EE 382V: Parallel Algorithms **Summer 2017**

Lecture 19: July 21

Lecturer: Vijay Garg *Scribe: Abed Haque*

19.1 Introduction

During this lecture we covered the following topics:

- The Maximal Independent Set (MIS) Problem
- Luby's Algorithm

This set of lecture notes will briefly re-examine the topics covered in this lecture, in the order in which they appeared during class.

19.2 The Maximal Independent Set Problem

Suppose that we are given an undirected graph, $G = (V, E)$. A set of vertices $V' \subseteq V$ is said to be *independent* if there is no edge between any two vertices V'. There can be many independent sets.

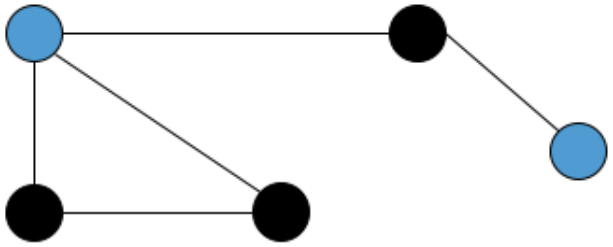

Figure 19.1: A possible independent set (blue vertices)

An independent set is considered to be *maximal* if there is no independent set that strictly contains V'. Figure 19.1 is maximal since no other vertex can be added to V' while still being an independent set. The largest possible maximal independent set is called a *maximum* independent set. Finding the maximum independent set is NP-hard, so we will only focus on how to find maximal independent sets instead.

19.2.1 Greedy Sequential Algorithm For MIS

```
1  I := ∅// Start with an empty independent set
2  for v := 1 to n do
3     if I ∩ T(v) = ∅ then
4          I := I ∪ {v}
```

$T(v)$ represents the neighbors of v. This has a time complexity of *O(n)*.

19.3 Luby's Algorithm

Luby's algorithm is a randomized parallel algorithm to find the maximal independent set. The input is a graph $G = (V, E)$ and the output is the independent set I. The steps are as follows:

```
1. I := ∅
2. repeat
  2.1 for all v ∈ V in parallel do
        if (d(v) = 0) then add v to I and delete from V   //corner case
        else mark v with probability 1/(2d(v))
  2.2 for all (u, v) ∈ E do in parallel
        if both u and v are marked then unmark the lower degree vertex
  2.3 for all v ∈ V in parallel
        if v is marked then add v to S; // S is the vertices in this iteration
  2.4 I := I ∪ S
  2.5 delete S ∪ T(S) from V and all incident edges to S ∪ T(s) until V = ∅
```

Intuitively, the nodes with fewer neighbors will be able to add more nodes. Above, $d(v)$ represents the degree of v. When $d(v)$ is large, there are lots of neighbors, and we can say that they are rich (socially). The richer the vertex, the smaller its probability of being marked is. Once the entire graph is empty we're done.

Each step of this algorithm is done in constant time, in parallel. We want to show that in log(n) iterations that we'll be done (with a high probability).

A vertex $v \in V$ is known as a *good vertex* if it has at least $d(v)/3$ neighbors of degree no more than $d(v)$. A vertex is *bad* if it is not *good*. An edge is *good* if at least one of its endpoints is a good vertex.

19.3.1 Overall Strategy

1. There is a constant probability of a good vector being deleted

2. There are a large number of good edges. If a constant fraction of good edges are deleted *and* good edges form a constant fraction of total edges, then a *constant fraction* of all edges gets deleted in every iteration. If the number of edges are reduced by a factor of 2 every time, it will take $log_2 n$ iterations. If we reduce the number of edges by a factor of 100 every time, it will take $log_{100} n$ iterations. (But we really don't care because its $O(log(n))$.

This parallel algorithm was a breakthrough!

19.3.2 Proofs

Let's prove part 1 of the overall strategy.

Lemma: Let $v \in V$ be a good vertex with degree $d(v) > 0$. the probability that some vertex $w \in \Gamma(v)$ gets marked with at least $1 - e^{-1/6}$.

