex coloring using only $\Delta + 1$ colors.

2.3 Maximal Matching

A *matching* M of a graph $G = (V, E)$ is a subset of E such that no node is contained in more than two edges in M. A *maximal matching* M' is a matching such that $M' \cup \{e\}$ is no longer a matching for each $e \notin M'$. That means we cannot locally improve a maximal matching.

2.4 Remark

For all of those three problems, it is fairly easy to verify a solution locally. For $(\Delta + 1)$-coloring, we just need to look at one node and all of its neighbors. Hence, one node can talk to each of its neighbors to check if they do not use the same color.

In addition, the above problems are somewhat special in distributed computing in a sense that they do so-called symmetry breaking in distributed setting. For instance, in a matching problem, there are many different ways to match two nodes to get a matching. However, we need to identify one way of matching nodes, and we do this locally to get a solution which globally satisfies the desired property.

In an iteration of the parallel algorithm for Lovász Local Lemma, we find a maximal independent set which consists of violated events. Any violated event can be selected, but we cannot select two events at the same time if they are adjacent in the dependency graph. In that case, we need to choose one of those two. When there is symmetry between those two events, it is hard to choose which to be selected for our independent set. Hence, we need a way of breaking symmetry.

3 Maximal Independent Set

In this lecture, we learn how to get a maximal independent set using a distributed algorithm. Again, the number of rounds of the algorithm is an important measure of efficiency.

The first strategy we can think of is that we repeatedly pick nodes randomly until we get a maximal independent set. However, we do not know the global topology of our graph and sometimes we do not even know the number of nodes in the graph. That means it is hard to pick a random node in distributed setting. Moreover, it might take potentially $O(n)$ rounds.

The following is the desired distributed algorithm for finding a maximal independent set.

3.1 Luby's Maximal Independent Set Algorithm

Consider the following algorithm.

Algorithm 1: A maximal independent set algorithm

```
Input : A graph $G = (V, E)$;
Output : A maximal independent set $M$;
Set $M := \emptyset$;
while $V \neq \emptyset$ do
    Compute an independent set $S$ in $G = (V, E)$;
    $M := M \cup S$;
    $V := V \setminus (\Gamma^+(S))$;
    $E := E \setminus \{(u, v) \in E : u \in \Gamma^+(S)\}$;
end
Return $M$;
```

$\Gamma^+(S)$ denotes the set $S \cup \{v \in V : \exists u \in S, (u, v) \in E\}$. It is easy to show that the above algorithm correctly finds a maximal independent set. First, final M is clearly an independent set, because we remove all the neighbor nodes of S from V when we add an independent set S to M. Second, final M is maximal, because each node in V is either selected as a node of an independent set S or removed as a node of $\Gamma^+(S) \setminus S$. Thus, adding a node in $V \setminus M$ to M creates an edge inside.

Still, we need to figure out how to get an independent set S at each iteration. This is the part where we consider distributed setting in the algorithm. We introduce two possible approaches in this lecture.

First, we use coin flip. We can flip a coin for each node. Each node sends its coin flip result to its neighbors and receives their coin flip results in one round. If a node u gets HEAD(=1) and all the neighbors get TAIL(=0), then we choose u and add it to S. If the probabilities of getting HEAD and TAIL are equal, then the probability that a node u is selected is pretty row since some neighbor nodes of u also get HEADs from their coin flips. In other words, there is high level of symmetry in this case. In fact, the probability of u being selected for S is $(1/2)^{|\Gamma^+(v)|}$. Then the number of nodes selected for S is $\sum_v (1/2)^{|\Gamma^+(v)|}$ in expectation. Notice that the number is quite low, so the number of total iterations needed would be really big. Instead of using a fair coin, we can use a biased coin to resolve this problem. If the probability of getting HEAD is ϵ, then the probability that v is selected for S is $\epsilon(1-\epsilon)^{|\Gamma(v)|}$ which can be made bigger than $(1/2)^{|\Gamma^+(v)|}$.

Another approach is to use random priority numbers for nodes. For each node v, pick a number from $[R]$ uniformly at random for its priority number π_v. Then, each node sends its priority number to its neighbors and receives their priority numbers in one round. The number of bits that one node sends is just $O(\log R)$. If the priority number π_u of a node u is greater than that of any neighbor, then we choose u and add it to S. If R is 1, then $[R]$ is just $\{0, 1\}$ and thus it is equivalent to coin flip. As we saw previously, small value of R possibly generates lots of symmetry. Note the following.

$$Pr[\exists (u, v) \in E \text{ s.t. } \pi_u = \pi_v] \leq \sum_{(u,v)\in E} Pr[\pi_u = \pi_v] = |E|/R.$$

Therefore, if we choose R as n^c for some sufficiently large c, then all nodes get distinct priority numbers with high probability. Thus, we need just $O(\log n)$ bits to communicate, so CONGEST model is also plausible in that case. Based on this observation, we introduce a distributed algorithm for finding a maximal independent set which is called Luby's algorithm.

