Lecture 26: Small-World Peer-to-Peer Networks and Their Security Issues

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

April 27, 2017
11:42am

Goals:

1. Differences Between Structured P2P and Small-World P2P
2. Freenet as Originally Envisioned by Ian Clarke
3. The Small-World Phenomenon
4. Demonstration of the Small-World Phenomenon by Computer Simulation
5. Decentralized Routing in Small-World Networks
6. Small-World Based Examination of the Original Freenet
7. Sandberg's Decentralized Routing Algorithm for Freenet
8. Security Issues with the Freenet Routing Protocol
9. Gossiping in Small-World Networks

CONTENTS

26.1: DIFFERENCES BETWEEN STRUCTURED P2P AND SMALL-WORLD P2P

- First of all, both structured and small-world P2P networks are most commonly overlaid on top of the internet. So we can refer to them as **structured P2P overlays** and **small-world P2P overlays**.

- As we saw in Lecture 25, structured P2P overlays place topological constraints on what other nodes any given node is aware of for the purpose of data lookup or data retrieval. In a structured P2P overlay, a more-or-less uniformly distributed integer, $nodeID$, is assigned to each node. In the Chord protocol, for example, a node is aware of its immediate successor, which would be a node with the next larger value for $nodeID$. And, through its routing table, a node is also aware of a small number of additional nodes up ahead whose $nodeID$ values sample the node identifier space logarithmically.

- Structured P2P overlays of Lecture 25 are founded on the assumption that any node *can* exchange data with any other node *in the underlying network* (meaning the internet). [Say that A and B are nodes in a structured P2P overlay. Let's say that at a given moment in time, B is not A's immediate neighbor in the P2P overlay and that B does not make any appearance at all in A's routing table. So A is not likely to forward its queries to B at this moment. But, after the addition of a few other nodes or departures thereof, it is entirely possible that B could become A's immediate successor (or predecessor) and/or that B would make an appearance in A's routing table. Should that happen, there would need to be a direct communication link in the underlying internet between A and B.]

- In small-world P2P overlays, on the other hand, it is the human owner of a node who decides which other nodes his/her node will communicate with directly. This feature of small-world P2P overlays could be used by a bunch of people to create their own private overlay network that would be invisible to the rest of the internet. **Such closed overlays are called *darknets*.**

- In this lecture we will assume that it is NOT our intent to create a *closed* private overlay with a small-world P2P. Without requiring approval from all of the current participants, we want a human to be able to have his friends connect with his node — in the same manner that humans form and extend their friendships. In other words, in this lecture, we are interested in **open-ended small-world P2P overlays.**

- Such open-ended small-world P2P networks are also referred to as **unstructured** P2P networks.

- Considering the *ad hoc* nature of the connections in unstructured network overlays, we are interested in studying how messages are routed in such overlays and whether there exist any security problems with a given routing strategy.

- The best example of a small-world (unstructured) P2P overlay today is the Freenet that was proposed initially by Ian Clarke in a dissertation at the University of Edinburgh in 1999. Clarke's main focus was on creating a distributed system for key-indexed storage from where individuals could retrieve information while remaining anonymous. [As mentioned in Lecture 25, the system of web pages is an example of key-indexed storage in which the URLs are the keys and, for each key, the web page at that URL the corresponding value or data.] In other words, Clarke was interested in creating a "decentralized information distribution system" that would provide anonymity to both the providers and the consumers of information. [In Clarke's thinking, the regular internet is a highly centralized information system in which the routing is orchestrated by the DNS that directs an information consumer's query to the web pages of the information providers who stay at fixed locations. According to Clarke, the regular internet makes it all too easy to keep track of the information providers and and the information consumers.]

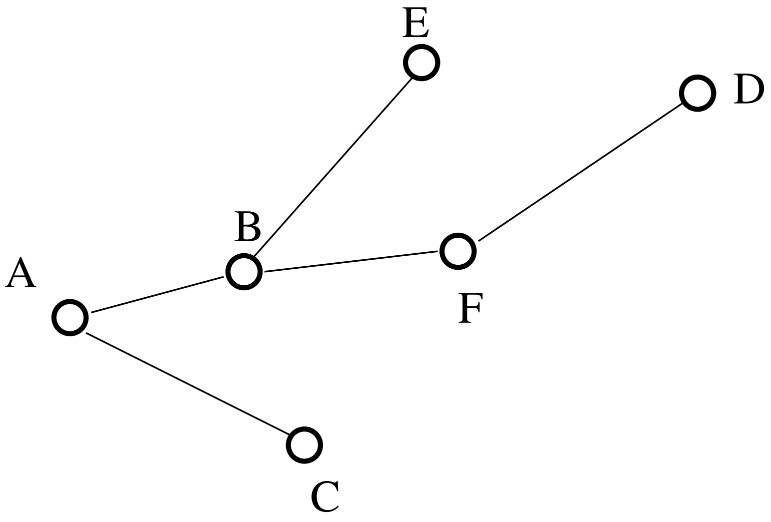
- The next section explains Clarke's original idea for the Freenet in greater detail.

26.2: Freenet AS ORIGINALLY ENVISIONED BY IAN CLARKE

- In the Freenet "protocol" proposed by Clarke, a random key is associated with each data object that we wish to store in a Freenet overlay. The key is assumed to be uniformly distributed over all possible data objects. In a practical implementation, this key would be the hash code of the data object calculated with a mutually agreed upon algorithm. You can think of the key as the data object's address.

- In order to store a data object in the Freenet, your machine issues a `PUT(key, data_object)` message, where `key` is the hash code of the data object. Later we will see how this message propagates in the network to the node where the data object is stored.

- Consider the Freenet overlay of Figure 1. Such an overlay would come into existence on a pairwise trust basis. A and B are each other's neighbors because they trust each other. And the same goes for all the other direct links in Figure 1. The result is a web of trust in which trust can exist between two not-directly-connected nodes because there exists a path of trust between them.



A Freenet Overlay Network with Six Nodes

Each link was established by a human. Messages can only
travel along the links shown. The topology of the network
can only change by new nodes joining and/or old nodes leaving.

Figure 1: *The labels A through F designate the nodes in a Freenet overlay network. Pairs of nodes are connected on the basis of mutual trust between their human owners.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Let's say that node A has the key-value pair `<key,data_object>` in its data store. Let's further say that some other node, D, somehow finds out about this data object (without knowing where exactly this data object resides in the overlay) and would like to download it. D issues the request `GET(key)` for this data object. This request goes to the nodes that are D's neighbors in the Freenet. In our case, that is node F. Since F is not able to find the key `key` in its data store, it forwards the request to its neighbors (not including the node where the request originated). In this manner the `GET` request will reach node A. A copy of the data object is sent from A to D. An important element of the "protocol" is that the data object is cached at all the nodes that are en route between A and D — that means at the nodes B and F. This fact results in replicated storage of data objects.

- Each Freenet node allocates a specific amount of memory for the data store at its location. This implies that, eventually, as the store fills up, there will be a need to delete some of the objects stored in the memory. **The least recently accessed data objects are the first to be deleted when memory runs short.** This implies that any specific data object would eventually be deleted at a node as the node accumulates more and more objects, unless, of course, the data object has migrated to some other node, in which case its survival would be subject to the memory constraints at that node.

- The above-mentioned deletion of the least recently accessed data objects is implemented with the help of a stack data structure. As each new data object is received, the key and a reference to the immediate neighbor from where the data object was received are pushed into the stack. As the stack reaches its storage limit, the data objects corresponding to the keys that fall off the other end of the stack are deleted from memory. If a query for a data object whose key is already in the stack is seen again, its key is again pushed into the stack and the key removed from where it resided in the stack previously. [Clarke's report actually mentions storing a triple in the stack for each new object — the key, the reference to the node from where query was received, and the data object itself. But it seems to me that a more efficient implementation would store just the keys and the neighbor references in the stack for the purpose of deciding which data objects to delete and have a separate key-sorted store for the data objects for the purpose of caching and retrieval.]

- A new data object is inserted into the network when a node issues a `PUT(key, data_object)` message. Such a message is accompanied with a TTL (Time to Live) integer, with the integer decremented by one for each pass through an en route node. If the TTL associated with a `PUT` message received at a node is greater than 0, **the node caches the object** and at the same time forwards the `PUT` message to one of its immediate neighbors. (This is the second mechanism that results in the replicated storage of a data object.) A greedy algorithm driven by the difference between the hash key associated with the data object and the location keys associated with the nodes decides which neighbor the `PUT` message is forwarded to. **We will have more to say about the nature of location keys in Freenet very**

shortly.

- The TTL value we mentioned above is meant to prevent a data insert query from making endless rounds in a Freenet overlay. For the same reason, a data retrieval query also has associated with it a TTL integer value.

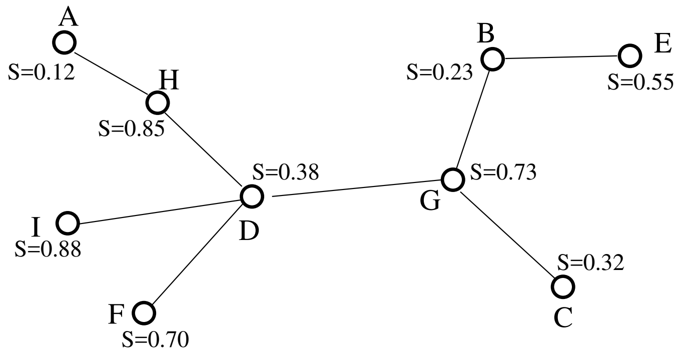
- Each node is assigned an **identifier**, which can be its IP address, and a unique **location key** that is a **randomly chosen number between 0 and 1.** The key values in the range 0 and 1 are to be thought of as being real numbers arranged on a circle, with 0 and 1 being the same number. In other words, the key values are cyclic over the range from 0 and 1 and any arithmetic on the key values is carried out modulo 1. [This is analogous to how the keys are envisioned in a distributed hash table based on, say, the Chord protocol (see Lecture 25).]

- The `GET` and `PUT` messages propagate in the network on the basis of the difference between the location key and the key associated with the data object — subject to a bounded depth-first search for the best destination node. Earlier we mentioned that as a `GET` or a `PUT` message courses its way through the network, at each node its TTL is decremented and it is forwarded to that node for which the difference between location key and the object key is the smallest. Since the search path extended in this manner may lead to a dead-end, the messages are allowed to backtrack in a depth-first manner in order to find alternative paths. The original TTL would obviously control the depth of the search for the destination node.

- How exactly TTL controls the search for the best node has to be understood with care because it is possible for a node to reset the TTL value to what it was originally in order to extend a path.

- The interplay between the TTL values and the bounded depth-first search will be illustrated with the help of the Freenet overlay shown in Figure 2 where the s values are the location keys at each of the nodes. Note that as a request wends its way through the network, it takes along with it the list of nodes already visited.

- Let's first consider a `GET` request issued at node F for a data object whose hash key is 0.10 and let's assume that the TTL value associated with this request is 2. Since F is only allowed to talk to D, D will receive the request with a TTL of 1. D will examine its data store and, not finding the object there, will forward the request to that neighbor whose location key is closest to the data key; in this case, D will forward the request to G with a TTL of 0. When G does not find the data object in its store, G will check the location keys at all its neighbors (not including the one from which the request was received) before responding negatively to the `GET` request. G will discover that B's location key is closer to the requested data key of 0.10 than its own location key. So it will broadcast the `GET` request to all its immediate neighbors (excluding the neighbor from which the request was received) after resetting its TTL to its original value of 2. The search will continue in this manner until TTL is zero and the location keys at all the neighbors are further away from the data key than the node that is the current holder of the request. At that point, the current node will either respond with the data object if its exists in its store or will report nonexistence of the data object.



A Freenet Overlay Network with Nine Nodes

Each link was established by a human. Messages can only travel along the links shown. The topology of the network can only change by new nodes joining and/or old nodes leaving.

Figure 2: *The s values shown are the location keys at the nodes labeled A through H in a Freenet overlay.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- But note a fundamental problem with the bounded depth search for a data object as explained above. The data object of key 0.10 could actually be in the data store at node A but the search path may fail to reach that node. **This points to a fundamental**

shortcoming of the Freenet overlays: In an arbitrarily extended overlay, there is no theoretical guarantee that a data object will be found or that a data object will be stored at its globally best node. It is for this reason that a Freenet overlay works best in small networks created by friends who trust one another and when every node is directly connected with every other node.

- The same logic as presented above applies to `PUT` requests.

- Earlier we mentioned that as a data object is either looked up with a `GET` message or inserted into the network with a `PUT` message, it is cached at all the en route nodes. This caching serves the same purpose as data replication in a structured P2P network such as Chord or Pastry. (See Lecture 25 for Chord and Pastry.)

- To repeat what is probably the most significant difference between the structured and the unstructured overlays, whereas the logic used for deciding where to store a data object in a Freenet is essentially the same as in a structured P2P overlay based on, say, the Chord protocol, any two nodes in a Freenet overlay are directly connected only if the human operators who own the nodes trust each other.