Proof: If v is good $\Rightarrow$ there are at least $d(v)/3$ neighbors with degree at most $d(v)$

1. The probability of one of these neighbors not being marked is $\leq 1 - 1/(2d(v))$

2. The above is independent, therefore we can multiply probabilities. The probability that none of these neighbors are marked is at most $\leq (1 - 1/(2d(v)))^{d(v)/3} \leq e^{-1/6}$

Let's bring some special attention to this last inequality, as it is very simple, but perhaps the most useful inequality in randomized algorithms.

$$1 + x \leq e^x \text{ for all } x$$

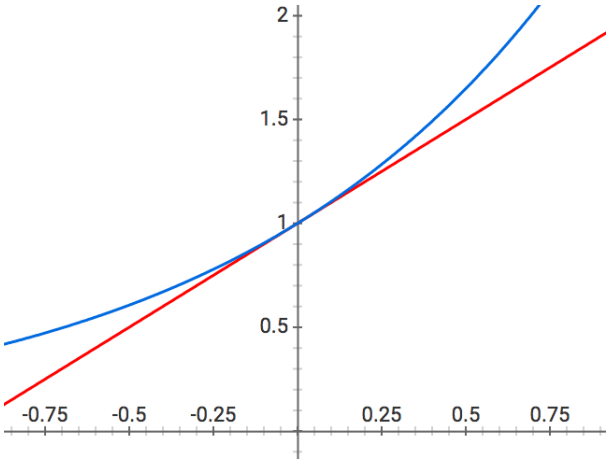

Figure 19.2: $y = e^x$ (blue) and $y = 1 + x$ (red)

Now back to the proof. Let's first multiply the exponent portion of the probability by 2:

$$(1 - 1/(2d(v)))^{2d(v)/6} \leq e^{-1/6}$$

Now let's use the above inequality and substitute.

$$(1 - 1/x)^x \leq e^{-1}$$

By substitution $n = 2d(v)$ we get the desired result:

$$(e^{-1})^{1/6} < e^{-1/6}$$

We have shown constant probability of getting marked! Now let's show that the probability of getting *unmarked* is also constant.

Lemma: During any iteration, if a vertex v is marked, then it is selected to be in S with a probability of at least $1/2$.

Proof: The vertex w is not selected only if one of its neighbors with degree $\geq d(w)$ is also marked. Each such neighbor is marked with probability $\leq 1/(2d(w))$. By union bound, the probability that a marked vertex is selected to be in S is $\leq \sum_v (1/(2d(w)) = 1/2$.

Lemma: In a group $G(V, E)$, the number of good edges is at least $|E|/2$

Proof: A vertex is bad if at least $2/3$ of its neighbors are richer. Direct the edges from a lower degree end-point to a higher degree endpoint.

$$d_o(v) - d_i(v) \geq d(v)/3 = (d_o(v) + d_i(v))/3$$

Now let's sum everything. The total degree of bad vertices is equal to

$$2e(V_B, V_B) + e(V_B, V_G) + e(V_G, V_B)$$

The first term indicates edges going from V_B to V_B and thus are counted twice. This can be rewritten as:

$$= \sum_{v \in V_B} (d_o(v) + d_1(v))$$

Now let's use the inequality from before:

$$\begin{aligned} &\leq 3 \sum_{v \in V_B} (d_o(v) - d_1(v)) \\ &= 3(e(V_B, V_G) - e(V_G, V_B)) \\ &\leq 3(e(V_B, V_G) + e(V_G, V_B)) \end{aligned}$$

Resulting in:

$$e(V_B, V_B) \leq e(V_B, V_G) + e(V_G, V_B)$$

This is saying that edges going from bad to bad are one less than good edges. Bad edges $\leq$ good edges implies that good edges $\geq |E|/2$, which is what we were trying to show.

Now we have a theorem that says that since a constant fraction of the edges are on good vertices, and good vertices get eliminated with a constant probability, the expected number of edges eliminated during an iteration is a constant fraction of the current set of edges. If a vertex gets marked, they will stay marked with a probability of $1/2$.

19.4 Euler Tour Technique

Given some tree, let's say that we want to find the height of the tree. If it was a balanced tree, this would be easy to do. But what if the tree was skewed? The height may not be $log(n)$ in this case. The Euler Tour Technique is a way to find the height of a tree in $log(n)$ time independent of the height.

Let's think of a linked list as a lopsided tree. With pointer jumping, we were able to find the length of the linked list in $log(n)$ time. We can use a similar technique for a tree if we represent it as a linked list.

Given a directed graph, a Euler Tour is a walk along the edges of the graph, starting from any node v, in which every edge is traversed exactly once and ends at the vertex v. A graph has an Euler Circuit if and only if every node has an even degree (an even number of edges).

We can view a tree as a linked list of edges. Let's construct a directed graph from a tree and then define a Euler Tour for the graph. It is sufficient to specify the next edge (successor) for every edge e in the directed graph. Let v_0, v_{m-1} be the neighbors of u. Then $succ(v_i, u)$ is given by $(u, v_{(i+1 \ mod \ m)})$. (This is depth first search).

