CPSC 536N: Randomized Algorithms 2011-12 Term 2

Lecture 4

Prof. Nick Harvey *University of British Columbia*

In today's lecture we will see two applications of negative binomially distributed random variables.

1 Example: Quicksort

Quicksort is one of the most famous algorithms for sorting an array of comparable elements. (We assume for simplicity that the array has no equal elements.) The quicksort algorithm is recursive. In each recursive call it picks a pivot, then partitions the current array into two parts: the elements that are strictly smaller than the pivot, and the elements that are at least the pivot. This partitioning process takes time that is linear in the size of the current array. It then recursively sorts the two parts, stopping the recursion when the current array consists of a single element.

It is well-known that quicksort probably takes $O(m \log m)$ time to sort an array of length m if each partitioning step chooses the pivot element uniformly at random from the current array. There are many ways to prove this fact. We now give a short proof using Chernoff bounds applied to random variables with the negative binomial distribution.

Let m be the size of the original input array. Notice that the total amount of work done by all subroutines at level i of the recursion is $O(m)$, since each element of the input array appears in at most one subproblem at level i. So we obtain a runtime bound of $O(mL)$ if we can show that the maximum level of any recursive subproblem is L.

Every leaf of the recursion corresponds to a distinct element of the input array, so there are exactly m leaves. We will show that every leaf has level $O(\log m)$. To do so, fix a particular element x of the input array and consider all partitioning steps involving x. Intuitively, we would like each recursive call to partition the current array into two halves of nearly equal size. To formalize this, we say that a partitioning step is "good" if it partitions the current array into two parts, both of which have size *at least one third* of that array. So the probability of being good is $1/3$.

Each good partitioning step shrinks the size of the current array by a factor of $2/3$ or better, so after $\log_{3/2}(m) < 3 \ln m$ good partitioning steps the current array has size at most 1. So x can be involved in at most $3 \ln m$ good partitioning steps, irrespective of the random decisions. Our only worry is that x could be involved in many bad partitioning steps. We can upper bound the number of partitioning steps involving x using a negative binomially distributed random variable: the number of trials needed to see $k = 3 \ln m$ successes when the probability of success is $1/3$.

Recall that in the previous lecture we used the Chernoff bound to prove the following tail bound for negative binomially distributed random variables.

Claim 1 *Let Y have the negative binomial distribution with parameters k and p. Pick $\delta \in [0,1]$ and set $n = \frac{k}{(1-\delta)p}$. Then $\Pr[Y > n] \leq \exp\big(-\delta^2 k / 3(1-\delta) \big)$.*

We apply this claim with $\delta = 3/4$, so $n = 36 \ln m$. Then the probability that more than n trials are needed to see k successes is at most $\exp(-\delta^2 (3 \ln m)/3(1-\delta)) = m^{-2.25}$. Applying a union bound over all m elements of the array, the probability that any leaf of the recursion has level greater than $36 \ln m$ is at most $m^{-1.25}$. Therefore the running time is $O(m \log m)$ with probability at least $1 - m^{-1.25}$.

2 Peer-to-peer Systems

Around 10 years ago, there was a lot of interest in "peer-to-peer systems", both academically, and in the real world. A peer-to-peer system is a decentralized approach for organizing computers and data in a distributed system.

On the academic side, there was interesting work giving novel ways to organize data and nodes in a distributed system (Consistent Hashing, Chord, Pastry, Kademlia, etc.). In the real world, many large peer-to-peer systems were developed, primarily for file sharing (Gnutella, Kazaa, BitTorrent, etc.). Several large technology companies were formed based on these technologies (e.g., Akamai and Skype).

The rough design goals of these peer-to-peer systems are as follows.

- Every node should maintain connections to a small number of other nodes. (In other words, the graph of connections between nodes should have small degree.)
- No node knows who all the other nodes in the system are, or how they are connected, or even the number of nodes.
- Any node A should be able to send a message to any other node B by traversing few intermediate nodes. (In other words, the graph of connections between nodes should have small diameter.) Moreover, the node A should be able to efficiently *find* a short path from itself to B.
- We assume that nodes are mostly cooperative: if node A asks node B to take some action, e.g., form a connection, or forward a message onwards, then node B will oblige.