- Shown below is an email from Ian Clarke saying that this conclusion of mine as stated above is not correct. Since he has addressed so succinctly the issue of the scalability of the Freenet as envisioned by him, I have reproduced his email here. This was done with his permission.

Date: Mon, 22 Sep 2008 12:08:52 -0500
From: "Ian Clarke"
To: "Avi Kak"
Subject: Re: Freenet makes its way into education

Thanks Avi - that is great, although I think I may have discovered an important misunderstanding/omission :-(

The bold bullet-point on page 13 says:

"any two nodes in a Freenet overlay are directly connected only if the human operators who own the nodes trust each other."

This is only true of earlier pre-release versions of Freenet 0.7 (versions released in 2006 and 2007), and is not true of the original Freenet design, nor of versions of Freenet released towards the end of 2007 and in 2008. In recent versions of Freenet, this is only true if a user's Freenet node is in "darknet mode".

The Freenet design described in my original dissertation[1] allowed Freenet to create and remove connections between peers, users were not responsible for creating these connections manually. In fact, this process was central to allowing Freenet to scale. See the second paragraph of section 5.1 of my original dissertation - "When the information is found, it is passed back through the nodes that originally forward on the message, and each is updated with the knowledge of where the information was stored". Also note section 7.1.3.3 where I show that retrieval path lengths remain short even as the size of the network is increased from 500 to 900 nodes. This is possible only because of this rewiring process (since in those simulations, the original network configuration was random).

The basic idea is that when a node initiates *or* routes a request for data, and the data is found, that node establishes a new connection to the node were the data was found (this may occur with a probability less than 1.0). This means that when data is found, every node that participated in the retrieval of that data will (with a certain probability) establish a connection to the node which had the data. Since nodes have a limited number of connections, this may require that they drop their least recently used connection to make room for the new one.

We later discovered that this very simple "rewiring" algorithm caused the network to converge to a near-perfect Kleinberg network. This is the key to how the original Freenet was able to scale. In section 2.1.3.3 of my original dissertation[1] you can see that it was successful in retrieving data with a path-length of around 10 in networks up to 900 nodes.

Oskar does a much deeper study of destination sampling in Part I of his thesis[2], and I recommend reading that over my original paper for a robust explanation of this. Our conjecture is that destination sampling is as effective a means to create a small world network as other more complicated approaches such ,as Chord.

In late 2007 we re-introduced destination sampling, albeit in a simpler form than in the original Freenet proposal, we called this "opennet", in contrast to "darknet", where users can only connect to their friends.

I hope this is helpful, please don't hesitate to let me know if you have any questions, or if I can be of any further assistance.

Kind regards,

Ian.

[1] http://freenetproject.org/papers/ddisrs.pdf

[2] http://www.math.chalmers.se/~ossa/lic.pdf

26.3: THE SMALL-WORLD PHENOMENON

- At roughly the same time when Ian Clarke was putting together his ideas on Freenet, Duncan Watts and Steven Strogatz published a computer simulation of the small world phenomenon. This simulation study and subsequent work by others related to routing in small-world networks have played important roles in the recent evolution of the Freenet. The rest of this section is devoted exclusively to the small world phenomenon. We will come back to the computer simulation experiments by Watts and Strogatz in the next section.

- The small world phenomenon, demonstrated experimentally by the famous psychologist Stanley Milgram in 1967, says that **most humans are connected by chains of friendships that have roughly six individuals in them.** [Milgram arrived at this conclusion by asking people in American heartland cities like Wichita and Omaha to send letters to specifically named east-coast individuals (they were referred to as “targets”) who were not known to the senders. A condition placed on each sender was that he/she could only mail the letter to someone with whom the sender was on a first-name basis. It was expected that of all the friends the sender knew, he/she would send the letter to a friend who was most likely to send/forward the letter to its ultimate destination. The same condition was placed on each recipient of the letter — he/she could only mail the letter to a friend with whom he/she was on a first-name basis and who appeared to be most likely to route the letter to its final destination. Obviously, a recipient on a first-name basis with the target would send the letter directly to its final destination. When Milgram examined the mail chains that were completed successfully in this manner, he discovered that, on the average, each letter took six steps between the original sender and the target.]

- To be sure, the notion of the world being small (in the sense that any two human beings are connected through small chains of friendship) was in our collective consciousness even before Milgram did his famous experiments. Since time immemorial, when people have met their friends at the unlikeliest of places (say you live in a small town in the US and you bump into a US friend at a train station in Japan), people have often exclaimed "It's a small world." People have often said the same thing upon being introduced to a stranger at a party and discovering that they have several friends in common with this new person. [Also note that what some consider to be the world's best-loved song "It's a small world (after all)" was written by Sherman brothers for Walt Disney Studios in 1964, three years before Milgram's experiments. Some people might say that the song is less about people being connected through short chains of friendship and more about all people being the same despite superficial differences. Nonetheless, the song's title was probably inspired by the perceived smallness of the world in all aspects of life — including who knows whom through what connection.]

- It would probably be fair to say that, in the best tradition of psychologists, Milgram carried out his experiments to test what many people seemed to believe intuitively.

- Although Milgram's experiments created a lot of excitement in the popular culture and made "six degrees of separation" a part of our lexicon, it is important to bear in mind that only a very small number of the chains started in Milgram's experiments were completed. (In fact, in his first experiment, only 384 out of 24,163 chains were completed.) Non-completion of the chains does not mean that the small world phenomenon does not exist. [It is easy to understand why most chains would not be completed. To many people, receiving a letter that they would need to forward to a friend must have seemed like a chain letter. (Most people react to chain letters by simply ignoring them.) And if people did not think it was a chain letter and actually appreciated the seriousness of the experiment, they might still have considered it to be too much of a bother to mail that letter again.]

- As to whether the small-world phenomenon as uncovered by Milgram's experiments is a true reflection of the social networks in the real world depends on what you think of the "mechanisms" underlying the grouping of people in a mail-forwarding chain. There is obviously some "self-selection" bias in choosing the individual for the next mail forwarding step, in the sense that you are more likely to select someone who has the time and the attitude to engage in the experiment than someone who is your friend but would be reluctant to cooperate.

- Do the above two statements mean that the small world phenomenon is more a myth than a reality? Not at all. Despite the problems with the experiments carried out by Milgram, the mail forwarding chains that were completed are believed to be more a reflection of reality than the chains that were never completed.

26.4: DEMONSTRATION OF THE SMALL-WORLD PHENOMENON BY COMPUTER SIMULATION

- In 1998, Duncan Watts and Steven Strogatz published a "small world" computer simulation study in the journal "Nature" that attracted the attention of researchers in many different areas of "hard sciences."

- The intent behind the Watts and Strogatz computer-simulation study was to see what sort of an interconnection model would capture the small-world phenomenon uncovered by Milgram. Recall from the previous section that the small-world phenomenon according to Milgram meant that any two individuals are connected by a small chain of friends. On the average, the number of friends in the chain is around 6. [The exact length of the chain, even in the average sense, is not an issue. The important point is that this number is small.]

- The interconnection model used by Watts and Strogatz and their simulation experiments do capture the fact that any two nodes in a network are connected, on the average, by a small chain in a manner that is independent of the size of the network. What is interesting is that, in addition to measuring the average length of the shortest chain connecting any pair of nodes, Watts and Strogatz also measured the **local clustering** as would be perceived by any node locally.

- By **local clustering** in a human context, we mean the extent to which an individual's friends are each other's friends.

- If the interconnection model used by Watts and Strogatz reflects how humans relate to one another, that implies that, for the most part, we think of ourselves as existing in small communities — each individual exists in his/her own small world. But, with a small probability, someone in one community will know someone else in a different community. Even though these inter-community links (called *long-range contacts*) are rare so as to be largely imperceptible to most of us individually, the overall effect is the Milgram phenomenon. That is, the shortest path between any two individuals — including individuals living in different communities — never has more than a few other individuals in it.

- Watts and Strogatz carried out their simulations on a ring lattice in which all the nodes in a network are assumed to be arranged in the form of a ring as shown in Figure 3. We will assume that the total number of nodes in a network is N. Initially, each node is provided with k local contacts. For example, in the network shown in Figure 3, we have $k = 4$. That is, we assume that each individual exists in a world with only 4 other individuals in it. [Note that n nodes can have a maximum of $n(n-1)/2$ edges that connect them (resulting in a *fully connected* subgraph or a *clique*). In the construction shown in Figure 3, each neighborhood consists of one node along with its four immediate neighbors, two on either side. That is, each neighborhood consists of 5 nodes. Note that 5 nodes along with 10 edges, with no duplicate edges between any pair of nodes, will constitute a clique. In the Watts and Strogatz construction shown in Figure 3, each neighborhood consisting of 5 nodes has only 7 arcs that connect these nodes. While each such local cluster is not fully connected to form a clique, nonetheless it exhibits a high degree of clustering.]

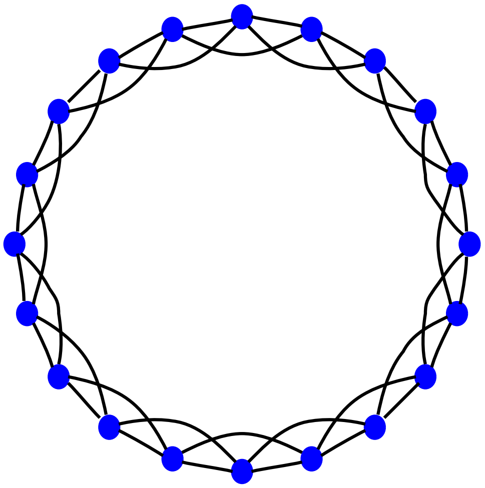


Figure 3: *Every node in this ring lattice has 4 local contacts.*
(*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- A small-world network is created by **rewiring** the basic network diagram, such as the one shown in Figure 3, so that a small number of randomly selected nodes are also connected to more distant nodes.

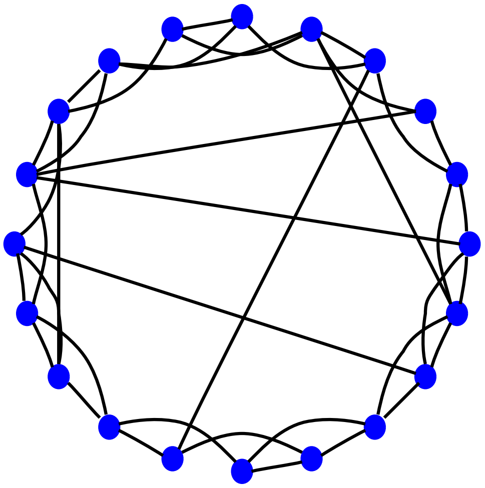


Figure 4: *A rewired version of the ring lattice network of Figure 3 when the probability with which an arc is chosen for rewiring is 0.08.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- To be more specific, when a node is chosen for rewiring, the reewiring at the node consists of redirecting one of the outgoing arcs at the node to some other destination node. The extent of rewiring in the network is controlled by a probability p. This can be accomplished for each rewiring try by calling a random-number generator function, such as $rand()$ in Perl, that returns a random real number that is distributed *uniformly* between 0 and 1 and deciding to rewire an arc if the value returned is less than p; otherwise leaving the arc unchanged. Shown in Figure 4 is the network obtained with $p = 0.08$.

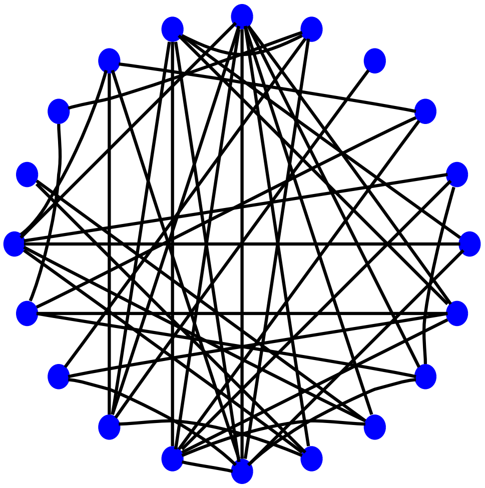


Figure 5: *A rewired version of the ring lattice network of Figure 3 when the probability with which an arc is chosen for rewiring is 1.0.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- As the value of p increases from 0 to 1.0, you will see a progression of network connectivity ranging from what was shown in Figure 3, going through what is shown in Figure 4, and finally ending up in a randomly rewired graph, as shown in Figure 5. The graph we get when $p = 1.0$ is a close approximation to what are known as the Erdos-Renyi random graphs. (These are named after the mathematicians Paul Erdos and Alfred Renyi.)

