CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC13: SHORTEST PATH

Lecture slide courtesy of Prof. Steven Skiena

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: UNWEIGHTED GRAPHS

- In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which 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 sometimes visiting more edges can lead to shorter distance,
  - ie.1 + 1 + 1 + 1 + 1 + 1 + 1 < 10.
- Note that there can be an exponential number of shortest paths between two nodes – so we cannot report all shortest paths efficiently.

NEGATIVE EDGE WEIGHTS

- Note that 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 in our discussions, 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,...,x,...,t is the shortest path from s to t, then s,...,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).
- We can use an array to store the length of the shortest path to each node. Initialize each to1 to start.
- 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

ShortestPath-Dijkstra(G, s, t)

  $known = \{s\}$

  for $i = 1$ to n, $dist[i] = \infty$

  for each edge (s, v), $dist[v] = w(s, v)$

  $last = s$

  while ($last$!= t)

    select v_{next}, the unknown vertex minimizing $dist[v]$

    for each edge (v_{next}, x)

      $dist[x] = \min[dist[x], dist[v_{next}] + w(v_{next}, x)]$

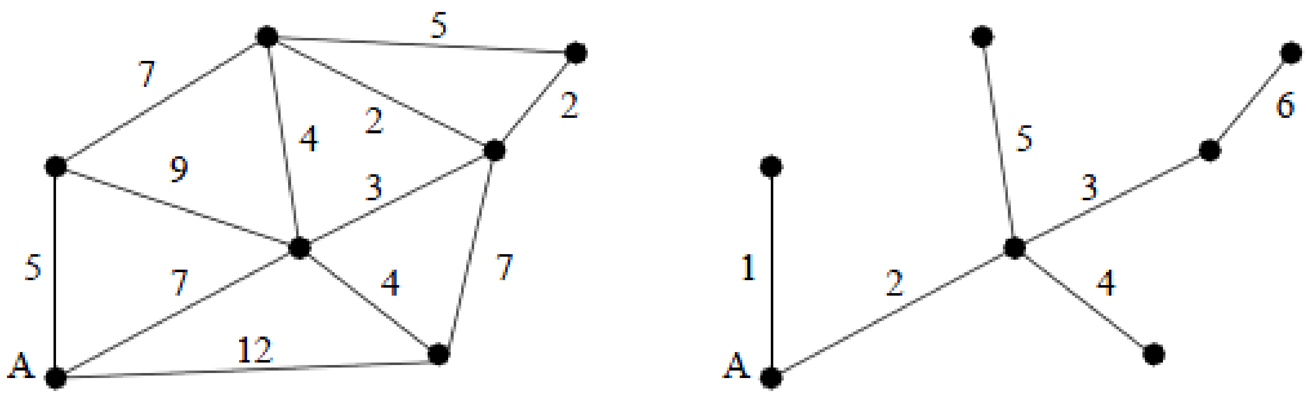
    $last = v_{next}$

    $known = known \cup \{v_{next}\}$

Complexity: $O(n^2)$.

The basic idea is very similar to 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+1];           /* is the vertex in the tree yet? */
    int distance[MAXV+1];          /* 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;
            if (distance[w] > (distance[v]+weight)) {  /* CHANGED */
                distance[w] = distance[v]+weight;    /* CHANGED */
                parent[w] = v;                       /* CHANGED */
            }
            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 a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices (there may be a cheaper way through the new vertex) can be done in O(n) time.
- Total time is $O(n^2)$.

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:
  - $D[i, j] = \infty$, if $i \neq j$ and (v_i, v_j) is not in E
  - $D[i, j] = w(i, j)$, if $(v_i; v_j) \in E$
  - $D[i, j] = 0$, if $i = j$

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

CHARACTERIZATION BASED ON PATH LENGTH

- There are several ways to characterize the shortest path between two nodes in a graph.
- Note that the shortest path from i to j, $i \neq j$, using at most M edges consists of the shortest path from i to k using at most *M-1 edges+W(k; j)* for some k.
- This suggests that we can compute all-pair shortest path with an induction based on the number of edges in the optimal path.

RECURRENCE ON PATH LENGTH

- Let $d[i, j]^m$ be the length of the shortest path from i to j using <u>at most m edges</u>.
- What is $d[i, j]^0$?

  $d[i, j]^0 = 0$ if $i = j$

  $= 1$ if $i \neq j$

- What if we know $d[i, j]^{m-1}$ for all i, j?

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

  $= \min(d[I, k]^{m-1} + w[k,j]);\ 1 \leq k \leq i$

  since $w[k, k] = 0$

NOT FLOYD IMPLEMENTATION

- This gives us a recurrence, which we can evaluate in a bottom up fashion:

for i = 1 to n
    for j = 1 to n
        $d[i,j]^m = \infty$
        for k = 1 to n
            $d[i,j]^0 = Min(\, d[i,k]^m, d[i,k]^{m-1} + \, d[k,j])$

TIME ANALYSIS

- This is an $O(n^3)$ algorithm just like matrix multiplication, but it only goes from m to m + 1 edges.
- Since the shortest path between any two nodes must use at most n edges (unless we have negative cost cycles), we must repeat that procedure n times (m = 1 to n) for an $O(n^4)$ algorithm.
- Although this is slick, observe that even $O(n^3 \log n)$ is faster than running Dijkstra's algorithm starting from each vertex!

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,…,k *as possible intermediate vertices.*
- 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

- In general, adding a new vertex k + 1 helps iff a path goes through it, so

$$d[i,j]^k \quad = w[i,j] \text{ if } k = 0$$

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

- Although this looks similar to the previous recurrence, it isn't.

IMPLEMENTATION

- The following algorithm implements it:

$d^0 = w$

for k = 1 to n

  for l = 1 to n

    for k = 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.