CS 336 Lecture 22– 4/13/98

Announcements

- Project designs due Friday
- Self-scheduled midterm available from my office or dept. secretary now until Friday.
- I have put a copy of the midterm from my last offering of this course (when it was a tutorial) online.

Shortest Path Routing

1. Last time we began discussing algorithms for finding shortest paths based on modeling networks as graphs
   - Each computer/router in the network is represented by a node in the graph.
   - Links between computers in the network are represented by directed edges (one in each direction).
   - A cost or weight is associated with each edge.

     For our discussion, we will use the symbols $d_{i \to j}$ to denote the cost (distance) of the edge from node i to node j.
   - We will use the symbols $D_{i \to j}$ to denote the length of the shortest path from i to j.
2. The first algorithm, known as the Bellman-Ford algorithm is based on a simple formula relating values of $D_{i \to j}$.
   - The Bellman-Ford algorithm works by computing a series, $D^i_{i \to j}$, of approximations to these values that converges to the actual path lengths.
   - Initally, the algorithm set $D^0_{i \to j} = \infty$ for $i \neq j$ and $D^0_{i \to i} = 0$.
   - The algorithm then repeatedly sets

     $$D^{n+1}_{i \to j} = \min_k (d_{i \to k} + D^n_{k \to j})$$

     until on some iteration nothing changes.
   - Note that his version of the Bellman-Ford algorithm differs from that in the book in several ways.
     - First, for now I'm assuming a sequential, centralized implementation. Shay presents a somewhat distributed version.
     - More importantly, the iteration step of the algorithm in the book is more accurately described by the formula:

       $$D^{n+1}_{i \to j} = \min_k (D^n_{i \to k} + D^n_{k \to j})$$

       That is, while the $d_{i \to j}$ values are used to initialize the algorithm, they are not involved in the update process. While this may speed the initial convergence of the algorithm, it makes it unable to adjust to dynamic changes in the costs associated with network links.
3. The second algorithms is Dijkstra's algorithm which is also sometimes known as the "SPF" or Shortest Path First algorithm.
   - Now, imagine we select some node "s" (for source) and try to find the best paths from that source to all the nodes in the graph.
   - The trick to Dijkstra's algorithm is that it finds the shortest path from "s" to each other node in the graph by considering nodes in order of the length of their shortest path from s.
     - Once "s" is selected, we can order all the nodes of the graph by the values of the shortest path from "s" to each node. That is we will place node i before node j if $D_{s \to i} < D_{s \to j}$.
   - To get the job started, it is obvious that the cost of the minimal path from s to itself is 0. Therefore, the algorithm begins with s and simply declares that the shortest path from s to s is the null path with cost 0.
   - At any point as the algorithm proceeds it will have found the shortest paths to some subset of the graph's nodes, W. For each node, i, in W it will know $D_{s \to i}$ The next step is to find the node in G-W whose distance from s is minimal.
   - Suppose the next node to be added is n. The minimal path from s to n can not contain any nodes in G-W other than n. If it did, then the length of the shotest path from s to that node would be less than the length of the shortest path from s to n (contradicting the assumption that n is the next node to add to W).

     So, to find "n", the algorithm only needs to consider paths contained completely in W except for the last step.
   - Therefore, rather than worrying about minimal path lengths as we go, what we can do is keep track of a value $D^W_{s \to j}$ which will be the length of the shortest path from s to j using only nodes of W as intermediate nodes.
     - Initally (i.e. when W only contains s), $D^W_{s \to j} = d_{s \to j}$ for all j.

- Assuming we keep track of the values of $D^W_{s\to j}$ as the algorithms proceeds then when we add a node "n" to W, all we need to do is set

$$D^W_{s\to j} = \min(D^W_{s\to j}, D^W_{s\to n} + d_{n\to j})$$

- In fact, when we update the D^W values after adding n, we only need to consider those j that are neighbors of n. Thus, the total work required to update the values of D^W over the course of the algorithm is proportional to the number of edges.
- Each time we add a node to W, we have to at most look at the D^W values of all but one node in the graph. So, the total cost of these additions to W will be proportional to the square of the number of nodes.

