CIS 575. Introduction to Algorithm Analysis
Material for March 22, 2024

Dynamic Programming for All-Pairs Shortest Paths

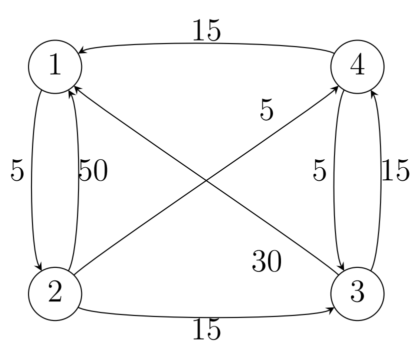
The topic of this note is presented in *Cormen*'s Section 23.2.

1 All Pairs Shortest Path: Problem

We are given a *directed* graph with n nodes, named $1 \ldots n$, where edges have "length": if there is an edge from node i to node j then we let $L(i, j)$ denote the length of that edge.

Our goal is for *each*[1] $i, j \in 1 \ldots n$ to find the *shortest* path from i to j, where "shortest" measures *not* the number of edges but the **sum of their lengths**. (Observe that the shortest path from i to j does not necessarily have the same length as the shortest path from j to i.)

As our running example, we shall consider the graph



Observe that from node 1 to node 3 there is no path with only one edge, but there is a path with two edges: it goes through node 2, and has length $5 + 15 = 20$. Still, the shortest path from node 1 to node 3 has *three* edges: it goes through first node 2 and next node 4, for a total length of $5 + 5 + 5 = 15$.

> *Remark:* we can actually allow the length of an edge to be negative. But we can*not* allow the presence of any *cycle* with negative length, as then no shortest path (involving the nodes in that cycle) would exist; it would always be possible to find a shorter path, by taking the cycle one more time.
>
> In general, to find the shortest paths, it suffices to consider only *acyclic* paths.

[1] This is why we call it "all pairs"; later we shall study an algorithm (Dijkstra's) which from a *specific* node find the shortest paths to all other nodes.

2 All Pairs Shortest Path: Solution

We shall employ dynamic programming to find an efficient solution to the all-pairs shortest path problem. We shall arrive at what is known as *Floyd's algorithm*, which is also known as the *Floyd-Warshall algorithm*, as well as by several other names[2].

As usual, a crucial step towards a dynamic programming algorithm is to find a suitable domain of (sub)problems. The idea of Floyd et al is to define, for each $i, j \in 1 \dots n$ and each $k \in 0 \dots n$:

> $D_k(i, j)$ is the length of a shortest path from i to j where all *intermediate* nodes belong to $1 \dots k$.

Then for each $i, j \in 1 \dots n$, the desired answer (where there are no restrictions on which nodes to pick) can be found as $D_n(i, j)$.

Basic Equations Observe that $D_0(i, j)$ is the length of the shortest path from i to j that has *no* intermediate nodes. Therefore, for all $i, j \in 1 \dots n$:

$$\begin{aligned} D_0(i,i) &= 0 && (1) \\ D_0(i,j) &= L(i,j) \text{ when there is an edge from } i \text{ to } j && (2) \\ D_0(i,j) &= \infty \text{ when there is no edge from } i \text{ to } j && (3) \end{aligned}$$

In our example, the table D_0 will be

$i \backslash j$	1	2	3	4
1	0	5	∞	∞
2	50	0	15	5
3	30	∞	0	15
4	15	∞	5	0

Recursive Equation Now let us derive an equation of D_k when $k > 0$. For an acyclic path from i to j, where all intermediate nodes are in $1 \dots k$, there are two possibilities:

1. The path does not go through node k. The shortest such path has length $D_{k-1}(i, j)$.
2. it does go through node k, but only once. The shortest such path is composed by the shortest path from i to k, with length $D_{k-1}(i, k)$, and the shortest path from k to j, with length $D_{k-1}(k, j)$.

We therefore get the equation

$$D_k(i,j) = \mathbf{min}(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)) \quad (4)$$

In our example, the table D_1 will be given by

$i \backslash j$	1	2	3	4
1	0	5	∞	∞
2	50	0	15	5
3	30	35^1	0	15
4	15	20^1	5	0

[2]You may consult Wikipedia for the historical background.

which is very much like D_0, except for the two entries with superscript 1 which reflect that when going from node 3 or 4 to node 2, it helps to go through node 1.