19.4.1 Implementation of Euler Tour on a Graph from a Tree

Given the tree T shown in Figure 19.3 below, we will construct a Euler Tour. Figure 19.4 shows the adjacency list representation of the T. Now let's make two modifications to Figure 19.4. First, lets make the lists circular instead of null terminating. Second, lets put a bidirectional pointer between edges that represents the same undirected edge, represented by a blue dotted line. This results in the circular adjacency list in Figure 19.5.

In this example $succ(1, 4) = (4, 2)$. In Figure 19.5, *succ* is the element that is shown by the blue dotted edges followed by black edges. So we can can construct the following Euler Tour directly from Figure 19.5:

$(1, 4) \rightarrow (4, 2) \rightarrow (2, 4) \rightarrow (4, 3) \rightarrow (3, 4) \rightarrow (4, 1) \rightarrow (1, 5) \rightarrow (5, 1) \rightarrow (1, 4)$

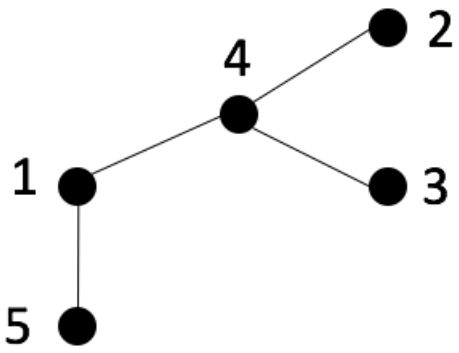

Figure 19.3: A tree T

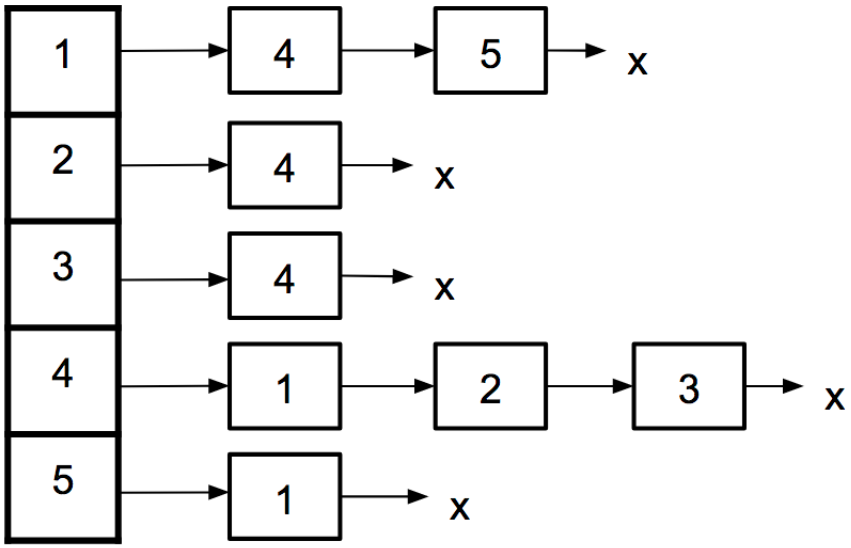

Figure 19.4: The adjacency lists of the tree T

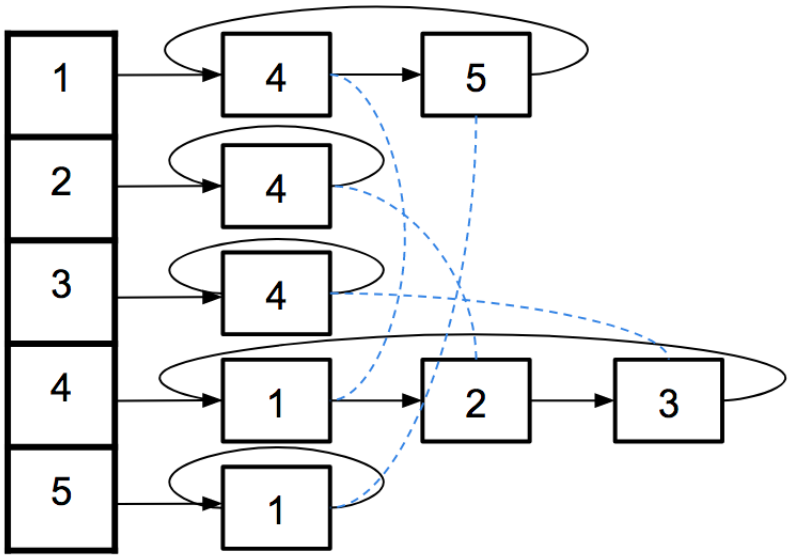

Figure 19.5: The circular adjacency lists of tree T with additional pointers