- Strictly speaking, an Erdos-Renyi graph is obtained by starting with N isolated nodes, visiting each of the $N(N-1)$ possible node pairs, and selecting with probability p a node pair for a direct connection with an edge. [As mentioned earlier, the maximum number of edges in a graph of N nodes is $\begin{pmatrix} N \\ 2 \end{pmatrix} = \frac{N!}{(N-2)!2!} = N(N-1)/2$. In an Erdos-Renyi random graph, each of these possible edges is present with probability p. Additionally, the selection of each edge is independent of all other edges.] Selecting a node pair for a direct connection with probability p can be accomplished by firing up a random number generator as we are considering each node pair. Assuming the random number generator outputs real numbers distributed uniformly between 0 and 1, if the value output is less than p, we draw an edge between the two nodes. Otherwise, we move on and consider the next node pair.

- In an Erdos-Renyi graph, the probability that the degree of a node is d is given by the binomial distribution

$$prob\{degree = d\} \quad = \quad \begin{pmatrix} N-1 \\ d \end{pmatrix} p^d (1-p)^{N-1-d}$$

The average degree of a node in such a graph can be expressed as $z = (N-1)p$. Expressing the probability p in terms of z, we can write for the degree distribution

$$prob\{degree = d\} \quad = \quad \begin{pmatrix} N-1 \\ d \end{pmatrix} \left[\frac{z}{N-1}\right]^d \left[1 - \frac{z}{N-1}\right]^{N-1-d}$$

$$\approx \quad \frac{z^d}{d!} e^{-z}$$

where the last approximation becomes exact as N approaches infinity. That is, as N becomes large, we can expect the probability distribution of the node degrees to become a Poisson distribution. (A consequence of the Poisson law for degree distribution is that we can use the maximum of the Poisson distribution to associate a **scale** with the graph.)

- Unfortunately, the degree distribution in real-life large graphs, such as the graph in which the nodes are the different websites (or the web pages) in the internet and the arcs the URL links between the websites (or the web pages), is not Poisson. It is therefore generally believed that the Erdos-Renyi random graph is not the right model for real-life large networks of nodes. This has given rise to a second method of modeling random graphs — the method of **preferential attachment**. These graphs are also called **scale-free** graphs and **Barabasi-Albert** graphs.

- The degree distribution in Barabasi-Albert graphs exhibits a power law. That is, in a Barabasi-Albert graph, the probability that the degree of a node is d is given by $prob\{degree = d\} = c/d^{\alpha}$ for some positive constants c and α. Typically, $2 \leq \alpha \leq 3$. To fit the Barabasi-Albert model to an actual network, you plot its degree distribution on a log-log plot and then fit the best straight line to the plot. The absolute value of the slope is α. On the other hand, to generate a Barabasi-Albert graph, you start with a connected graph of m_0 nodes (this graph only needs to be connected and not necessarily complete) and, at each time step, you add a new node to the network. The new node is connected with the $m \leq m_0$ other nodes. The probability that the new node is connected to a given existing node i is $\frac{d_i}{\sum_{j=1}^{i-1} d_j}$ where d_j is the degree at node j. This formula makes a node that is already well connected more attractive as a connection destination for a new node. Figure 6 shows an example of a Barabasi-Albert graph that we get after 200 iterations of the algorithm with $m_0 = 10$. The value of m, the number of nodes that the new node is connected at each iteration was set to 1. (Barabasi-Albert random graphs are named after the physcists Albert-Laszio Barabasi and Reka Albert.)

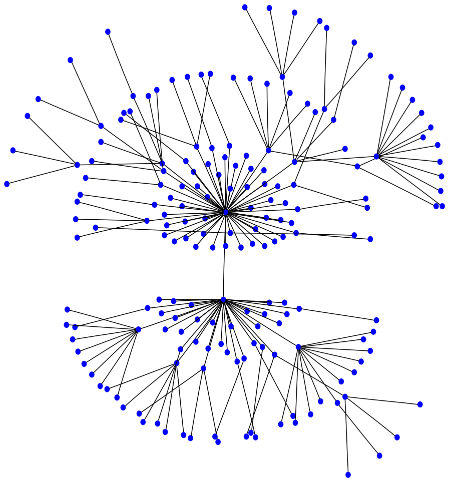


Figure 6: *An example of a Barabasi-Albert random graph. It was generated with 200 iterations of the algorithm and with the parameters $m_0 = 10$ and $m = 1$.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Structured and random graphs that are of interest to us are characterized by two properties: 1) the **diameter** of the graph; and 2) the **clustering coefficient** of the graph.

- By the **diameter** of a graph, we mean the maximum value of the shortest path between any pair of nodes in the graph. The length of a path between any two nodes A and B means the total number of edges on a path that connects A with B. So if there is a direct arc between A and B, the length of the path between A and B through the direct arc is 1. The diameter of a graph is also often taken to be the average value of the shortest path between every pair of nodes in the graph. [Strictly speaking, this definition makes sense only for fully connected graphs. (Note that we only said "fully connected graph" and not a "complete graph." The diameter of a complete graph is always 1.) When a graph consists of multiple connected components or when a graph contains isolated nodes, it is not clear how to compute the diameter either in the sense of it being the maximum value of the shortest distance between every pair of nodes, or in the sense of it being the average value of the shortest distance between every pair of nodes. Most people simply ignore the node pairs that are not connected from the computation of either the maximum or the average.]

- The **clustering coefficient** of a graph measures the average extent to which the immediate neighbors of any node are also each other's immediate neighbors. [Since the clustering coefficient is an average of the "neighbors of a node are also one another's neighbors," it is not clear how to account for isolated nodes in this average. Most people simply ignore the isolated vertices.]

- Both types of random graphs we talked about — the Erdos-Renyi graphs and the Barabasi-Albert graphs — possess **small diameters**. The asymptotic value of the diameter of an Erdos-Renyi random graph is given by $\ln N/\ln(pN)$ where N is the total number of nodes in the graph and p the probability that any pair of nodes is connected by a direct link. The asymptotic value of the diameter of a Barabasi-Albert random graph is given by $\ln N/\ln\ln N$.

- So far we have focused much on the random end of the graphs used by Watts and Strogatz in their computer simulation of the small world phenomenon. As mentioned earlier, the graphs used by Watts and Strogatz consist of random rewirings of the base graph of Figure 3, the extent of rewiring controlled by the probability p. When $p = 1$, we end up with a random graph like an Erdos-Renyi graph.

- We can therefore expect that when $p = 1$ in the Watts and Strogatz computer simulation, we will end up with a small diameter graph. Obviously, $p = 1$ will destroy the local clustering embedded in the ring lattice of Figure 3. So we can expect the clustering coefficient to approach zero as p approaches 1. When $p = 0$, we can obviously expect the graph diameter to become large in direct proportion to the size of the total number of nodes in the graph, but clustering to remain large.

- We will use $L(p)$ and $C(p)$ to denote the diameter and the clustering coefficient, respectively, of a Watts and Strogatz graph. We have already seen that $L(p = 1)$ is proportional to $\ln N$ and $C(p = 1)$ is close to zero. We also know that $L(p = 0)$ is linearly proportional to N and $C(p = 0)$ is close to unity.

- What made Watts and Strogatz paper such a celebrated piece of work was the demonstration by the authors that the diameter $L(p)/L(0)$ falls off rapidly as the value of p is increased even slightly from 0. On the other hand, the clustering coefficient $C(p)/C(0)$ remains pegged at close to unity for these values of p. This is demonstrated by the plots of $L(p)/L(0)$ and $C(p)/C(0)$ shown in Figure 7 for a ring lattice network in which the total number of nodes is set as $N = 200$ and the number of arcs emanating at each node set as $K = 6$. $L(p)/L(0)$ is shown by the solid red plot and $C(p)/C(0)$ by the dashed green plot. Note that the horizontal axis is logarithmic so that we can see more clearly as to what happens to the two ratios when the probability p increases even slightly beyond zero. It is clear that when we introduce just a few long-range contacts by choosing a small non-zero value for p, the network "shrinks" rapidly in terms of its diameter, the local clustering remains substantially the same. **This is the small-world phenomenon in its classic sense.**

- **We refer to a network (or a graph) as a small-world network (or a small-world graph) if it has a small diameter and a large clustering coefficient.**

- In the rest of this section, we will show the Perl script that was used for the two plots presented in Figure 7. The same code can also be used to construct the graphs presented in Figures 3 through 5. If you wish to construct a ring lattice of the sort shown

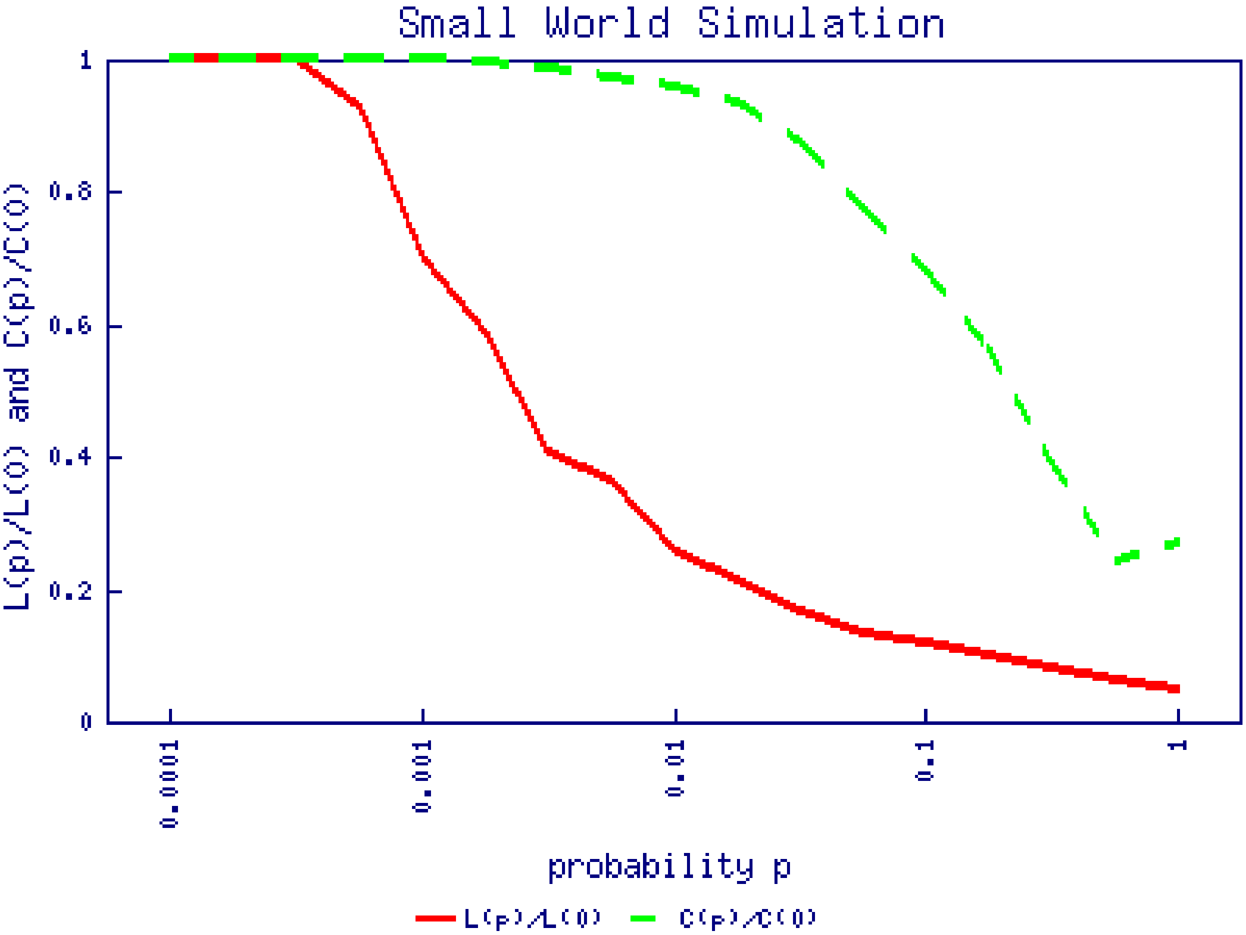


Figure 7: *As the probability p for a long-range contact increases even slightly beyond zero, the diameter of the network shrinks rapidly, as shown by the solid red plot, while the local clustering coefficient remains substantially unchanged, as shown by the dashed green plot.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

in Figures 3 through 5, you'd need to comment out the lines (I) through (P) of the script that comes next. Edit the values of $\$N$, $\$K$, and $\$p$ in lines (A), (B), and (C) as necessary to generate the ring-lattice graphs. You can control the diameter and the degree of clustering by changing the value of the probability $\$p$ in line (C). After you have commented out the lines (I) through (P), you can invoke the script by

```
small_world.pl wsfig.dot
```

where `wsfig.dot` is the name of the "DOT" file that is needed by the GraphViz program `neato` to create a postscript of the image of your graph structure. "DOT" is the language used by the GraphViz library for describing graphs as consisting of nodes and arcs, along with their labels, visualization attributes, etc. (See www.graphviz.org for further details.) The contents of `wsfig.dot`, as produced by a call such as above, can be converted into a postscript figure by

```
neato -Gsplines=true -Gsep=3.0 -Tps -Gcenter -Gsize="6,6" wsfig.dot -o wsfig.ps
```

The `splines=true` option is to override the default of neato to use only straight edges between the nodes. With this option, at least some of the edges will be curved. The `sep=3.0` option is an attempt to increase the distance between the edges. You can display the postscript graph by

```
gv wsfig.ps
```

Examples of such graphs were shown in Figures 3 through 5. If you also want to see the pairwise shortest distances between the nodes in the graph, the diameter of the graph, and its clustering coefficient, make the same calls as above but now also include the lines (I) through (N) of the script. But note that that will work only for small graphs, typically when the total number of nodes is less than 20, since otherwise the matrix of pairwise numbers will be too large for a typical display window on your terminal screen.