* To determine the correct new node "n" to add to W at each step we simply search for the node in G-W with associated with the smallest $D^W_{s\to n}$ value.
* Note that in the process of finding the lengths of the shortest paths from s to each node, the algorithm is effectively building a spanning tree for the graph rooted at s. At each step, the edge from some node in W to the node n in W that is the last step in the minimum length path from s to n is part of this tree. The best path from s to any n is then the only path from s to n in the spanning tree.

Shortest Path Algorithms in Practice

1. Having briefly considered the two basic approaches to finding shortest paths in graphs that are used commonly used as network routing algorithms, we now examine the advantages one has over the other.
   - In your typical algorithms course, our concern would probably involve the number of steps and amount of memory required by each algorithm.
   - Instead, we will primarily examine how well the algorithms fit relative to the requirements that network routing be done in a distributed rather than in a centralized fashion and that the routes be changed dynamically to adapt to changes in load or failures in the network.
2. We begin by looking at how one could distribute the computations performed by each of these algorithms.
   - However the actual computation is distributed, there are some simple things one can say about the way the data involved must be distributed through the network.
     - Although data values may be sent across the network to other machines as needed, each value will have a home machine corresponding to the machine that must generate the data or the node that needs the data most for routing.
     - If one keeps in mind the fact that we eventually need to consider how the algorithms will adapt to changes in load, it is clear that the edge costs, $d_{i\to j}$ must originate at the sending edge of the associated link, i.
     - In order to send packets each node ultimately needs information about the best route to every destination in the network. Ultimately, the value of $D_{i\to j}$ is needed at node i to determine the route to node j (at the very least, by checking this value node i can determine if node j is even reachable).
       So, it is natural to try to store the "primary" copy of each $D_{i\to j}$ at node i. travel a single hop!
   - Now, consider the basic step in the iteration of the Bellman-Ford algoritm:

$$D^{n+1}_{i\to j} = \min_k(d_{i\to k} + D^n_{k\to j})$$

     The node at which one would expect to update $D_{i\to j}$ is i. All of the $d_{i\to j}$ values needed for this update will also be at i.

     The $D_{k\to j}$ values will not be stored at i, but they will all be stored at neighbors of i. Therefore, the packets required to send them to i will only have to
   - This leads to a simple distributed version of the Bellman- Ford algorithm:
     - Each node i keeps just the list of values $D_{i\to j}$ of its best known routes to other nodes. This list is called a *distance vector*.
     - Periodically (at least within some timeout of and change in a distance vector value or the cost of one of the nodes links), each node sends a packet containing its distance vector to its neighbors.
     - Upon receiving a distance vector from a neighbor n, a node i uses the value of $d_{i\to n}$ together with the $D_{n\to j}$ values in the distance vector to see if any of its $D_{i\to j}$ values should be updated
     - This algorithm ultimately performs the same updates as the sequential Bellman-Ford algorithm (though in a potentially haphazard order). It can be shown to eventually compute the correct values regardless of the initial values chosen.
   - The distribution of data does not work out as well for Dijkstra's Shortest Path First algorithm.

- In one way, things looks better. Given a source node i, Dijksta's algorithm can be used to compute $D_{i \to j}$ for every j without bothering to compute $D_{k \to j}$ for any k not equal to i.

  By contrast with the Bellman-Ford algorithm given a destination node j, one can compute $D_{i \to j}$ for every i without having to determine $D_{i \to k}$ for any other destination k.

  This is basically because Dijkstra's algorithm starts building paths from the source working toward the destination while the Bellman-Ford algorithm starts at the destination and works back to sources.
- Alas, the information required for the basic iteration of Dijkstra's algorithm is not as localized as in the Bellman-Ford algorithm.

  When a node n is added to W and we attempt to update the values of $D^{W}_{i \to j}$ for each j that is a neighbor of n, we need the values $d_{n \to j}$. The shortest path from the source of this information, n, to i may involve many hops.

  Looked at another way, when a $d_{n \to j}$ values changes, the node n will have to make sure to notify not just its neighbors but every node in the graph of the change.