The table D_2 shows us that when going from node 1 to node 3 or 4, it helps to go through node 2:

$i \backslash j$	1	2	3	4
1	0	5	20^2	10^2
2	50	0	15	5
3	30	35^1	0	15
4	15	20^1	5	0

The table D_3 shows us that when going from node 2 to node 1, it helps to go through node 3:

$i \backslash j$	1	2	3	4
1	0	5	20^2	10^2
2	45^3	0	15	5
3	30	35^1	0	15
4	15	20^1	5	0

Finally, the table D_4 shows us that there are three situations (marked with superscript 4) where it helps us to be allowed to go through node 4:

$i \backslash j$	1	2	3	4
1	0	5	15^4	10^2
2	20^4	0	10^4	5
3	30	35^1	0	15
4	15	20^1	5	0

We already mentioned that the shortest path from node 1 to node 3 has length 15. This is confirmed by the table D_4 since $D_4(1,3) = 15$. Moreover, using the superscripts we can *find a shortest path* (which after all was what we were asked to do):

The entry $D_4(1,3)$ has superscript 4 so the shortest path from node 1 to node 3 will go through node 4, and thus be composed by

1. a shortest path from node 1 to node 4; since the entry $D_4(1,4)$ has superscript 2, that path will go through node 2 and thus be composed by
   (a) a shortest path from node 1 to node 2, which is just the edge since $D_4(1,2)$ has no superscript
   (b) a shortest path from node 2 to node 4, which is just the edge since $D_4(2,4)$ has no superscript
2. a shortest path from node 4 to node 3, which is just the edge since $D_4(4,3)$ has no superscript.

We conclude that the shortest path from node 1 to node 3 is:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 3$$

Implementation Assuming that we have already constructed D_0 (from the given graph), we can implement Floyd's algorithm by nested `for`-loops:

```
for k ← 1 to n
   for i ← 1 to n
      for j ← 1 to n
         v ← D_{k-1}[i, k] + D_{k-1}[k, j]          (***)
         if v < D_{k-1}[i, j]
            D_k[i, j] ← v
         else
            D_k[i, j] ← D_{k-1}[i, j]
```

Observe that we could have arranged the `for`-loops differently, for example let i go from n down to 1, but surely k should *not* go from n down to 1 as then we would refer undefined entries.

Running Time It is obvious that Floyd's algorithm runs in time $\mathbf{\Theta(n^3)}$.

Space Use Naively, we may think that we need $n + 1$ tables of size $n \times n$, and that space usage is therefore $\Theta(n^3)$. But it is easy to see that it suffices to allocate only two tables: after computing D_1, it is no longer necessary to hold D_0 and thus D_2 can be computed in the table previously occupied by D_0; after computing D_2, it is no longer necessary to hold D_1 and thus D_3 can be computed in the table previously occupied by D_1; etc.

Thus Floyd's algorithm uses space $\mathbf{\Theta(n^2)}$.

But actually, having *one* table suffices! In line (***) of the algorithm, it doesn't matter if the right hand side uses D_{k-1} or D_k, since

$$D_k(i, k) = D_{k-1}(i, k) \text{ and } D_k(k, j) = D_{k-1}(k, j)$$

which follows from the calculation (and a similar one)

$$D_k(i, k) = \min(D_{k-1}(i, k), D_{k-1}(i, k)+D_{k-1}(k, k)) = \min(D_{k-1}(i, k), D_{k-1}(i, k)+0) = D_{k-1}(i, k)$$

We have justified that Floyd's algorithm may be implemented by the code

```
for i ← 1 to n
   for j ← 1 to n
      if i = j
         D[i, i] ← 0
      else if there is an edge from i to j
         D[i, j] ← L(i, j)
      else
         D[i, j] ← ∞
for k ← 1 to n
   for i ← 1 to n
      for j ← 1 to n
         v ← D[i, k] + D[k, j]
         if v < D[i, j]
            D[i, j] ← v
```