- If you just want to produce the plots shown in Figure 7, comment out the lines (C) through (N) and make sure that the lines (O) and (P) are uncommented if you happened to have commented them out earlier. Now you can execute the script by just calling

  ```
  small_world.pl
  ```

  Make sure that the two arguments needed by the call in line (O) are as you want them. The first argument sets the number of nodes in the ring-lattice graph and the second argument the number of neighbors connected directly to each node. The call shown above outputs a ".gif" file called `SmallWorld.gif` for the plots using the Perl module `GD::Graph`. This is done in the function `plot_diameters_and_clustering_coefficients()` whose implementation begins in line (Q).

- Shown below is the script:

```
#!/usr/bin/perl -w

# small_world.pl

# by Avi Kak  (kak@purdue.edu)

# updated October 23, 2008

# Generate a Watts-Strogatz small world network consisting
# of $N nodes.  Each node is directly connected to $K
# immediate neighbors that are located symmetrically in
# the ring lattice on two sides of the node.

# We will model the network as a hash.  The keys in the
# hash are the node indices.  So if there are a total
# of N nodes in the network, the hash will consist of N
# <key,value> pairs. The value for each key is again a
# hash.  The keys for this inner hash at a given node in
# the network are the indices of the destination nodes at
# the outgoing arcs.  So if we focus on node $i in the network,
# and if 'base_graph' is the main hash representing the
# network, $base_graph{i} would stand for a hash consisting of
# the <key,value> pairs such that the keys are the destination
# nodes that the node $i is directly connected with and
# values would be set to 1 for all such destintation nodes.
# For all other network nodes that are not the destination
# nodes for the outgoing arcs emanating from $i,
# $base_graph{i}{j} would be left undefined.  It is obviously
# the case that if $base_graph{i}{j} is set to 1, then
# $base_graph{j}{i} must also be set to 1.  And if we delete an
# arc at node i by undefing $basic_graph{i}{j}, then that arc
# is not completely deleted until we also undef $base_graph{j}{i}.
# It is interesting to observe that each arc from node $i to
# node $j gets a double representation, once in the hash
# $base_graph{i} and then again in the hash $base_graph{j}.

# Of the various functions that are shown below, the functions
# make_base_graph() and rewire_base_graph() are based on the
# code in Mary Lynn Reed's article "Simulating Small-World
# Networks" that appeared in Dr. Dobb's Portal in April 2004.
# The functions shortest_paths() and display_shortest_distance()
# are based on the article "Empirical Study of Graph Properties
# with Particular Interest towards Random Graphs" by Lee Weinstein.


use strict;

my $out_dot_file = shift;

my $N = 20;                                                        #(A)
my $K = 4;                                                         #(B)
my $p = 0.08;                                                     #(C)

my $seed = time();                                                #(D)
srand($seed);                                                     #(E)

my %base_graph = make_base_graph( $N, $K );                       #(F)

my %rewired_graph = rewire_base_graph( $p, %base_graph );         #(G)

display_graph_on_ring_lattice( %rewired_graph );                  #(H)

my %floyd_warshall_matrix = shortest_paths( %rewired_graph );     #(I)

display_shortest_distances( %floyd_warshall_matrix );             #(J)

my $dia = diameter( %floyd_warshall_matrix );                     #(K)

printf "Diameter of the graph is %.3f\n", $dia;                   #(L)

my $cluster_coeff = clustering_coefficient( %rewired_graph );     #(M)

printf "Average cluster coefficient is %.3f \n", $cluster_coeff;   #(N)

# The first arg below is the total number of nodes in the
# graph and the second arg the total number of neighbors.
# Choose an even number for the second arg so that the
# immediate neighbors of a node will be symmetrically placed
# on the two sides of the node.
my $plot_data = diameters_and_clustering_for_different_p(100, 4);  #(O)

plot_diameters_and_clustering_coefficients( $plot_data );         #(P)

#########################################################################
#                          Subroutines
#########################################################################

# This subroutine uses the GD::Graph package to construct
# the plots shown in Figure 7.  The plot output is
# deposited in a file called 'SmallWorld.gif'.  The subroutine
# needs for its input a reference to an array that in turn
# contains references to the following three arrays: 1) an array
# of labels to use for the x-axis; 2) an array of the graph
# diameters for different values of probability; and 3) an
# array of clustering coefficients for different value of
# probability.
sub plot_diameters_and_clustering_coefficients {                  #(Q)
    my $plot_data = shift;

    use GD::Graph::lines;
    use GD::Graph::Data;
    my $sw_graph = new GD::Graph::lines();
    $sw_graph->set(
     x_label => 'probability p',
     y_label => 'L(p)/L(0) and C(p)/C(0)',
     title => 'Small World Simulation',
     y_max_value => 1.0,
     y_min_value => 0,
     y_tick_number => 5,
     y_label_skip => 1,
        x_labels_vertical => 1,
        x_label_skip => 4,
        x_label_position => 1/2,
     line_types => [ 1, 2 ],
     line_type_scale => 8,
     line_width => 3,
    ) or warn $sw_graph->error;

    $sw_graph->set_legend( 'L(p)/L(0)', 'C(p)/C(0)' );
    $sw_graph->plot($plot_data) or die $sw_graph->error;
    my $ext = $sw_graph->export_format;
    open( OUTPLOT , ">SmallWorld.$ext") or
     die "Cannot open SmallWorld.$ext for write: $!";
    binmode OUTPLOT;
    print OUTPLOT $sw_graph->gd->$ext();
    close OUTPLOT;
}


# This subroutine calculates the diameter of a graph of nodes
# and its clustering coefficient for different values of
# the probability p:
sub diameters_and_clustering_for_different_p {                    #(R)
    my $N = shift;        # The total number of nodes in graph
    my $K = shift;        # The immediate neighbors of each node
    my %base_graph = make_base_graph( $N, $K );

    # Figure out the values of the probability $p for which
    # you want to compute the diameter and the clustering
    # coefficient.  To demonstrate the small-world phenomenon,
    # you need a logarithmic scale for p.  We will choose
    # values for p that span the range 0.0001 and 1.0 in such
    # a way that the tick marks on the horizontal axis are
    # equispaced.  Note that on a logarithmic scale, the middle
    # point between two given points is the geometric mean of
    # the two.
    my $x = 1.0 / sqrt(10);
    my $y = $x * sqrt( $x );
    my $z = sqrt( $x );
    my @p_array = (0.0001, $y * 0.001, $x * 0.001, $z * 0.001, 0.001,
                       $y * 0.01,  $x * 0.01,  $z * 0.01,  0.01,
                          $y * 0.1,   $x * 0.1,   $z * 0.1,   0.1,
                          $y * 1.0,   $x * 1.0,   $z * 1.0,   1.0);
    my $dia_no_rewire;
    my $clustering_no_rewire;
    my @dia_array;
    my @clustering_coeffs;
    my @x_axis_tick_labels;
    foreach my $p (@p_array) {
        my %rewired_graph = rewire_base_graph( $p, %base_graph );
        my %floyd_warshall_matrix = shortest_paths( %rewired_graph );
        my $dia = diameter( %floyd_warshall_matrix );
        $dia_no_rewire = $dia if $p == $p_array[0];
        my $dia_ratio = $dia / $dia_no_rewire;
        my $cluster_coeff = clustering_coefficient( %rewired_graph );
        $clustering_no_rewire = $cluster_coeff if $p == $p_array[0];
        my $clustering_ratio = $cluster_coeff / $clustering_no_rewire;
        printf "For p=%.5f,  L(p)/L(0) = %.2f  C(p)/C(0) = %.2f \n",
                         $p, $dia_ratio, $clustering_ratio;
        push @dia_array, $dia_ratio;
        push @clustering_coeffs, $clustering_ratio;
        if ( ($p == 0.0001) || ($p == 0.001) || ($p == 0.01)
             || ($p == 0.1)  || ($p == 1.0) ) {
            push @x_axis_tick_labels, $p;
        } else {
            push @x_axis_tick_labels, undef;
        }
    }
    return [ \@x_axis_tick_labels, \@dia_array, \@clustering_coeffs ];
}

# Create the base graph consisting of nodes and arcs.  As explained in
# the top-level comments, a graph is represented by hash of a hash.
# The keys in the outer hash are the node indices, and the values
# anonymous hashes whose keys are destination nodes connected to a
# given node in the graph and whose values are 1 for those destination
# nodes:
sub make_base_graph {                                              #(S)
    my $N = shift;     # total number of nodes in the graph
    my $k = shift;     # neighbors directly connected on lattice
    my %graph;
    foreach my $i (0..$N-1) {
        my $left = int($K / 2);     # Number of nodes to connect to the left
        my $right = $K - $left;     # Number of nodes to connect to the right
        foreach my $j (1..$left) {
            my $ln = ($i - $j) % $N;
            $graph{$i}{$ln} = 1;
            $graph{$ln}{$i} = 1;
        }
        foreach my $j (1..$right) {
            my $rn = ($i + $j) % $N;
            $graph{$i}{$rn} = 1;
            $graph{$rn}{$i} = 1;
        }
    }
    return %graph;
}

# Rewire each edge with probability $p
sub rewire_base_graph {                                            #(T)
    my $p = shift;         # probability for rewiring a link
    my %graph = @_;
    my $N = keys %graph;   # total number of nodes in the graph

    foreach my $i (keys %graph) {
        foreach my $j (keys %{$graph{$i}}) {
            my $r = rand();
            if ($r < $p) {
                # randomly select a new node $jnew to connect to $i
                my $done = 0;
                my $jnew;
                while (!$done) {
                    $jnew = int($N * rand());
                    if ( ($jnew != $i)  && ($jnew != $j) ) {
                        $done = 1;
                    }
                }
                # remove edge $i  <--> $j
                undef $graph{$i}{$j};
                undef $graph{$j}{$i};
                # add edge $i <-->  $jnew
                $graph{$i}{$jnew}++;
                $graph{$jnew}{$i}++;
            }
        }
    }
    return %graph;
}

# This is the function that is called to display a ring lattice.
# It dumps its output into a DOT file that can then be visually
# displayed as a ring lattice of nodes by the neato program.
sub display_graph_on_ring_lattice {                                #(U)
    die "No output DOT file specified" if !defined( $out_dot_file );
    my %argGraph = @_;
    my $NumNodes = keys %argGraph;    # number of nodes in the graph
    my %graph;
    foreach my $i (0..$NumNodes-1) {
        foreach my $j (0..$NumNodes-1) {
            $graph{$i}{$j} = $argGraph{$i}{$j};
        }
    }
    use constant PI => 3.14159;

    open OUT, "> $out_dot_file";
    print OUT "graph WS { \n";
    print OUT "node [shape=point,color=blue,width=.1,height=.1];\n";

    foreach my $i (keys %graph) {
        my $delta_theta = 2 * PI / $NumNodes;
        my $posx = sin( $i * $delta_theta );
        my $posy = cos( $i * $delta_theta );
        print OUT "$i [pos = \"$posx,$posy!\" ];";
    }
    print OUT "\n";

    # This is my attempt to decrease the "strength" associated
    # with each edge in order to make it more pliable for curving
    # by the spline option set in the command line:
    print OUT "edge [weight=0.001];\n";

    foreach my $i (keys %graph) {
        foreach my $j (keys %{$graph{$i}}) {
            print OUT "$i -- $j;\n" if $graph{$i}{$j};
            undef $graph{$j}{$i};
        #       print OUT "$i -- $j\n";
        }
    }
    print OUT "}\n";
    close OUT;
}


# This is an implementation of the Floyd-Warshall All Pairs Shortest
# Path algorithm.  Note that this is not the most efficient way to
# compute pairwise shortest distances in a graph; however, it is easy to
# program.  The time complexity of this algorithm is O(N^3) where N is
# the number of nodes in the graph.  A faster version of this
# algorithm is Seidel's All Pairs Shortest Distance Algorithm.
sub shortest_paths {                                                #(V)
    my %argGraph = @_;                  # Copy argument graph into %g:
    my $N = keys %argGraph;             # Number of nodes in graph
    my %g;
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            $g{$i}{$j} = $argGraph{$i}{$j};
        }
    }
    my %tempg;
    foreach my $p (0..$N-1) {
        foreach my $q (0..$N-1) {
            $g{$p}{$q} = 0 if $p == $q;
            $g{$p}{$q} = 1000000 if !defined( $g{$p}{$q} );
        }
    }
    foreach my $t (0..$N-1) {
        foreach my $i (0..$N-1) {
            foreach my $j (0..$N-1) {
                $tempg{$i}{$j} = $g{$i}{$j} < $g{$i}{$t} + $g{$t}{$j} ?
                                     $g{$i}{$j} : $g{$i}{$t} + $g{$t}{$j};
            }
        }
        %g = %tempg;
    }
    # Undefine the edges that were not there to begin with:
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            undef $g{$i}{$j} if $g{$i}{$j} >= 1000000;
        }
    }
    return %g;
}

# Compute the diameter as the average of all pairwise shortest
# path-lengths in the graph:
sub diameter {                                                     #(W)
    my %graph = @_;
    my $N = keys %graph;           # Number of nodes in graph
    my $diameter = 0;
    foreach my $p (0..$N-1) {
        foreach my $q ($p..$N-1) {
            $diameter += $graph{$p}{$q} if defined $graph{$p}{$q};
        }
    }
    return $diameter / ($N * ($N - 1) / 2.0 );
}

# Utility routine good for troubleshooting:
sub display_shortest_distances {                                   #(X)
    my %g = @_;               # Copy argument graph into %g:
    my $N = keys %g;          # Number of nodes in graph
    foreach my $p (0..$N-1) {
        foreach my $q (0..$N-1) {
            if ( defined($g{$p}{$q}) ) {
                print "$g{$p}{$q} ";
            }
            else {
                print "  ";
            }
        }
        print "\n";
    }
}
```

```
# Calculates the clustering coefficient of a graph.  See
# the text for what is meant by this coefficient.
sub clustering_coefficient {                                       #(Y)
    my %g = @_;
    my $N = keys %g;
    my @cluster_coeff_arr;
    my %neighborhood;
    # Initialize the neighborhood for each node. Obviously,
    # each node belongs to its own neigborbood:
    foreach my $i (0..$N-1) {
        $neighborhood{$i} = [$i];
    }
    foreach my $i (0..$N-1) {
        foreach my $j (0..$N-1) {
            if (defined($g{$i}{$j}) && ($g{$i}{$j} == 1)) {
                push @{$neighborhood{$i}}, $j;
            }
        }
    }
    # For troubleshooting:
    # foreach my $i (0..$N-1) {
    #     print "@{$neighborhood{$i}}\n";
    # }
    foreach my $i (0..$N-1) {
        my $n = @{$neighborhood{$i}};        # size of neighborhood
        foreach my $j (@{$neighborhood{$i}}) {
            foreach my $k (@{$neighborhood{$i}}) {
                if (defined($g{$j}{$k})) {
                    $cluster_coeff_arr[$i]++ if $g{$j}{$k} == 1;
                }
            }
        }
        # Divide by n(n-1) because every edge in the neighborhood will be
        # counted twice.  Ordinarily, you would divide by n(n-1)/2.
        $cluster_coeff_arr[$i] /= $n * ($n - 1) unless $n == 1;

        # For troubleshooting:
        # print "for $i, the cluster coefficient is $cluster_coeff_arr[$i]\n";
    }
    my $total = 0.0;
    foreach my $i (0..$N-1) {
        $total += $cluster_coeff_arr[$i] if defined $cluster_coeff_arr[$i];
    }
    my $average_cluster_coeff = $total / $N;
    # For troubleshooting:
    # print "the average cluster coefficient is $average_cluster_coeff\n";
    return $average_cluster_coeff;
}
```

