Section 6.3: Details of the Shortest Path Algorithms

October 29, 2007

Abstract

Some examples are provided in gory detail.

1 Shortest Path Algorithms

1.1 Dijkstra's Algorithm

Example: #3, p. 441/507

We start with the adjacency matrix:

$$\begin{bmatrix} \infty & 3 & 5 & \infty & 8 & 1 & \infty & \infty \\ 3 & \infty & 2 & \infty & \infty & \infty & 1 & \infty \\ 5 & 2 & \infty & 1 & \infty & \infty & \infty & 2 \\ \infty & \infty & 1 & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & 4 & \infty & 6 & \infty & 1 \\ 1 & \infty & \infty & \infty & 6 & \infty & 5 & \infty \\ \infty & 1 & \infty & \infty & \infty & 5 & \infty & 1 \\ \infty & \infty & 2 & \infty & 1 & \infty & 1 & \infty \end{bmatrix}$$

The adjacency matrix serves to indicate to the computer which nodes are adjacent to which others.

Now, we're going to keep track of the "settled nodes", starting with the initial node ($IN = \{1\}$). We will also keep track of their distances to node 1, and we'll keep a list of their nearest neighbor along their shortest path back to 1. They start this way, therefore:

	1	2	3	4	5	6	7	8
d	0	3	5	∞	8	1	∞	∞
s	–	1	1	1	1	1	1	1

Note that once a node is settled, its entries won't be changing.

The next closest node is 6: it is one unit away from 1. Any other node must be further away, as to go directly from 1 requires more than 1 unit, and passing by node 6 would still be farther than one unit (one unit and change). Hence 6 is added to $IN = \{1, 6\}$, and

	1	2	3	4	5	**6**	7	8
d	0	3	5	∞	8	1	∞	∞
s	–	1	1	1	1	1	1	1

Since node 6 has neighbors $\{1, 5, 7\}$, these are the only nodes whose distances could be updated. Node 1 will not change, however, as it is already settled in for its long winter's nap. Looking the others over, we see that there are some improvements to both 5 and 7:

	1	**2**	3	4	5	**6**	7	8
d	0	3	5	∞	7	1	6	∞
s	–	1	1	1	6	1	6	1

We now settle 2, with neighbors $\{1, 3, 7\}$, of which only 3 and 7 can change:

	1	**2**	3	4	5	**6**	**7**	8
d	0	3	5	∞	7	1	4	∞
s	–	1	1	1	6	1	2	1

Node 3 sticks, but 7 can be reached in only 4. This makes it our next settled node, with neighbors $\{2, 6, 8\}$: only 8 can still change! And it does:

	1	**2**	**3**	4	5	**6**	**7**	8
d	0	3	5	∞	7	1	4	5
s	–	1	1	1	6	1	2	7

We now take 3 as our next settled node, although the choice is arbitrary (both 3 and 8 are at distance 5). Node 3 has neighbors $\{1, 2, 4, 8\}$, so only 4 and 8 can change (and only 4 does):

	1	**2**	**3**	4	5	**6**	**7**	**8**
d	0	3	5	6	7	1	4	5
s	–	1	1	3	6	1	2	7

Now node 8 is settled, whose neighbors are $\{3, 5, 7\}$. Only 5 can change, and it does:

	1	**2**	**3**	4	5	**6**	**7**	**8**
d	0	3	5	6	6	1	4	5
s	–	1	1	3	8	1	2	7

Now, if our algorithm is smart, it will decide ties in favor of the destination node. So let's assume a smart algorithm: then the next settled node will be node 5, our final destination. The arrays end, then, as

	1	**2**	**3**	4	**5**	**6**	**7**	**8**
d	0	3	5	6	6	1	4	5
s	–	1	1	3	8	1	2	7

which indicates that the shortest path between nodes 1 and 5 has weight or cost 6, and the path is given by the s array: 5's adjacent neighbor on the path to 1 is 8; 8's is 7; 7's is 2; and 2's is 1. Hence a shortest path is

$$5 -> 8 -> 7 -> 2 -> 1$$

(note that the shortest path may not be unique: hence we say "a" rather than "the").

1.2 Bellman-Ford Algorithm

This algorithm allows us to find the shortest distance from the initial node to all other nodes, and is hence a generalization of Dijkstra's algorithm (at least as presented in our book).

We're going to compute shortest paths of 1 arc, 2 arcs, ..., (n-1) arcs, which are the longest paths we would possibly use to get to any node from 1 (otherwise we would be visiting a node twice, which would be foolish!).

Example: #12, p. 444/509

Fortunately we're using the same graph, so the adjacency matrix is essentially the same. We start with the adjacency matrix

$$\begin{bmatrix} 0 & 3 & 5 & \infty & 8 & 1 & \infty & \infty \\ 3 & 0 & 2 & \infty & \infty & \infty & 1 & \infty \\ 5 & 2 & 0 & 1 & \infty & \infty & \infty & 2 \\ \infty & \infty & 1 & 0 & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & 4 & 0 & 6 & \infty & 1 \\ 1 & \infty & \infty & \infty & 6 & 0 & 5 & \infty \\ \infty & 1 & \infty & \infty & \infty & 5 & 0 & 1 \\ \infty & \infty & 2 & \infty & 1 & \infty & 1 & 0 \end{bmatrix}$$

with zeros down the diagonal in place of the infinities before.