Algorithm 2: Luby's maximal independent set algorithm

```
Input : A graph $G = (V, E)$;
Output : A maximal independent set $M$;
Set $M := \emptyset$;
while $V \neq \emptyset$ do
    Each node $v$ chooses a value $\pi_v$ from $[n^c]$ uniformly at random;
    Let $S$ be the set of nodes such that each of them has bigger priority
    number than any of its neighbor ($v \in S$ if $\pi_v > \pi_u$ for $u \in \Gamma(v)$);
    $M := M \cup S$;
    $V := V \setminus (\Gamma^+(S))$;
    $E := E \setminus \{(u, v) \in E : u \in \Gamma^+(S)\}$;
end
Return $M$;
```

3.2 Analysis of Luby's Algorithm

Clearly, Luby's algorithm gives a maximal independent set. What remains is to show that the algorithm is efficient in a sense that it takes a small number of rounds. In this lecture, we prove that Luby's algorithm needs $O(\log n)$ rounds in expectation. In fact, it needs $O(\log n)$ rounds with high probability. We will show this in the next lecture.

Lemma 1 *Expected number of edges that are removed in one iterations is at least $|E|/2$ where E denotes the set of remaining edges in the graph.*

Proof idea :

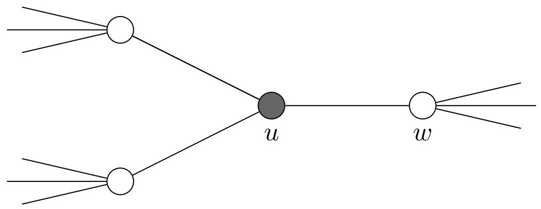

Figure 1: $u \in S$ and edges to be removed

In Figure 2, assume u is selected for S. Black nodes are in S, and white nodes are in $\Gamma(S)$. When we delete u and $\Gamma(u)$ from V, edges which are incident to u and the nodes in $\Gamma(u)$ are removed. Note that the number of edges incident to the nodes in $\Gamma(u)$ is at least the number of edges incident to u. We will show that counting only the edges incident to the nodes in $\Gamma(u)$ is sufficient.

Proof Let $u \in S$ and $w \in \Gamma(u)$. Then the edges incident to w will be removed after this round, because we selected u to be in S. It is also possible that there exists another vertex v in S such that v is adjacent to w. That means we can also say that the edges incident to w will be removed, because $v \in S$. Think of an edge (w, w') where $w' \neq u, v$.

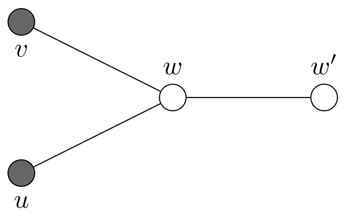

Figure 2: $u \in S$ and edges to be removed

Then, the edge (w, w') will be removed because of u and v. However, we want to count this case exactly once. Among $u, w \in S$ such that u and v are adjacent to w, we pick a node with the largest priority number. Let's say it is u. Then we say the edge (w, w') is removed because "u rescues w". In other words, we say "u rescues w" if $(u, w) \in E$ and u has the largest priority value among $\Gamma^+(u) \cup \Gamma^+(v)$. It is possible that there exists $u' \in S$ such that (w, w') is removed because "u' rescues w'". The important thing is that we count the edge (w, w') at most twice using this method. That is because at most one u satisfies "u rescues w".

If (w, w') is removed because "u rescues w", then actually all the edges adjacent to w are removed because "u rescues w". Besides, π_u is the largest number among the priority numbers of all nodes in $\Gamma^+(u) \cup \Gamma^+(w)$. The probability that u gets the largest priority number among all nodes in $\Gamma^+(u) \cup \Gamma^+(w)$ is exactly $\frac{1}{d(u)+d(w)}$ where $d(r)$ denote the degree of a vertex $r \in V$. Therefore, $\frac{d(w)}{d(u)+d(w)}$ edges which are adjacent to w are removed because "u rescues w" in expectation.

Let's formally define the indicator random variable $X_{u \to w}$ for the event "u rescues w". Then $Pr(X_{u \to w} = 1) = \frac{1}{d(u)+d(w)}$. Therefore, the expected number of edges removed because "u rescues w" for $u, w \in V$ is

$$\begin{aligned}
\frac{1}{2} \sum_{u,w \in V:(u,w) \in E} Pr(X_{u \to w}) d(w) &= \frac{1}{2} \sum_{(u,w) \in E} Pr(X_{u \to w} = 1) d(w) + Pr(X_{w \to u} = 1) d(u) \\
&= \frac{1}{2} \sum_{(u,w) \in E} \frac{d(w)}{d(u)+d(w)} + \frac{d(u)}{d(u)+d(w)} \\
&= |E|/2
\end{aligned}$$

The fraction $\frac{1}{2}$ comes from the fact that our counting method counts each edge at most twice. It is obvious that the expected number of edges removed in one iteration is at least the expected number of edges removed because "u rescues w" for $u, w \in V$. Hence, at least $|E|/2$ edges are removed in expectation. ■