26.5: DECENTRALIZED ROUTING IN SMALL-WORLD NETWORKS

- The Milgram experiments and the computer simulations by Watts and Strogatz tell us about the *existence* of the small-world phenomenon, meaning that humans form networks that are characterized by small network diameters and large clustering coefficients.

- But Milgram's experiments additionally demonstrated that humans also possess an innate ability to navigate through such networks. A letter wending its way through individuals who belong to different friendship clusters is an example of this human-driven navigation. Another example would be a human finding his/her way to a far off destination by asking for directions along the way (assuming that the individuals encountered during the journey may not know directly how to get to the final destination from where they are, but do know someone else who might provide further similar help).

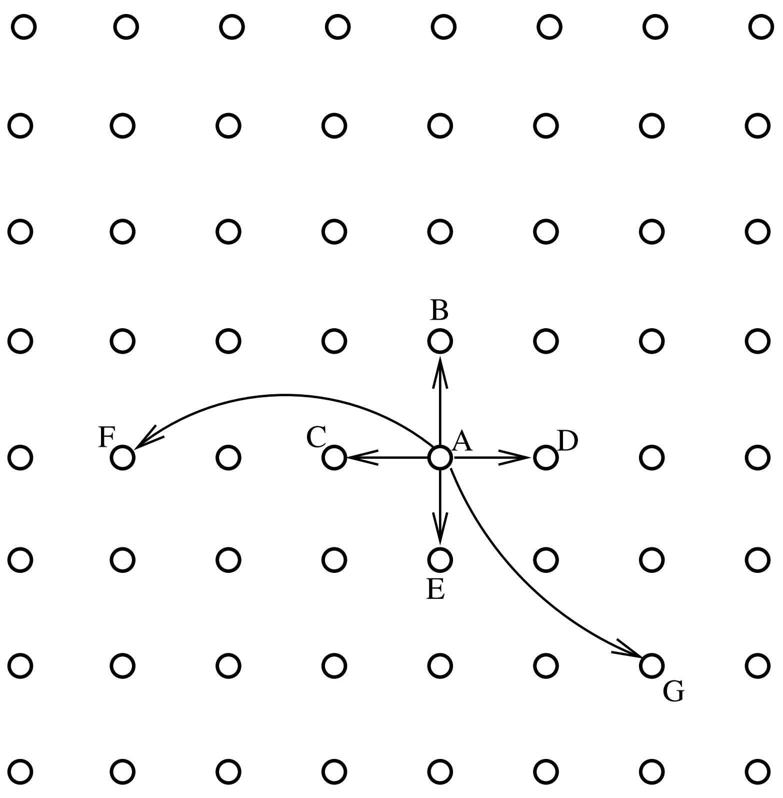
- Jon Kleinberg was the first to make the observation that Milgram's experiments also constituted a discovery of the innate ability of humans to find short paths to their destinations using only local information. In another celebrated paper dealing with the small world phenomenon, Kleinberg then raised the question as to what conditions would have to prevail in a network for there to exist decentralized algorithms that would allow the short paths to be discovered.

- Kleinberg used a two-dimensional lattice of nodes for answering the above question. With (i, j) denoting the coordinates of a node in a grid, Kleinberg used the following L_1 metric to measure the distances in the grid:

$$d(\ (i,j), (k,l)\ ) \quad = \quad |k\ -\ i|\ +\ |l\ -\ j| \tag{1}$$

  This metric is also known as the *city block metric* or the *Manhattan distance.*

- Kleinberg used three integer parameters, pp, qq and rr, to specify the connectivity in a grid lattice of the sort shown in Figure 8. For any given node A in the grid, all its immediate neighbors within the L_1 distance of pp are A's local contacts. That is, node A has outgoing arcs that connect it with all nodes at a distance of up to and including pp from A. (When $pp = 1$, that would only be the four immediate neighbors, to the east and west, and to the north and south. That is the case shown in Figure 8.) In addition, the node A has $qq \geq 0$ long-range contacts. The qq



The connectivity of the nodes is controlled by three integer parameters p, q, and r. For a node such as A, all the other nodes within distance p are its local contacts. In addition, every node such as A has q long−range contacts. The parameter r controls the probability that a node at a certain distance is A's long−range contact. For the grid shown, p = 1 and q = 2.

Figure 8: *Shown is a two-dimensional lattice of nodes in which a node such as A has all its neighbors within some L_1 distance forming its local contacts. A also has qq long-range contacts at distances that are set randomly.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

long-range contacts for a node are selected by firing up a random number generator that spits out qq random numbers X according to the inverse r^{th}-power distribution:

$$prob(X \ = \ x_i) \quad \propto \quad |x_i|^{-rr} \qquad\qquad i \ = \ 1, 2, \ldots, qq \qquad (2)$$

for some prespecified constant rr. Construing each of these random numbers as an L_1 distance from node A, we randomly select a long-range destination node at each such distance. Note that the L_1 metric gives rise to diamond-shaped equidistance contours in the plane of the grid. So, in general, there will be several candidates for the long-range contact at each distance. Of these, we will select one with uniform randomness. Referring to Figure 8, we can say that the probability of a node such as F to be A's long-range contact is given by

$$\frac{|d(A,F)|^{-rr}}{\sum_Y |d(A,Y)|^{-rr}} \qquad (3)$$

where the summation in the denominator is over all qq long-range contacts of node A.

- When $rr = 0$ in the above model, a node's long-range contacts will be selected uniformly from all the nodes in the network. This corresponds to how the long-range contacts are selected in the Watts and Strogatz model.

- The network created by the procedure outlined above can be considered to be a superposition of a structured base graph and a sparse random graph. The nodes, along with each node's local contacts, constitute the base graph and the randomly-selected long-range contacts the superimposed random graph. The base graph exhibits a high clustering coefficient in the sense that the neighbors of a node are highly likely to be each other's neighbors also. The superimposed random graph provides the occasional short-cuts needed for giving a small diameter to the network.

- Using the above network model, Kleinberg has theoretically established that there exists an efficient **decentralized** algorithm for routing a message from any node to any other node but only when $rr = 2$. *The "time" taken by the decentralized algorithm is $O(\log^2 N)$ where N is the total number of nodes in the network.* The decentralized algorithm consists of each en route node making a greedy routing decision based purely on (i) the coordinates of its local and long-range contacts; (ii) the coordinates of the nodes that the message was previously routed through; and (iii) the coordinates of the target node. The message that needs to be delivered carries with it the target node coordinates and the coordinates of the nodes already visited. This result by Kleinberg applies to the empirically interesting case of $pp = 1$ and $qq = 1$. That is, each node is directly connected with its four closest neighbors and has one long-range contact.

- All of the preceding discussion in this section has focused on two dimensional graphs, that is, graphs whose nodes can be located with two indices. Kleinberg has shown the results can be extended to k-dimensional graphs for arbitrary k. That is, one can prove the existence of a decentralized greedy algorithm for efficiently discovering the small-world paths in a time that is a polynomial in $\log N$, where N is the total number of nodes in the network, provided the distribution of long-range contacts follows an inverse k-power distribution. In other words, the probability that a node y is a long-range contact for a node x should be given by

$$prob(x,y) \quad = \quad \frac{|d(x,y)|^{-k}}{\sum_z |d(x,z)|^{-k}} \tag{4}$$

where the summation in the denominator is over all the qq long-range contacts that any node in the network is allowed to have.

- **An Open Question:** Kleinberg's work shows that the connectivity in a network must obey certain specific mathematical conditions so that the short paths typical of small worlds can be discovered at all on the basis of just local reasoning. Kleinberg has also demonstrated the non-existence of decentralized algorithms for finding the short paths when the mathematical conditions on the connectivity are violated. On the other hand, Milgram's work has shown that humans appear to have the ability to find the small-world short paths in their social networks. Does that mean that the human networks implicitly satisfy the mathematical constraints discovered by Kleinberg? As matters stand today, we do not yet know the answer to this fundamental question.

- A problem with modeling computer networks in which humans directly restrict the connections between the machines on the basis of trust with Kleinberg's graphs is that it is not clear how to interpret the node indexing used in the graphs. The most straightforward interpretation would be based on geographical locations of the nodes. But that frequently does not make sense for the case of overlay networks.

26.6: SMALL-WORLD BASED EXAMINATION OF THE ORIGINAL CONCEPTUALIZATION OF FREENET

- This confluence of the Freenet ideas proposed by Clarke and the computer-simulation-based confirmation of the small-world phenomenon by Watts and Strogatz led to the belief that, since the individual connections in a Freenet are based on friendships and trust, the routing patterns in a Freenet would exhibit the small world phenomenon. One thought that, even as the number of nodes in a Freenet grew arbitrarily, there would exist short paths between any pair of nodes.

- But, as made clear by Kleinberg's work summarized in the previous section, the *existence* of small-world short paths in a network is different from the existence of decentralized routing algorithms for the discovery of those short paths.

- The data insertion and data migration notions incorporated in Clarke's original proposal for the Freenet do not provide any mathematical guarantee that a data object present at a given node would be retrievable by *all* other nodes in a Freenet network.

Those notions do not even ensure that a previously inserted data object would survive in the network as the data stores at the various nodes begin to fill up with the data objects currently in demand.

- In summary, Clarke's original ideas may work well in small friend-to-friend networks in which each node is directly connected with all the other nodes. However, such may not be the case in arbitrary P2P networks.

