CS 6901 (Applied Algorithms) – Lecture 13

Antonina Kolokolova*

October 30, 2014

All pairs Shortest Path Problem

In the previous lectures we considered the problem of finding a shortest (cheapest) path from a given vertex s in a graph to all other vertices. Now, we want to find all pairwise costs of cheapest paths at the same time. That is, given a directed, weighted graph, we wish to find, for every pair of vertices u and v, the cost of a cheapest path from u to v.

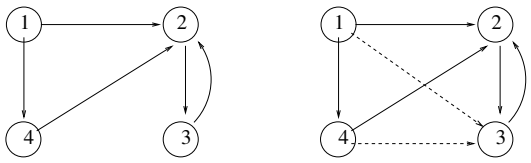
If the graph has only non-negative weights, and the number of edges is not much more than the number of vertices (that is, the graph is sparse), then running Dijkstra's algorithm n times is a decent solution. Recall that the running time of Dijkstra's algorithm (standard implementation, connected graph) is $O(m \log n)$, so running it n times will give time $O(nm \log n)$. In this lecture, the main general algorithm we will show is of time $O(n^3)$, so running Dijkstra's algorithm n times beats it for $m < n^2/\log n$. For sparse graphs with negative weights the algorithm by Johnson in fact exploits Dijkstra's algorithm (as well as Bellman-Ford algorithm) to solve the all-pairs shortest path problem: for graphs with no negative cycles, it uses the distances computed by Bellman-Ford to make all weights positive, and then runs Dijkstra's n times.

But now suppose that we got a dense graph, that is, the number of edges is closer to n^2. In that case, the two main approaches to all-pairs shortest path problems are the dynamic programming solution (Floyd-Warshall algorithm) that we will spend most of the lecture discussing, and a solution relying on a version of matrix multiplication (where the entry is a min over sums of pairs of cells, rather than the sum of products). In both cases, we will switch from a adjacency-list to adjacency-matrix representation of the graph.

As a warm-up, and give a taste of the matrix multiplication techniques, consider computing not a shortest (cheapest) path, but just the transitive closure of a graph: that is, a graph G^T which has an edge for every pair (u, v) of vertices from the original graph G such that there is a path in G from u to v.

*This set of notes is based on the course notes of U. of Toronto CS 364 as taught by Stephen Cook

For example, here is a graph and next to it its transitive closure (additional edges are shown as dashed lines). As there is no path from any vertex to vertex 1, or from 2 or 3 to 4, the corresponding edges are not in the transitive closure. A transitive closure of a strongly connected graph is a complete graph.

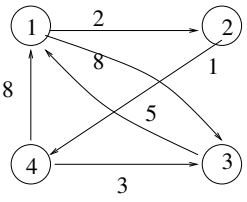


To compute transitive closure, start by representing the graph G by its adjacency matrix M, with 1 for cells corresponding to edges (or (i, i), and 0 otherwise. In this case, we will treat these 0 and 1 as Boolean False and True, and do the multiplication accordingly, with multiplication of entries interpreted as an AND, and addition as an OR. Now, if M is a Boolean matrix encoding the graph, then M^2, is a matrix encoding the graph with 1 for every path of length 2 in G, M^3 gives all paths of length at most 3 and so on, with M^n coding the transitive closure of the graph. The complexity of this algorithm is $O(n^4)$ if we multiply by M^i by M at each step; it can be improved to $O(n^3 \log n)$ by doing multiplication using repeated squaring: computing M^2, M^4, M^8 and so on, $\log n$ times (and keeping additional values if n is not a power of 2).

However, Floyd-Warshall algorithm that we will give provides an $O(n^3)$ solution to transitive closure, as a special case of all-pairs shortest path (any cell that is non-infinity will be in the transitive closure). But if more advanced matrix multiplication methods are used, such as Coppersmith-Winograd algorithm, then the algorithm above becomes asymptotically faster. However, these advanced methods tend to have large constants hidden in them, which makes them less practical.

Next, we will describe the Floyd-Warshall dynamic programming algorithm for the all-pairs shortest path problem.

Let us assume that $V = \{1, \cdots n\}$, and that *all* edges are present in the graph. We are given n^2 costs $c(i, j) \in \mathbb{R} \cup \{\infty\}$ for every $1 \leq i, j \leq n$. Also assume that the graph does not contain negative cycles. We allow ∞ as a cost in order to denote that we really view that edge as not existing.



	1	2	3	4
1	0	2	8	∞
2	∞	0	∞	1
3	5	∞	0	∞
4	8	∞	3	0

For $1 \leq i, j \leq n$, define $D(i, j)$ to be the cost of a cheapest path from i to j. Our goal is to compute *all* of the values $D(i, j)$.

Step 1: Describe an array of values we want to compute.
For $0 \leq k \leq n$ and $1 \leq i, j \leq n$, define
$A(k, i, j)$ = the cost of a cheapest path from i to j, from among those paths from i to j whose *intermediate* nodes are all in $\{1, \cdots, k\}$. (We define the intermediate nodes of the path $v_1, \cdots, v_m$ to be the set $\{v_2, \cdots, v_{m-1}\}$; note that if m is 1 or 2, then this set is empty.)

Note that the values we are ultimately interested in are the values $A(n, i, j)$ for all

$1 \leq i, j \leq n$.

Step 2: Give a recurrence.

- If $k = 0$ and $i = j$, then $A(k, i, j) = 0$.
  If $k = 0$ and $i \neq j$, then $A(k, i, j) = c(i, j)$.

  This part of the recurrence is obvious, since a path with *no* intermediate nodes can only consist of 0 or 1 edge.

- If $k > 0$, then $A(k, i, j) = \min\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$.