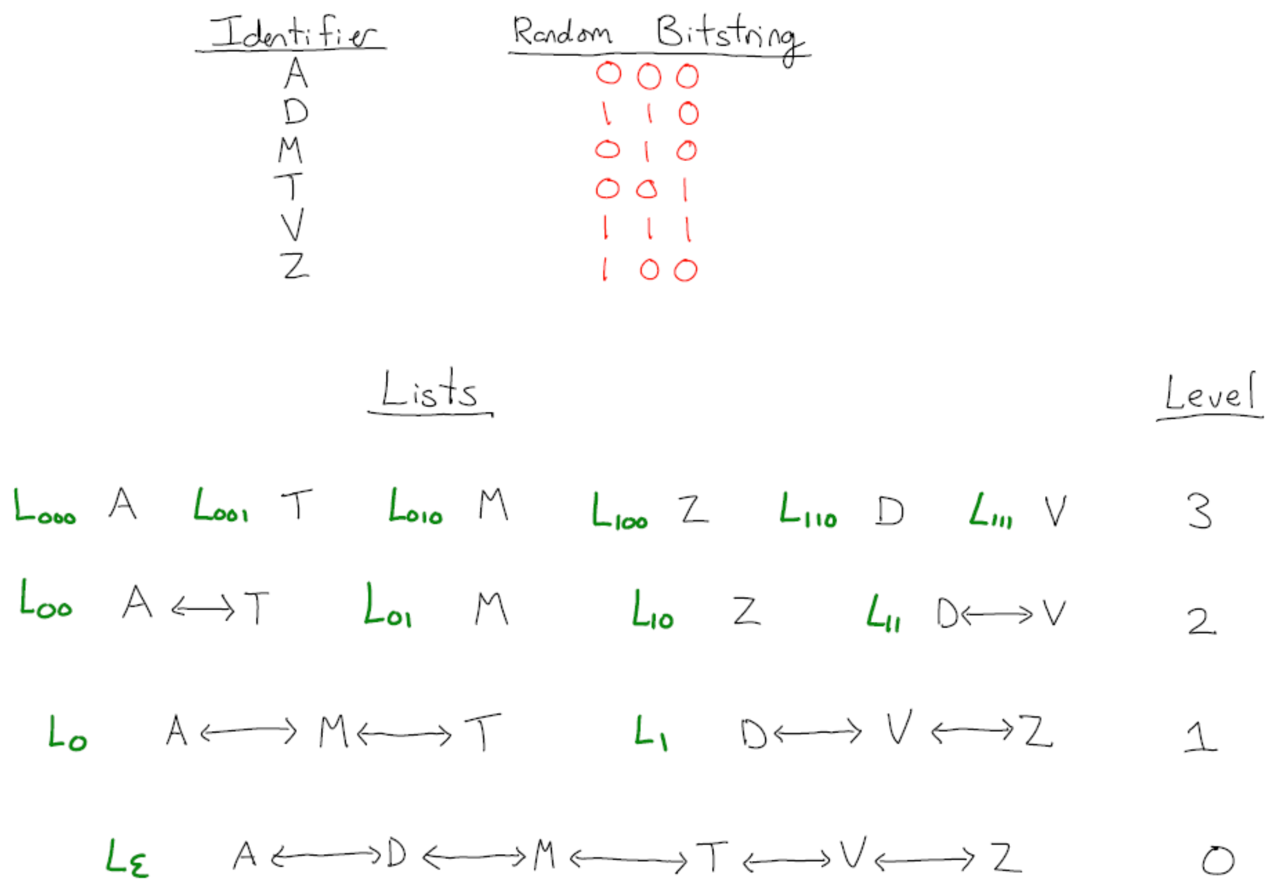
2.1 Example: SkipNet

We will discuss a system SkipNet which meets these design goals. Specifically, in a system with n nodes, every node will maintain connections to $O(\log n)$ other nodes, and any node can send a message to any other node while traversing only $O(\log n)$ intermediate nodes. The system's design is based on ideas from theoretical computer science, in particular the dictionary data structure known as Skip Lists. We discuss this system because its analysis illustrates the usefulness of random variables with the negative binomial distribution.

Suppose there are n nodes in the peer-to-peer system. Every node has a string identifier x (e.g., $x =$"www.cs.ubc.ca") and a random bitstring $y \in \{0,1\}^m$, for some parameter m whose value we will choose later.

The nodes are organized into several doubly-linked lists. Each node is a member of several of these lists, but not all of them. Each of these lists is sorted using the nodes' string identifiers. Every pair of nodes that are adjacent in these lists forms a bidirectional network connection.

Formally, for every bitstring z of length at most m, there is a list L_z. We denote the length of bitstring z by $|z|$. The list L_z contains all nodes for which z is a prefix of their random bitstring. We say that the list L_z has "level" equal to $|z|$. For example, for the empty bitstring $z = \varepsilon$, there is a list L_ε which contains *all* nodes, sorted by their identifiers. This list has level 0. At level 1 there are two lists, L_0 and L_1, each of which contain a subset of the nodes: the list L_0 contains all nodes whose random bitstrings start with a 0, and L_1 contains all nodes whose random bitstrings start with a 1. So, for each level, every node belongs to exactly one list at that level.