26.7: SANDBERG'S DECENTRALIZED ROUTING ALGORITHM FOR FREENET

- Section 26.2 mentions associating an immutable identifier and a randomly selected unique location key with each node. To briefly review some of the other statements made in that section: The key value is cyclic over the range from 0 to 1 and any arithmetic on the keys is modulo 1. A data object is stored at a node whose location key is closest to the hash key associated with the data object. When a data object is inserted into a Freenet or retrieved from it, it is cached at all the en route nodes. This caching plays the same role in a Freenet as data replication in a structured P2P network.

- To remedy the shortcoming of the Freenet as originally conceptualized (that we only have a guarantee of the *existence* of small-world like short paths between any two nodes but **no guarantee** of the *existence* of a decentralized algorithm for the discovery of those short paths), Oskar Sandberg has proposed a new routing algorithm for the Freenet.

- Sandberg's routing algorithm is based on Kleinberg's theorems on when a network is allowed to possess a decentralized routing algorithm for finding the small-world short paths. We will refer to a network whose connectivity allows for efficient decentralized routing algorithms to exist as a **Kleinberg network**.

- Obviously, the probability distribution associated with the long-range contacts in a Kleinberg network obeys the inverse k-power law for a k-dimensional graph.

- Also recall that a Kleinberg network can be considered to be a superposition of a **base lattice** in which every node is directly connected with a certain number of all its immediate neighbors and a random network that only contains the long-range contacts. The L_1 distance function that drives the greedy algorithm for decentralized routing is defined in the base lattice.

- Sandberg's Freenet routing algorithm is derived by pretending that a Freenet graph corresponds to the long-range contacts in some unknown k-dimensional Kleinberg network. If the underlying base lattice of such a Kleinberg network could be found (note that we already have the graph of the long-range contacts), that would automatically provide us with an L_1 metric for driving a greedy algorithm for the discovery of short paths.

- Sandberg has shown that finding the unknown base grid can be cast as a problem in statistical estimation in which the lattice coordinates of the actual Freenet nodes are the parameters to be estimated. Sandberg used the MCMC (Markov-Chain Monte-Carlo) technique for this estimation.

- To briefly present Sandberg's formulation of the problem, let V represent the nodes in an actual Freenet. Let G represent the underlying base lattice. Let ϕ denote the positions assigned to the nodes of V in the base lattice G. Now let E denote the set of edges in the Freenet network. Since Sandberg assumes that the Freenet edges are the long-term contacts in a k-dimensional base lattice G that corresponds to a Kleinberg network, it must be the case that

$$prob(E|\phi) \quad = \quad \prod_{i=1}^{m} \frac{1}{d\left(\phi(x_i), \phi(y_i)\right)^k H_G} \qquad (5)$$

where x_i and y_i denote the two nodes at the two ends of an edge and where we assume that the Freenet has a total of m edges. H_G is the normalizing constant.

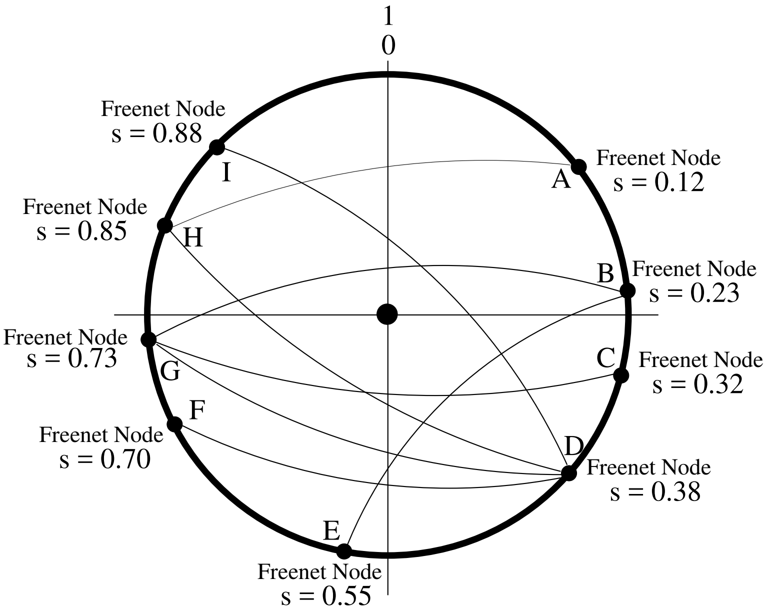
- At least theoretically, the equation shown above could be used to construct a maximum-likelihood estimate for the unknowns ϕ for a given value of the dimensionality k. That is, we would want ϕ that maximizes the likelihood of the observations E. Evidently, $prob(E|\phi)$ would be maximized by finding those assignments of Freenet nodes to base lattice positions that minimize the product of the edge lengths shown in the denominator. There is obviously a combinatorial issue in trying every possible assignment of base lattice to network node mappings to find out the mapping that minimizes the product of the edge lengths. This, as intuition might suggest, turns out to be an NP-complete strategy.

- Using Bayes' rule, Sandberg casts the problem as as exercise in stochastic optimization:

$$prob(\phi|E) \quad = \quad \frac{prob(E|\phi) \cdot prob(\phi)}{prob(E)} \qquad (6)$$

So our goal is to construct a Bayesian estimate for the $\phi : \ G \ \rightarrow \ V$ mapping function that results in the largest value for the posterior distribution on the left hand side above.

- Let's assume that the underlying base lattice is one dimensional. A convenient way to model a 1-D lattice is as a ring lattice, something that becomes intuitive if you assign location keys to the nodes whose values are between 0 and 1. [Recall that earlier in this lecture on page 11, we assigned a location key to each node in a Freenet overlay. Since the value of the location key was between 0 and 1 in that discussion, we can visualize those nodes as being located on a circle. I should also mention that the Identifier Circle of Lecture 25 is the same thing as a ring lattice.] Shown in Figure 9 is a Freenet overlay whose nodes have been assigned the 1-D location keys between 0 and 1. The location keys are shown as the values of the s parameter. Again as previously mentioned on page 11, the location distance between any two nodes is to be measured along the circle modulo 1. The chordal arcs shown in Figure 9 indicate the human-established pairwise connections between the nodes. Information between the nodes can only flow along the chordal arcs. The Freenet overlay shown in Figure 9 on a ring lattice would be more commonly visualized as a regular graph, as shown in Figure 2.



In a Freenet overlay, the connection between each pair of nodes
is established by human operators on the basis of trust and friendship.
The arcs drawn directly between the nodes indicate such communication links.

Figure 9: *A ring-lattice visualization of a Freenet overlay whose nodes have been assigned location keys, s, between 0 and 1.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- In our current context, assigning location keys to Freenet nodes in the manner shown in Figure 9 amounts to embedding the nodes in a 1-D base graph. The assignment of s keys is one possible embedding in an imaginary base graph on the ring lattice. Since the direct connections between the nodes are supposed to correspond to the long-range links in the base graph, for the case of 1-D base graphs, we can now write

$$prob(E|\phi) \quad = \quad \prod_{i=1}^{m} \frac{1}{|\phi(x_i) - \phi(y_i)|_s H_G} \tag{7}$$

where x_i and y_i denote the two nodes at the two ends of the i^{th} direct link in the overlay, where $\phi(x)$ returns location value assigned to node x, and where the distance $|.|_s$ means that distance is to be measured modulo 1 along the unit circle. Our goal is to find that mapping $\phi : V \rightarrow [0, 1)$ that maximizes the posterior probability shown previously as $prob(\phi|E)$ in Equation (6).

- But maximization of the posterior $prob(\phi|E)$ in Equation (6) presents certain computational challenges that are best seen if we write that equation in the following form:

$$prob(\phi|E) \quad = \quad \frac{prob(E|\phi) \cdot prob(\phi)}{\int_\phi prob(E|\phi) \cdot prob(\phi) d\phi} \tag{8}$$

The computational challenge is how to compute the denominator for any candidate $prob(\phi)$. This is where the MCMC technique we describe next comes in handy. MCMC allows us to generate samples of ϕ that conform to a candidate $prob(\phi)$ distribution.

Subsequently, these samples can be used for a Monte Carlo based approach to finding the value of the integral in the denominator.

- An aside on MCMC: [MCMC, for *Markov Chain Monte Carlo*, is a variation on the traditional Monte-Carlo simulations for solving difficult problems in parameter estimation, integration, combinatorial optimization, etc. Let's say you want to estimate the integral $\int_{\mathbf{x}\in\mathcal{X}} f(\mathbf{x})d\mathbf{x}$ where the points $\mathbf{x}$ belong to some high-dimensional space. The Monte-Carlo approach to estimating the integral would be to draw a set of N points $\mathbf{x}_i$ from a uniform distribution over the domain, with the condition that the points are selected independently, and to then form the sum $(1/N)\sum_{i=1}^{N} f(\mathbf{x}_i)$. One can show that this summation is an unbiased estimate of the true integral. This approach extends straightforwardly to the estimation of $\int_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})f(\mathbf{x})d\mathbf{x}$, where $p(\mathbf{x})$ is a probability density function, if $p(\mathbf{x})$ is simple, like a uniform or a Gaussian density, but, as you would expect, the N samples $\mathbf{x}_i$ must now be drawn according to the density $p(\mathbf{x})$. That is, in this case also, an unbiased estimate of the integral $\int_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x})f(\mathbf{x})d\mathbf{x}$ would be given by the summation $(1/N)\sum_{i=1}^{N} f(\mathbf{x}_i)$. Unfortunately, this standard Monte-Carlo approach does not work when $p(\mathbf{x})$ is a complicated probability density function — simply because it is non-trivial to sample complicated density functions algorithmically. This is where MCMC approaches become useful. **MCMC sampling is based on the following intuitions:** For the very first sample, $\mathbf{x}_1$, you accept any value that belongs to the domain of $p(\mathbf{x})$, that is, any randomly chosen value $\mathbf{x}$ where $p(\mathbf{x}) > 0$. At this point, any sample is as good as any other. For the next sample, you again randomly choose a value from the interval where $p(\mathbf{x}) > 0$ but now you must "reconcile" it with what you chose previously for $\mathbf{x}_1$. Let's denote the value you are now looking at as $\mathbf{x}_*$ and refer to it as our *candidate* for $\mathbf{x}_2$. As to having to "reconcile" $\mathbf{x}_*$ with the previously selected $\mathbf{x}_1$ before accepting the candidate as the next sample, here is what I mean: Your desire for obvious reasons should be to select a large number of samples in the vicinity of the peaks in $p(\mathbf{x})$ and, relatively speaking, fewer samples where $p(\mathbf{x})$ is close to 0. You can capture this intuition by examining the ratio $a1 = \frac{p(\mathbf{x}_*)}{p(\mathbf{x}_1)}$. If $a1 > 1$, then accepting $\mathbf{x}_*$ as $\mathbf{x}_2$ makes sense because your decision would be biased toward placing samples where the probabilities $p(\mathbf{x})$ are higher. However, should $a1 < 1$, you need to exercise some caution in accepting $\mathbf{x}_*$ for $\mathbf{x}_2$. While obviously any sample $\mathbf{x}_*$ where $p(\mathbf{x}_*) > 0$ is a legitimate sample, you nonetheless want to accept $\mathbf{x}_*$ as $\mathbf{x}_2$ with some hesitation, your hesitation being greater the smaller the value of $a1$ in relation to unity. You capture this intuition by saying that let's accept $\mathbf{x}_*$ as $\mathbf{x}_2$ with probability $a1$. In an algorithmic implementation of this intuition, you fire up a random-number generator that returns floating-point numbers in the interval $(0, 1)$. Let's say the number returned by the random-number generator is u. You accept $\mathbf{x}_*$ as $\mathbf{x}_2$ if $u < a1$. **It is these intuitions that form the foundation of the original Metropolis algorithm for drawing samples from a specified probability distribution.** Since each sample chosen in this manner depends on just the sample selected previously, a sequence of such samples forms a Markov chain. **For that reason, this approach to drawing samples from a distribution for the purpose of Monte-Carlo integration of complex integrands is commonly referred to as the Markov-Chain Monte-Carlo approach, or, more conveniently, as the MCMC sampler.** To be precise, the Metropolis algorithm for MCMC sampling uses what is known as a *proposal distribution* $q(\mathbf{x}_*|\mathbf{x}_{t-1})$ to return a candidate $\mathbf{x}_*$ for the current sample $\mathbf{x}_t$ given the previous sample $\mathbf{x}_{t-1}$ and requires that $q(.|.)$ be symmetric with respect to its two arguments if you want the theoretical guarantee that the first-order probability distribution of the samples of the Markov Chain converge to the desired density $p(\mathbf{x})$. This restriction on the proposal distribution is removed in the more general Metropolis-Hastings (MH) algorithm, but now the ratio that is tested for the acceptance of the candidate $\mathbf{x}_*$ is given by the product $a = a1 \times a2$ where $a2 = \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_*)}{q(\mathbf{x}_*|\mathbf{x}_{t-1})}$. If $a \geq 1$, we accept the candidate $\mathbf{x}_*$ immediately for the next sample. Otherwise, we only accept it with probability a.]