  The reason this equation holds is as follows. We are interested in cheapest paths from i to j whose intermediate vertices are all in $\{1, \cdots, k\}$. Consider a cheapest such path p. If p doesn't contain k as an intermediate node, then p has cost $A(k-1, i, j)$; if p does contain k as an intermediate node, then (since there are no negative cycles) we can assume that k occurs only once as an intermediate node on p. The subpath of p from i to k must have cost $A(k-1, i, k)$ and the subpath from k to j must have cost $A(k-1, k, j)$, so the cost of p is $A(k-1, i, k) + A(k-1, k, j)$.

We leave it as an exercise to give a more rigorous proof of this recurrence along the lines of the proof given for the problem of scheduling with deadlines, profits and durations.

Step 3: Give a high-level program.
We could compute the values we want using a 3-dimensional array $B[0..n, 1..n, 1..n]$ in a very straightforward way. However, it suffices to use a 2-dimensional array $B[1..n, 1..n]$; the idea is that after k executions of the body of the for-loop, $B[i, j]$ will equal $A(k, i, j)$. We will also use an array $B'[1..n, 1..n]$ that will be useful when we want to compute cheapest paths (rather than just costs of cheapest paths) in Step 4.

All_Pairs_CP

```
for i : 1..n do
    B[i, i] ← 0
    B'[i, i] ← 0
    for j : 1..n such that j ≠ i do
        B[i, j] ← C(i, j)
        B'[i, j] ← 0
    end for
end for

for k : 1..n do
    for i : 1..n do
        for j : 1..n do
            if B[i, k] + B[k, j] < B[i, j] then
              B[i, j] ← B[i, k] + B[k, j]
```

```
            B'[i,j] ← k
          end if
        end for
      end for
    end for
```

We want to prove the following lemma about this program.

Lemma: For every k, i, j such that $0 \leq k \leq n$ and $1 \leq i, j \leq n$, after the kth execution of the body of the for-loop the following hold:

- $B[i,j] = A(k,i,j)$
- $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $A(k,i,j)$. (Note that this implies that $B'[i,j] \leq k$).

Proof: We prove this by induction on k. The base case is easy. To see why the induction step holds for the first part, we only have to worry about the fact that when we are computing the kth version of $B[i,j]$, some elements of B have already been updated. That is, $B[i,k]$ might be equal to $A(k-1,i,k)$, or it might have already been updated to be equal to $A(k,i,k)$ (and similarly for $B[k,j]$); however this doesn't matter, since $A(k-1,i,k) = A(k,i,k)$. The rest of the details, including the part of the induction step for the second part of the Lemma, are left as an exercise. □

This Lemma implies that when the program has finished running, $B'[i,j]$ is the smallest number such that there exists a path p from i to j all of whose intermediate vertices are in $\{1, \cdots, B'[i,j]\}$, such that the cost of p is $D(i,j)$.

Step 4: Compute an optimal solution.
For this problem, computing an optimal solution can mean one of two different things. One possibility is that we want to print out a cheapest path from i to j, for *every* pair of vertices (i,j). Another possibility is that after computing B and B', we will be given an arbitrary pair (i,j), and we will want to compute a cheapest path from i to j as quickly as possible; this is the situation we are interested in here.

Assume we have already computed the arrays B and B'; we are now given a pair of vertices i and j, and we want to print out the edges in some cheapest path from i to j. If $i = j$ then we don't print out anything; otherwise we will call PrintOpt(i,j). The call PrintOpt(i,j) will satisfy the following Precondition/Postcondition pair:
Precondition: $1 \leq i, j \leq n$ and $i \neq j$.
Postcondition The edges of a path p have been printed out such that p is a path from i to j, and such that all the intermediate vertices of p are in $\{1, \cdots, B'[i,j]\}$, and such that no

vertex occurs more than once in p. (Note that this holds even if there are edges of 0 cost in the graph.)

The full program (assuming we have already computed the correct values into B') is as follows:

procedure PRINTOPT(i, j)
  $k \leftarrow B'[i, j]$
  if $k = 0$ then
    put "edge from",i,"to",j
  else
    PRINTOPT(i, k)
    PRINTOPT(k, j)
  end if
end PRINTOPT

if $i \neq j$ then PRINTOPT(i, j) end if

Exercise:
Prove that the call PRINTOPT(i, j) satisfies the above Precondition/Postcondition pair.
Prove that if $i \neq j$, then PRINTOPT(i, j) runs in time linear in the number of edges printed out; conclude that the whole program in Step 4 runs in time $O(n)$.

Analysis of the Running Time
The program in Step 3 clearly runs in time $O(n^3)$, and the Exercise tells us that the program in Step 4 runs in time $O(n)$. So the total time is $O(n^3)$. We can view the size of the input as n – the number of vertices, or as n^2 – an upper bound on the number of edges. In any case, this is clearly a polynomial-time algorithm. (Note that if in Step 4 we want to print out cheapest paths for *all* pairs i, j, this would still take just time $O(n^3)$.)

Remark: The recurrence in Step 2 is actually not as obvious as it might at first appear. It is instructive to consider a slightly different problem, where we want to find the cost of a *longest* (that is, most expensive) path between every pair of vertices. Let us assume that there is an edge between every pair of vertices, with a cost that is a real number. The notion of longest path is still not well defined, since if there is a cycle with positive cost, then there will be arbitrarily costly paths between every pair of points. It does make sense, however, to ask for the length of a longest *simple* path between every pair of points. (A simple path is one on which no vertex repeats.) So define $D(i, j)$ to be the cost of a most expensive simple path from i to j. Define $A(k, i, j)$ to be the cost of a most expensive path from i to j from among those whose intermediate vertices are in $\{1, 2, \ldots, k\}$. Then it is *not* necessarily true that $A(k, i, j) = \max\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}$. Do you see why?