Identifier
Random Bitstring
A 000
D 110
M 010
T 001
V 111
Z 100
Lists
Level
L000 A L001 T L010 M L100 Z L110 D L111 V 3
L00 A ⟷ T L01 M L10 Z L11 D ⟷ V 2
L0 A ⟷ M ⟷ T L1 D ⟷ V ⟷ Z 1
Lε A ⟷ D ⟷ M ⟷ T ⟷ V ⟷ Z 0

Claim 2 *Let $z \in \{0,1\}^*$ be a bitstring of length $|z| = k$. Then the expected size of the list L_z is $n/2^k$.*

PROOF: Consider any node and let its random bitstring be y. The probability that z is a prefix of y is exactly $1/2^k$. □

This suggests that for most bitstrings z with $|z| \gg \log n$ the list L_z is empty. The following claim makes this more precise.

Claim 3 *Let $k = 3\log_2 n$. With probability at least $1 - 1/n$, every list L_z with $|z| \geq k$ contains at most one node.*

PROOF: Note that if two nodes belong to different lists at level i then they also belong to different lists at every level $j \geq i$. So it suffices to prove the claim for the case $|z| = k$.

Consider two nodes with random bitstrings y_1 and y_2. These nodes belong to the same list at level k if and only if y_1 and y_2 have equal prefixes of length k, i.e., the first k bits of y_1 equals the first k bits of y_2. So the probability that these two nodes belong to the same list at level k is 2^{-k}. By a union bound,

$$\Pr[\text{any two nodes belong to the same list at level } k] \;<\; n^2 2^{-k} \;=\; 1/n.$$

□

Recall that we only create network connections between pairs of nodes that are adjacent in any list. So the previous claim has two implications. First of all, we should choose $m = 3\log_2 n$. There is no point in creating lists at any level higher than $3\log_2 n$ because they will almost certainly contain at most 1 node, and therefore will not be used to create any network connections between nodes. Second of all, since every node belongs to m lists, it has only $O(m)$ neighbors in total, and therefore participates in only $O(\log n)$ network connections. This satisfies our first goal in the design of the peer-to-peer system.

Sending Messages. It remains to discuss how a node can send a message to any another node. Recall that this will happen by a sequence of intermediate nodes forwarding the message towards the final destination. The simplest way to do this would be to use the list L_ε at level 0. Since every node belongs to this list, a node can just send the message to his neighbor until the message arrives at the destination. Note that this process does not involve "flooding": in our example above, if D wants to send a message to T then the message would traverse $D \to M \to T$ and would not be sent to nodes A, V or Z.

However, that process of forwarding messages along the list L_ε is not very efficient. If there are n nodes in the system, then the message might need to be forwarded $\Theta(n)$ times before arriving at its destination. (The Skype network has tens of millions of active users, so forwarding each message millions of times would not be very desirable!)

To send messages more efficiently, we will make use of the other lists. The main idea is: a list at a high level has few nodes, and those nodes are essentially uniformly distributed amongst all nodes, so the connections formed by high-level lists allow one to "skip" over many nodes in the L_ε list. To be a bit more precise, consider any bitstring z. Let $z0$ and $z1$ be the bitstrings respectively obtained by appending 0 and 1 to z. Every node in L_z randomly chooses (by choosing its random bitstring) whether to join L_{z0} or L_{z1}, with roughly half of the nodes going to each. If the nodes in L_z who join L_{z0} perfectly interleave the nodes who join L_{z1} then every connection between adjacent nodes in L_{z0} corresponds to a *path of two connections* in L_z. But, due to the randomness, the connections in L_{z0} may not correspond to a path of exactly two connections, it might correspond to just one connection in L_z, or a path of three connections, etc.

This discussion suggests that sending a message using the connections in L_{z0} instead of L_z allows one to make roughly twice as much progress towards the destination. So we'd prefer to send the message using only the high-level lists. Of course, that is not always possible: the source and the destination might belong to different high-level lists, in which case one must use the low-level lists too. So we devise the following rule for routing messages from a node x.

- Send the message through node x's level m list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_m.
- Send the message from e_m through node x's level $m-1$ list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_{m-1}.
- Send the message from e_{m-1} through node x's level $m-2$ list as far as possible *towards* the destination without going *beyond* the destination, arriving at node e_{m-2}.
- ...
- Send the message from e_1 through the level 0 list to the destination (which we can also call e_0).

Consider our previous example. Suppose that node Z wants to send a message to node A. Node Z's level 3 list contains only node Z, so it cannot be used to make progress towards the destination. So $e_3 = Z$. The same is true for its level 2 list. So $e_2 = Z$. Node Z's level 1 list has a connection to node V, which is not beyond the destination, so we send the message to V. Node V can continue sending the message through node Z's level 1 list to node D. So $e_1 = D$. Finally, node D can send the message through the level 0 list to A, which is the destination.

The correctness of this routing rule is guaranteed by the last step: sending the message through the level 0 list always reaches the destination because that list contains all nodes. It remains to analyze how efficient the rule is.