- The Metropolis-Hastings (MH) algorithm is the most popular algorithm for MCMC sampling. The algorithm is straightforward to implement, as can be seen by the Perl code for the function `metropolis_hastings()` shown next. The goal of the code is to generate an MCMC sequence whose first-order density function approximates $p(\mathbf{x}) = 0.3 \cdot e^{-0.2\mathbf{x}^2} + 0.7 \cdot e^{-0.2(\mathbf{x}-10)^2}$. This density function, calculated by the subroutine `desired_density()` in the Perl script, is shown by the line plot in Figure 10. A histogram for the first 500 MCMC samples produced by the script is shown as a bar graph in the same figure. [Ordinarily, it is best to discard several hundred samples at the beginning of such a sequence to eliminate the effects of initialization. After these initial samples are rejected, the rest of the sequence would follow even more closely the desired density.] As mentioned in the rather long small-font note in the previous bullet, the MH algorithm requires us to specify a **proposal** density function $q(x|y)$. (This is called the proposal density because its primary role is to propose the next sample of a sequence given the current sample.) The proposal density function used in the code is $q(x|y) = \mathcal{N}(y, 100)$, that is, it is a normal density that is centered at the previous sample with a standard deviation of 10. This standard-deviation was chosen keeping in mind the interval $(-5.0, 15.0)$ over which $p(\mathbf{x})$ is defined with values not too close to zero, as shown by the line plot in Figure 10.

- Shown below is a pseudocode description of the algorithm that is programmed in the `metropolis_hastings()` subroutine of the Perl script. In this description, $p(\mathbf{x})$ is the desired density function.

```
initialize x_0

for i=0 to N-1:

    -- draw a sample u from uniform density U(0,1)
       (this u is used only for the test on the variable
        a at the end of the pseudocode.  When a>1, we accept
        the new candidate sample x_* for the next sample.
        However, when a<1, we accept the candidate x_* only
        if a<u)
```

```
-- propose a new candidate sample  x_*  using q(x_*|x_i)

-- a1 = p(x_*) / p(x_i)

-- a2 =  q(x_i|x_*) / q(x_*|x_i)

-- a  = a1 . a2

-- if a >= 1:
        x_{i+1}  =  x_*
   else if u < a:
        x_{i+1}  =  x_*
   else:
        x_{i+1} = x_i
```

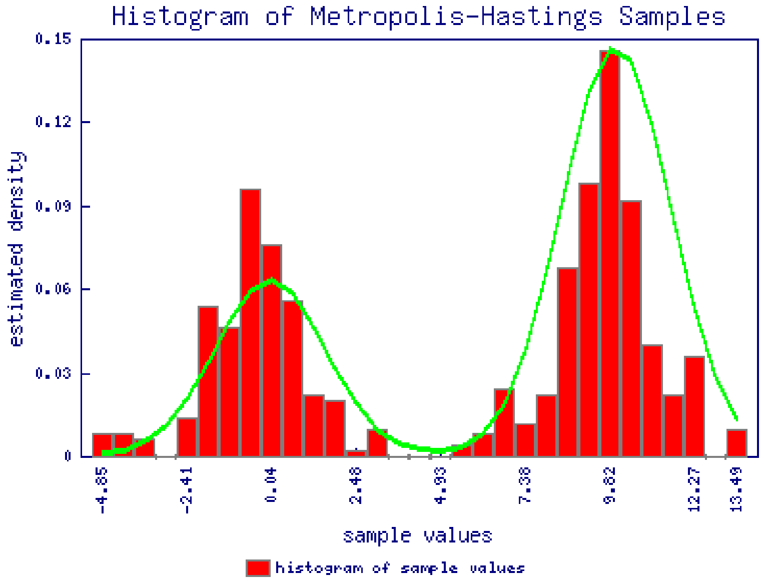


Figure 10: *A histogram of the samples produced by a Perl implementation of the Metropolis-Hastings algorithm.* (*This figure is from Lecture 26 of "Lecture Notes on Computer and Network Security" by Avi Kak*)

- Shown below is the code that the reader can play with to get deeper insights into the Metropolis-Hastings algorithm. As you will notice, as to what extent you will be able to match a given desired density will depend on your choice of the proposal density function $q(x|y)$.

```
#!/usr/bin/perl -w

# Metropolis_Hastings.pl

# by Avi Kak  (kak@purdue.edu)

# The Metropolis-Hastings sampling algorithm is used to generate a
# sequence of samples from a given probability density function.
# Such a sequence of samples can be used in MCMC (Markov-Chain
# Monte-Carlo) simulations.

# The workhorse of the code here is the metropolis_hastings()
# function that takes one argument --- the number of samples
# you want the function to return.  The implementation is based
# on the logic shown in Figure 5 of "An Introduction to MCMC for
# Machine Learning" by Andrieu, De Freitas, Doucet, and Jordan
# that appeared in the journal Machine Learning in 2003.  The
# desired density function used is the same as in their Figure 6.

# The code shown here deposits its histogram in a gif file
# called histogram.gif.  The visual display in the gif file
# shows both the histogram for the samples and the true
# desired density function for the samples.

use strict;
use Math::Random;           # for normal and uniform densities
use constant PI => 4 * atan2( 1, 1 );
use Math::Big qw/euler/;    # euler(x) returns e**x
use Algorithm::MinMax;      # efficiently calculates min and max in an array
use GD::Graph::mixed;       # for bar graphs and line plots

# Useful for creating reproducible results:
random_seed_from_phrase( 'hellojello' );

my @samples = metropolis_hastings(500);

# The following call returns a reference to an array whose
# first element is an anonymous array consisting of the
# horizontal labels for the histogram, whose second element
# an anonymous array consisting of the bin counts, and whose
# third element an anonymous array consisting of the samples of
# the desired density function:
my $histogram = make_histogram( @samples );

plot_histogram( $histogram );

#######################################################################
#                          Subroutines
#######################################################################

# This subroutine uses the GD::Graph Perl module to create a visual
# display that shows both the bar graph for the histogram
# of the MCMC samples created by the metropolis_hastings() function
# and a line plot of the desired density function for such
# samples.  The data fed to this subroutine is output by the
# make_histogram() subroutine.
sub plot_histogram {
    my $plot_data = shift;
    my $histogram_bars = new GD::Graph::mixed();
    $histogram_bars->set(
        types => [ qw( bars lines ) ]
    );
    $histogram_bars->set(
     x_label => ’sample values’,
     y_label => ’estimated density’,
     title => ’Histogram of Metropolis-Hastings Samples’,
     y_tick_number => 5,
     y_label_skip => 1,
        y_max_value => 0.15,
        y_min_value => 0.0,
        x_labels_vertical => 1,
        x_label_skip => 4,
        x_label_position => 1/2,
     line_type_scale => 8,
     line_width => 3,
    ) or warn $histogram_bars->error;

    $histogram_bars->set_legend( ’histogram of sample values’ );
    $histogram_bars->plot($plot_data) or die $histogram_bars->error;
    my $ext = $histogram_bars->export_format;
    open( OUTPLOT , ">histogram.$ext") or
     die "Cannot open histogram.$ext for write: $!";
    binmode OUTPLOT;
    print OUTPLOT $histogram_bars->gd->$ext();
    close OUTPLOT;
}

# This subroutine constructs a histogram from the MCMC samples
# constructed by the metropolis_hastings() subroutine.
# This subroutine also constructs an array from the desired
# density function whose calculation is encapsulated in the
# desired_density.pl subroutine.  Yet another array synthesized
# in this subroutine consists of the labels to use for the
# visual display constructed by the plot_histogram() subroutine.
sub make_histogram {
    my @data = @_;
    my $N = @data;
    my ($min, $max) = Algorithm::MinMax->minmax( \@data );
    my $num_bins = 30;
    my @hist = (0.0) x $num_bins;
    my @desired_density;
    my $bin_width = ($max - $min) / $num_bins;
    my @x_axis_labels;
    foreach my $x (@data) {
        my $bin_index = int( ($x - $min) / $bin_width );
        $hist[ $bin_index ]++;
    }
    foreach my $i (0..$num_bins) {
        my $xval_at_bin_edge = $min + $i * $bin_width;
        push @x_axis_labels, sprintf( "%.2f", $xval_at_bin_edge );
        push @desired_density, desired_density($xval_at_bin_edge);
    }
    @hist = map { $_ / $N } @hist;
    my ($hmin, $hmax) = Algorithm::MinMax->minmax( \@hist );
    my ($dmin, $dmax) = Algorithm::MinMax->minmax( \@desired_density );
    @desired_density = map { $_ * ($hmax / $dmax) } @desired_density;
    return [ \@x_axis_labels, \@hist, \@desired_density ];
}

# This subroutine constructs a Markov chain according to the
# Metropolis-Hastings algorithm.  This algorithm needs a
# proposal density, whose primary role is to help select
# the next sample given the current sample, and the desired
# density for the samples.  The number of samples constructed
# is determined by the sole argument to the subroutine.  Note
# that ideally you are supposed to discard many initial
# samples since it can take a certain number of iterations
# for the actual density of the samples to approach the
# desired density.
sub metropolis_hastings {
    my $N = shift;                # Number of samples
    my @arr;
    my $sample = 0;
    foreach my $i (0..$N-1) {
        print "Iteration number: $i\n" if $i % ($N / 10) == 0;

        # Get proposal probability q( $y | $x ).
        my ($newsample, $prob) = get_sample_using_proposal( $sample );
        my $a1 = desired_density( $newsample ) / desired_density( $sample );

        # IMPORTANT: In our case, $a2 shown below will always be 1.0
        # because the proposal density norm($x | $y) is symmetric with
        # respect to $x and $y:
        my $a2 = proposal_density( $sample, $newsample ) / $prob;
        my $a = $a1 * $a2;
        my $u = random_uniform();
        if ( $a >= 1 ) {
            $sample = $newsample;
        } else {
            $sample = $newsample if $u < $a;
        }
        $arr[$i] = $sample;
    }
    return @arr;
}

# This subroutine along with the subroutine proposal_density() do
# basically the same thing --- implement the normal density as the
# proposal density.  It is called proposal density because it is
# used to propose the next sample in the Markov chain.  The
# subroutine shown below returns both the proposed sample for
# the next time step and its probability according to
# the normal distribution norm($x, $sigma ** 2).   The next
# subroutine, proposal_density(), only evaluates the density
# for a given sample value.
sub get_sample_using_proposal {
    my $x = shift;
    my $mean = $x;      # for proposal_prob($y|$x)  =  norm($x, $sigma ** 2)
    my $sigma = 10;
    my $sample = random_normal( 1, $mean, $sigma );
    my $gaussian_exponent = - (($sample - $mean)**2) / (2 * $sigma * $sigma);
    my $prob = ( 1.0 / ($sigma * sqrt( 2 * PI ) ) ) * euler( $gaussian_exponent );
    return ($sample, $prob);
}

# As mentioned above, this subroutine returns the value of the
# norm($x, $sigma) at a given sample value where $x is the mean
# of the density.  The sample value where the density is to be
# known is supplied to the subroutine as its first argument.
sub proposal_density {
    my $sample = shift;
    my $mean = shift;
    my $sigma = 10;          # for  norm($mean, $sigma ** 2)
    my $gaussian_exponent = - (($sample - $mean)**2) / (2 * $sigma * $sigma);
    my $prob = ( 1.0 / ($sigma * sqrt( 2 * PI ) ) ) * euler( $gaussian_exponent );
    return $prob;
}

# This implements the desired density for an MCMC experiment.  That is,
# we want the first-order distribution of the Markov chain samples to
# possess the density as described by the function shown here:
sub desired_density {
    my $x = shift;
    return 0 if ($x < -10.0) or ($x > 20.0);
    my $prob = 0.3 * euler(-0.2*($x**2)) +  0.7 * euler(-0.2*(($x - 10.0)**2));
    return $prob;
}
```

- You can execute the code by calling `Metropolis_Hastings.pl`. It deposits the histogram of the values in the samples of the MCMC chain in a file called `histogram.gif`. Subsequently, you can display this histogram by calling, say, '`display histogram.gif`', where the `display` command from the `ImageMagick` suite of tools in your computer.

- In the MCMC based approach to the estimation of the best mapping for the embedding of the V nodes of a Freenet overlay on the location circle of Figure 9, we first define a state vector that consists of a $|V|$-tuple of real numbers from the interval $[0, 1)$. Starting from some randomly chosen state vector, we now construct a Markov chain with the Metropolis-Hastings algorithm so that the desired density is given by the posterior $p(\phi|E)$ we showed earlier. Note that the mapping ϕ can now be thought of as a state vector.

- In order to use the Metropolis-Hastings method, we need a proposal density $q(\mathbf{r}|\mathbf{s})$ that will help us generate a new candidate state $\mathbf{s}*$ from the current state $\mathbf{s}^i$. In "Distributed Routing in Small-World Networks," Sandberg has shown how we can define a symmetric proposal density that takes a Markov chain through a sequence of states, one state leading to another state by swapping just two node-to-location assignments subject to certain constraints, so that the resulting Markov chain will stabilize with the density that corresponds to the posterior $p(\phi|E)$ mentioned earlier.

