Lecture 14:
Shortest Paths

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Problem of the Day

Suppose we are *given* the minimum spanning tree T of a given graph G (with n vertices and m edges) and a new edge $e = (u, v)$ of weight w that we will add to G. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit, although slower but correct algorithms will receive partial credit.

Applications for Shortest Paths

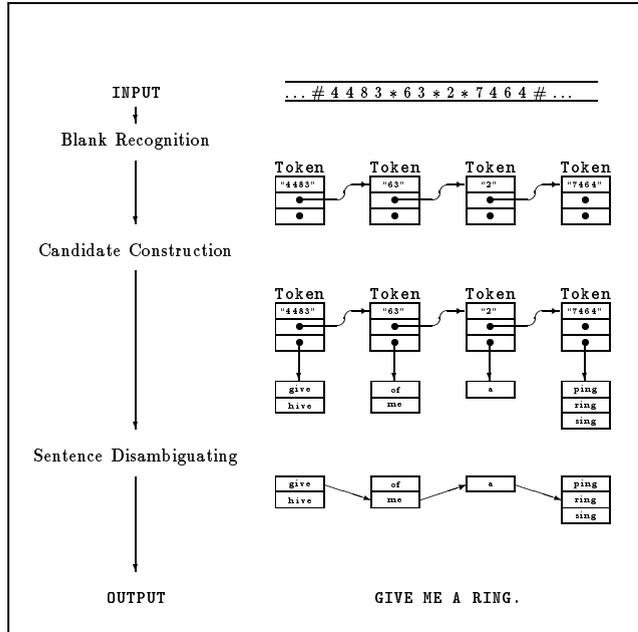Finding the shortest path between two nodes in a graph arises in many different applications:

- Transportation problems – finding the cheapest way to travel between two locations.

- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.

- Communications problems – how look will it take for a message to get between two places? Which two locations are furthest apart, ie. what is the *diameter* of the network.

Shortest Paths and Sentence Disambiguation

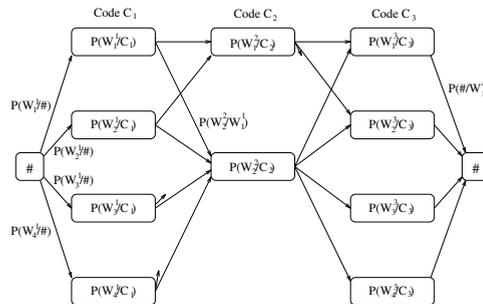In our work on reconstructing text typed on an (overloaded) telephone keypad, we had to select which of many possible interpretations was most likely.

We constructed a graph where the vertices were the possible words/positions in the sentence, with an edge between possible neighboring words.

The final system identified over 99% of characters correctly based on grammatical and statistical constraints.

INPUT   ... # 4 4 8 3 * 6 3 * 2 * 7 4 6 4 # ...

Blank Recognition

Token	Token	Token	Token
"4483"	"63"	"2"	"7464"
●	●	●	●
●	●	●	●

Candidate Construction

Token	Token	Token	Token
"4483"	"63"	"2"	"7464"
●	●	●	●
●	●	●	●

give	of	a	ping
hive	me		ring
			sing

Sentence Disambiguating

give	of	a	ping
hive	me		ring
			sing

OUTPUT   GIVE ME A RING.

Weighting the Graph

The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. 'hive me' would be less than 'give me', for example.

Dynamic programming (the Viterbi algorithm) can be used to find the shortest paths in the underlying DAG.

Shortest Paths: Unweighted Graphs

In an unweighted graph, the shortest path uses the minimum number of edges, and can be found in $O(n + m)$ time via breadth-first search.

In a weighted graph, the weight of a path between two vertices is the sum of the weights of the edges on a path.

BFS will not work on weighted graphs because visiting more edges can be less distance, e.g. $1+1+1+1+1+1+1 < 10$. There can be an exponential number of shortest paths between two nodes – so we cannot report *all* shortest paths efficiently.

Negative Edge Weights

Negative cost cycles render the problem of finding the shortest path meaningless, since you can always loop around the negative cost cycle more to reduce the cost of the path.

Thus we will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges.

Minimum spanning trees are unaffected by negative cost edges.

Dijkstra's Algorithm

The principle behind Dijkstra's algorithm is that if $(s, \ldots, x, \ldots, t)$ is the shortest path from s to t, then $(s, \ldots, x)$ had better be the shortest path from s to x.