We essentially add each row of the adjacency matrix to the current d vector, and check to see if we get any improvement. If so, we've found a shorter path! d contains the shortest distances determined so far from the initial node to every node in the graph.

Once again we're going to keep track of the distances from the initial node, starting with the initial node 1. We will also keep track of their nearest neighbor along their shortest path back to 1. They start this way, therefore:

	1	2	3	4	5	6	7	8
d	0	3	5	∞	8	1	∞	∞
s	–	1	1	1	1	1	1	1

Only when a node changes will it impact other nodes. We now ask about paths using two arcs: what are the shortest distances for each node from node 1? In order to answer this question, you need to examine each node's neighbors (use the adjacency matrix!), and check their nearest distances. Again, if these neighbor distances have not changed from one step to the next, then the distance to the given node will not change either!

At the second iteration, paths of two arcs, our distances look like this:

	1	2	3	4	5	6	7	8
d	0	3	5	6	7	1	4	7
s	–	1	1	3	6	1	2	3

For example, if we add the fourth node's row of the adjacency matrix to the original d array, we get

	1	2	3	4	5	6	7	8
$d + fourth$	∞	∞	6	∞	12	∞	∞	∞

which says that we can get to node 4 in 6, using node 3. We do the same for all the other nodes (other rows of the adjacency matrix).

Iterate: we again step through the rows, checking the neighbors of each node against their newly calculated values to see if there's any improvement. Only for node 8 do we see any change:

	1	2	3	4	5	6	7	8
d	0	3	5	6	7	1	4	5
s	–	1	1	3	6	1	2	7

Node 8's only neighbors are $\{3, 5, 7\}$, so only these can change in the next step: we use their rows from the adjacency matrix, and try again:

	1	2	3	4	5	6	7	8
d	0	3	5	6	6	1	4	5
s	–	1	1	3	8	1	2	7

Only 5 changed. Its neighbors are $\{1, 4, 7, 8\}$, but none of them change. Hence we are done! There can be no further change.

The d array gives us the nearest distances to 1 for each node, and their paths can be calculated exactly as for Dijkstra's algorithm.

1.3 Floyd's Algorithm

Example: #15, p. 444/510

The output below, from a "smarter implementation" in lisp that I wrote, shows the distances above the diagonal, and the original adjacency matrix below the diagonal.

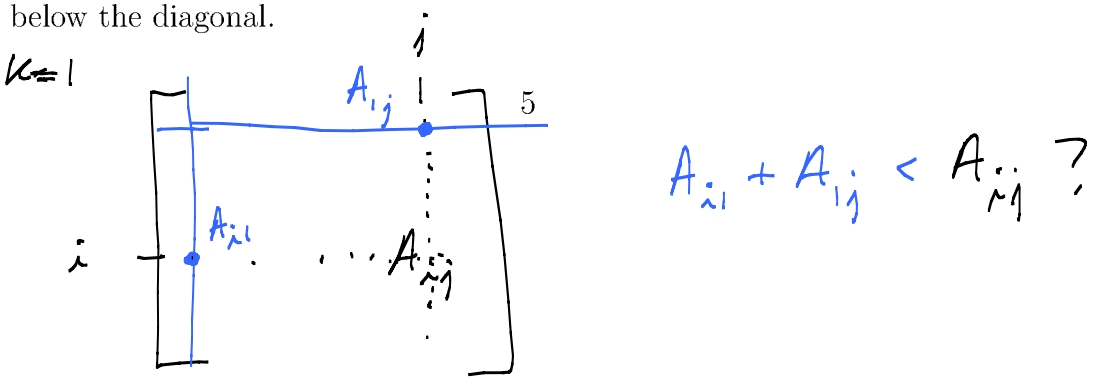

Table 1: Initial adjacency matrix

0	1	∞	4	∞
1	0	3	1	5
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0

Table 2: After the sweep with k=0 (for x - by indexing from 0, we'll have k line up with the node label for the rest of our iterations).

0	1	∞	4	∞
1	0	3	1	5
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0

Table 3: After the sweep with k=1

0	1	4	2	6
1	0	3	1	5
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0

Table 4: After the sweep with k=2

0	1	4	2	6
1	0	3	1	5
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0

Table 5: After the sweep with k=3

0	1	4	2	5
1	0	3	1	4
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0

Table 6: End matrix, after the sweep with k=4 (for y), with the shortest distances above the diagonal, and the original adjacency values below the diagonal.

0	1	4	2	5
1	0	3	1	4
∞	3	0	2	2
4	1	2	0	3
∞	5	2	3	0