- In the rest of this section, we will present a brief explanation of the

Sandberg algorithm as summarized from the paper "Routing in the Dark: Pitch Black" by Evans, GauthierDickey, and Grothoff. Recall that the goal is for a Freenet overlay to redo its location assignments so that the resulting assignments would constitute a small-world embedding in an imaginary base ring lattice. This the network will accomplish by having every pair of nodes examine their location keys periodically to see if they should switch their location keys. The following steps are undertaken by a pair of nodes to consider this switch:

1. Assume that nodes A and B are considering whether or not they should swap their location keys. Both A and B share the location assignments for their direct neighbors and they both compute the product $D_1(A, b)$ as defined below:

$$D_1(A,B) \quad = \quad \prod_{(A,n)\in E} |\phi(A) - \phi(n)|_s \cdot \prod_{(B,n)\in E} |\phi(B) - \phi(n)|_s$$

2. Next, the two nodes compute another product similar to the one above but under the assumption that they have swapped their location keys:

$$D_2(A,B) \quad = \quad \prod_{(A,n)\in E} |\phi(B) - \phi(n)|_s \cdot \prod_{(B,n)\in E} |\phi(A) - \phi(n)|_s$$

3. If $D_2 \leq D_1$, the two nodes A and B swap their location keys. Otherwise (and this is a direct consequence of how the logic of choosing the next state works in Metropolis-Hastings algorithm as shown earlier), the two nodes swap their location keys with the probability D_1/D_2.

- Let's apply the above algorithm to see if the nodes D and G in the Freenet overlay of Figure 2 (or, Figure 9) should swap their location keys. With the assignments as shown, we have

$$D_1(D,G) \quad = \quad (0.85{-}0.38)(0.70{-}0.38)(0.88{-}0.38)(0.73{-}0.38)\cdot(0.73{-}0.23)(0.73{-}0.32)$$

and

$$D_2(D,G) \quad = \quad (0.85-0.73)(0.73-0.70)(0.88-0.73)(0.73-0.38)\cdot(0.38-0.23)(0.38-0.32)$$

D_1 turns out to be equal to 0.0053 and D_2 to 0.000001. Since $D_1 > D_2$, the two nodes will swap their location keys.

- It is important to note that when two nodes swap their location keys that does not alter the physical connectivity of the network. In other words, with regard to who is whose neighbor, the network remains the same after a swap as it was before.

- From an operational standpoint, a major consequence of swapping the location keys is the possible migration of data objects from one node to another. Recall that the data objects are stored on the basis of closeness of their hash values to the location keys at a node. So if two nodes are swapping their location keys, they would also need to swap their data objects.

- Another major consequence of two nodes swapping their location keys is that any such swap will, in general, alter the search paths for locating a data object for retrieval and for finding the best node for storing a data object. As mentioned earlier in Section 26.2, when a new data object is inserted into the overlay, a bounded depth-first search is carried out for the node most appropriate for storing that data object. Since the branching decisions in this depth-first search are made on the basis of location keys, any time you change the location key associated with a physical node, you are likely to alter how that node appears in the search paths. The same applies to the search paths for retrieving data objects.

- Every pair of nodes in a Freenet overlay is supposed to **periodically** examine the location keys at the nodes to see if they need to be swapped.

- Sandberg has shown that this swapping action at every pair of nodes will eventually cause the location keys to converge to a state in which the routing needed for the GET and PUT requests will take only $O(\log N)$ steps with high probability.

26.8: SECURITY ISSUES WITH THE FREENET ROUTING PROTOCOL

- Apart from the problems that may be created by Denial-of-Service sort of attacks, I think it would be very difficult to subvert a small Freenet overlay in which each connection is created on the basis of the trust between the individuals involved. Obviously, a small group of friends who have created a Freenet overlay on the basis of mutual trust are not going to let an untrusted outsider access to their machines.

- However, Freenet overlays meant for large groups of people, especially when an overlay is thrown open to people who are not known personally to those on the inside, are vulnerable to problems that can be created by using fake location keys, engaging in illegal location key swaps to spread fake location keys, etc.

- Freenet is based on the assumption that the location keys are uniformly random, as are the hash keys for the data objects. When the keys are distributed uniformly over the nodes and the same is true for the data object keys, one can expect the nodes to be equally loaded. However, when this assumption regarding the distribution of keys is not valid, some nodes will see greater demands placed on their data stores than other nodes. As the reader will recall from Section 26.3, when there is pressure on the allocated memory, nodes are allowed to delete the least recently accessed data objects. For obvious reasons, data object deletion is more likely to occur when the nodes are non-uniformly loaded.

- As currently formulated, the Freenet protocol contains no verification mechanism to determine the authenticity of the location keys used at the different nodes. So, a malicious node (or malicious nodes acting in concert) could lay claim to portions of the location key space and let those keys propagate into the rest of the overlay through swapping. At the least, such location keys could cause the assumption of uniform distribution of keys to be violated.

- A security analysis of the Freenet routing was recently reported by Evans, GauthierDickey, and Grothoff in their paper "Routing in the Dark: Pitch Black". They describe two different attacks on Sandberg's routing algorithm: 1) Active Attack, and 2) Join-Leave Churn.

- Active attack consists of an attacking node (or a group of attacking nodes) to cause the location keys to get clustered in the neighborhood of some particular value. This can be done by taking advantage of the swapping action in the Sandberg routing protocol and also by taking advantage of the fact that an honest node has no means to verify the location value used by another neighboring node. Additionally, an honest node must accept a request to initiate location key swapping even if such a request is illegitimate. When the location keys become clustered, the assumption of the distribution of the location keys becomes invalid.

- Natural churn means the new nodes joining an overlay and existing nodes leaving. We can distinguish between the churn caused by **leave-join** activity and by the **join-leave** activity. As E, G, and G have pointed out, leave-join actions in which a node leaves the overlay temporarily and then rejoins with the same location key should not cause harm to the operation of a network because of how the network nodes cache the data objects (See Section 26.3 of these notes).

- The same cannot be said for **join-leave** actions in which a node stays in the overlay for a while and then leaves for good. This can result in loss of data objects, especially loss of those objects that are not cached elsewhere.

- The join-leave churn, however, has another impact that can cause the location keys to become clustered, thus invalidating the assumption of uniformity of distribution of the location keys. See the paper by Evans, GauthierDickey, and Grothoff for further details.

26.9: GOSSIPING IN SMALL-WORLD NETWORKS

- As mentioned earlier, a small world is characterized by short path lengths between any pair of nodes and high clustering coefficients locally. More precisely, as a network becomes larger and larger, the paths connecting any two nodes grow only logarithmically on the average and the clustering coefficient remains nearly constant. It is therefore interesting to investigate the following questions in such networks:

  1. If a rumor is injected into such a network and if at each time step those who have the rumor send it to others, how long will it take for the rumor to cover the entire network? [It is believed that models that can efficiently spread a rumor in a network can serve as models for how infectious diseases spread in human populations. The article by Demers et al. (see Section 26.10 for citation) was one of the earliest to draw these parallels between efficient distribution of information in a network and the spread of diseases.]

  2. Suppose each node makes a local observation that is of a numerical nature (such as the amount of free storage available at the node, the number of download requests for some locally

held resource, etc.), is it possible to compute aggregates of this information **using purely decentralized algorithms**? Decentralized algorithms use only locally available knowledge that exists at each node and at the node's immediate neighbors. **We want a decentralized algorithm to work in such a way that each node becomes aware of, say, the average of all the observations made by the different nodes of the network.** In particular, we are interested in what are known as **gossip algorithms** in which, at each time step, each node is allowed to communicate with only one other node from the rest of the network. In other words, at any given time step, a node may receive information from multiple nodes, but a node can send information to only one node.

- Both these questions are topics of great interest currently.

- Regarding both these questions, much is already known when a network forms a complete graph, that is, when it is possible for any node to communicate directly with every other node. Whereas a small-world network, in general, does not require that every pair of nodes have a direct communication link, such an assumption may not be too far off the mark for small Freenet overlays established on a friend-to-friend basis. In any case, various bounds may be safely assumed to be lower-bounded by the results obtained for complete graphs.

- Regarding the spreading of rumors, it is not that difficult to reason that the number of time steps it takes must be lower-bounded by $\log_2 N$ where N is the total number of nodes in a network. [The reasoning goes something like this: Assuming that a node can only talk to one other node at any given time, the node that first receives the rumor *could* contact another node randomly at the first time step. Subsequently, we would have two nodes that possess the rumor. At the next time step, these two nodes *could* spread the rumor to two other nodes. So we *could* end up with four holders of the rumor at the end of the second time step, and so on. This is obviously a geometric progression in powers of 2. For more exact bounds, how a rumor spreads depends obviously on the communication and the interconnection model assumed for the network. In 1987, Pittel showed that the time it takes for a rumor to completely cover a network is given by $\log_2 N + \ln N + O(1)$ as N becomes large.] How fast information injected at one node in a network spreads to all the other nodes is referred to as **the diffusion speed**.

- Regarding decentralized calculation of aggregates of the numerical parameters observed at the different nodes in a network, for illustration here is the Push-Sum algorithm by Kempe, Dobra, and Gehrke: Let x_i denote the observation at node i. At each time step t, each node computes a sum $s_{t,i}$ and a weight $w_{t,i}$. At node i, the sum is initialized by setting $s_{0,i} = x_0$ and the weight initialized by setting $w_{0,i} = 1$. Additionally, at time 0, the node i sends these two numbers, sum and weight, to itself. Subsequently, upon updating the sum and the weight using the received information, it sends $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ pair to a node randomly selected from the network and to itself. As the sum and weight pairs are received at each node, the decentralized algorithm works as follows:

```
Algorithm Push-Sum (by Kempe, Dobra, and Gehrke)
1. Let $(\hat{s}_r, \hat{w}_r)$ be all the pairs received at node $i$ at time $t-1$
2. Set $s_{t,i} = \sum_r \hat{s}_r$  and  $w_{t,i} = \sum_r \hat{w}_r$
3. Let node $i$ choose a target node $f_t(i)$ uniformly at random
4. Let node $i$ send the pair $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ to the target node $f_t(i)$
   and to itself.
5. At node $i$, the ratio $\frac{s_{t,i}}{w_{t,i}}$ is the estimate of the average at time
   $t$
```

- Kempe et al. have theoretically established that, with probability $1 - \delta$, the error in the estimate formed at each node vis-a-vis its network-wide true value will drop to ϵ in at most $O(\log N + \log \frac{1}{\epsilon} + \log \frac{1}{\delta})$ time steps. **Practically speaking, the estimate calculated at each node locally will converge to its network-wide true value in time proportional to the logarithm of the size of the network.** That is as efficient as it ever gets. But note that the guarantee made by the above algorithm regarding the convergence of the estimated answer to the true answer is probabilistic.

- The algorithm can be easily adapted to the decentralized computation of the sum of the local observations, as opposed to their average, by using the weight initialization $w_{t,i} = 1$ at only one node, while it is initialized to 0 at all other nodes.

- As mentioned earlier, the Kempe-Dobra-Gehrke algorithm is based on the assumption that the network graph is complete. Recently, Boyd, Ghosh, Prabhakar, and Shah have looked into aggregate-computing gossip algorithms for networks with arbitrary connection graphs.

26.10: FOR FURTHER READING

- Ian Clarke, “A Distributed Decentralized Information Storage and Retrieval System,” Technical Report, Division of Informatics, University of Edinburgh, 1999.
- Ian Clarke, “The Freenet Project,” http://freenetproject.org/
- Stanley Milgram, “The Small World Problem,” Psychology Today, pp. 60-67, 1967.
- Duncan Watts and Steven Strogatz, “Collective Dynamics of 'Small-World' Networks,” Nature, pp. 440-442, 1998.
- Albert-Laszlo Barabasi and Reka Albert, “Emergence of Scaling in Random Netowrks,” Science, pp. 509-512, 1999.
- Jon Kleinberg, “The Small-World Phenomenon: An Algorithmic Perspective,” Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'00), 2000.
- Oskar Sandberg, “Distributed Routing in Small-World Networks,” ALENEX 2006.
- Nathan Evans, Chris GauthierDickey, and Christian Grothoff, “Routing in the Dark: Pitch Black,” ACSAC 2007.
- Boris Pittel, “On Spreading a Rumor,” SIAM Journal Appl. Math, pp. 213-223, 1987.
- Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry, “Epidemic Algorithms for Replicated Database Maintenance,” Proc. of 7th ACM SOSP, pp. 1-12, 1987.
- David Kempe, Alin Dobra, and Johannes Gehrke, “Gossip-Based Computation of Aggregate Information,” Proc. IEEE Inter. Conf. on the Foundations of Computer Science, pp. 482-491, 2003.
- Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah, “Randomized Gossip Algorithms,” IEEE Trans. Information Theory, pp. 2508-2530, June 2006.