This suggests a dynamic programming-like strategy, where we store the distance from s to all nearby nodes, and use them to find the shortest path to more distant nodes.

Initialization and Update

The shortest path from s to s, $d(s, s) = 0$. If all edge weights are positive, the *smallest* edge incident to s, say (s, x), defines $d(s, x)$.
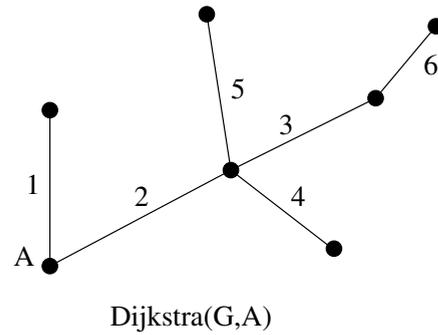
Soon as we establish the shortest path from s to a new node x, we go through each of its incident edges to see if there is a better way from s to other nodes thru x.

Pseudocode: Dijkstra's Algorithm

$known = \{s\}$
for $i = 1$ to n, $dist[i] = \infty$
for each edge (s, v), $dist[v] = d(s, v)$
last=s
while ($last \neq t$)
      select v such that $dist(v) = \min_{unknown} dist(i)$
      for each (v, x), $dist[x] = \min(dist[x], dist[v] + w(v, x))$
      last=v
      $known = known \cup \{v\}$

This is essentially the same as Prim's algorithm.

Dijkstra Example

G

Dijkstra(G,A)

Dijkstra's Implementation

See how little changes from Prim's algorithm!

```
dijkstra(graph *g, int start) (* WAS prim(g,start) *)
{
        int i; (* counter *)
        edgenode *p; (* temporary pointer *)
        bool intree[MAXV]; (* is the vertex in the tree yet? *)
        int distance[MAXV]; (* distance vertex is from start *)
        int v; (* current vertex to process *)
        int w; (* candidate next vertex *)
        int weight; (* edge weight *)
        int dist; (* best current distance from start *)

        for (i=1; i<=g->nvertices; i++) {
                intree[i] = FALSE;
                distance[i] = MAXINT;
                parent[i] = -1;
        }

        distance[start] = 0;
        v = start;
```

```
        while (intree[v] == FALSE) {
                intree[v] = TRUE;
                p = g− >edges[v];
                while (p ! = NULL) {
                        w = p− >y;
                        weight = p− >weight;
(* CHANGED *)           if (distance[w] > (distance[v]+weight)) {
(* CHANGED *)                   distance[w] = distance[v]+weight;
(* CHANGED *)                   parent[w] = v;
                        }
                        p = p− >next;
                }

                v = 1;
                dist = MAXINT;
                for (i=1; i<= g− >nvertices; i++)
                        if ((intree[i] == FALSE) && (dist > distance[i])) {
                                dist = distance[i];
                                v = i;
                        }
        }
}
```

Prim's/Dijkstra's Analysis

Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list.

After adding new vertex v to the tree, running through its adjacency list to update the cost of adding fringe vertices if we found a cheaper way through v can be done in $O(n)$ time. The total time is $n \times n = O(n^2)$.

Better Data Structures = Improved Time

An $O(m \lg n)$ implementation of Dijkstra's algorithm would be faster for sparse graphs, and comes from using a heap of the vertices (ordered by distance), and updating the distance to each vertex (if necessary) in $O(\lg n)$ time for each edge out from freshly known vertices.

Even better, $O(n \lg n + m)$ follows from using Fibonacci heaps, since they permit one to do a decrease-key operation in $O(1)$ amortized time.

Problem of the Day

Let $G = (V, E)$ be an undirected weighted graph, and let T be the shortest-path spanning treerooted at a vertex v. Suppose now that the edge weights in G are increased by a constant number k. Is T still the shortest-path-spanning tree from v?

All-Pairs Shortest Path

Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.

Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices.

We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n^3)$. Can we do better?

Dynamic Programming and Shortest Paths

The four-step approach to dynamic programming is:

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute this recurrence in a bottom-up fashion.

4. Extract the optimal solution from computed information.

Initialization

From the adjacency matrix, we can construct the following matrix:

$$D[i,j] = \infty, \qquad \text{if } i \neq j \text{ and } (v_i, v_j) \text{ is not in } E$$
$$D[i,j] = w(i,j), \quad \text{if } (v_i, v_j) \in E$$
$$D[i,j] = 0, \qquad \text{if } i = j$$

This tells us the shortest path going through no intermediate nodes.

The Floyd-Warshall Algorithm

An alternate recurrence yields a more efficient dynamic programming formulation. Number the vertices from 1 to n.

Let $d[i, j]^k$ be the shortest path from i to j using only vertices from $1, 2, \ldots, k$ as possible intermediate vertices.

This path from i to j either goes through vertex k, or it doesn't.

What is $d[j, j]^0$? With no intermediate vertices, any path consists of at most one edge, so $d[i, j]^0 = w[i, j]$.

Recurrence Relation

Adding a new vertex k helps if and only if a path goes through it, so for $1 \leq k \leq n$:

$$d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$$

Computing the values of this recursive equation defines an algorithm for finding the all pairs shortest-path costs.
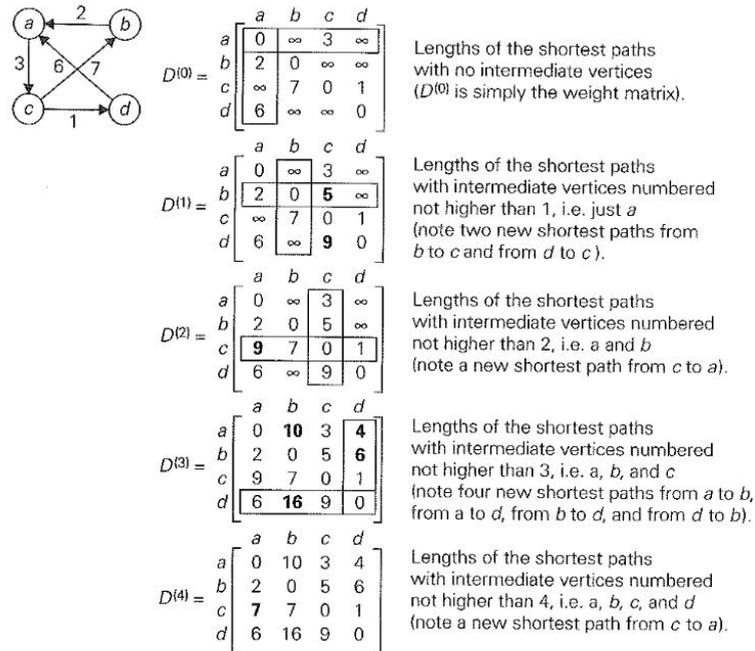
Floyd-Warshall Example

$D^{(0)} = \begin{array}{c@{\quad}} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array}\right] \end{array}$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} = \begin{array}{c@{\quad}} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{array}\right] \end{array}$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just a (note two new shortest paths from b to c and from d to c).

$D^{(2)} = \begin{array}{c@{\quad}} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{array}\right] \end{array}$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. a and b (note a new shortest path from c to a).

$D^{(3)} = \begin{array}{c@{\quad}} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{cccc} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{array}\right] \end{array}$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a, b, and c (note four new shortest paths from a to b, from a to d, from b to d, and from d to b).

$D^{(4)} = \begin{array}{c@{\quad}} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \left[\begin{array}{cccc} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{array}\right] \end{array}$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. a, b, c, and d (note a new shortest path from c to a).

FIGURE 8.7 Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.
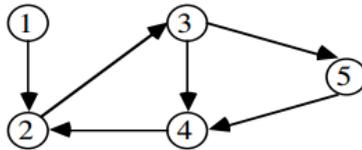
Implementation

The following algorithm implements it:

$d^o = w$
for $k = 1$ to n
    for $i = 1$ to n
        for $j = 1$ to n
        $d[i,j]^k = \min(d[i,j]^{k-1}, d[i,k]^{k-1} + d[k,j]^{k-1})$

This obviously runs in $\Theta(n^3)$ time, which is asymptotically no better than n calls to Dijkstra's algorithm.
However, the loops are so tight and it is so short and simple that it runs better in practice by a constant factor.

Transitive Closure

The *transitive closure* C of a directed graph A adds edge (i, j) to C if there is a path from i to j in A.

Transitive closure propagates the logical consequences of facts in a database, e.g. Is x related to y?