Lecture 6

All-Pairs Shortest Paths I

Supplemental reading in CLRS: Chapter 25 intro; Section 25.2

6.1 Dynamic Programming

Like the greedy and divide-and-conquer paradigms, **dynamic programming** is an algorithmic paradigm in which one solves a problem by combining the solutions to smaller subproblems. In dynamic programming, the subproblems overlap, and the solutions to "inner" problems are stored in memory (see Figure 6.1). This avoids the work of repeatedly solving the innermost problem. Dynamic programming is often used in optimization problems (e.g., finding the maximum or minimum solution to a problem).

- The key feature that a problem must have in order to be amenable to dynamic programming is that of **optimal substructure**: the optimal solution to the problem must contain optimal solutions to subproblems.
- Greedy algorithms are similar to dynamic programming, except that in greedy algorithms, the solution to an inner subproblem does not affect the way in which that solution is augmented to the solution of the full problem. In dynamic programming the combining process is more sophisticated, and depends on the solution to the inner problem.
- In divide-and-conquer, the subproblems are disjoint.

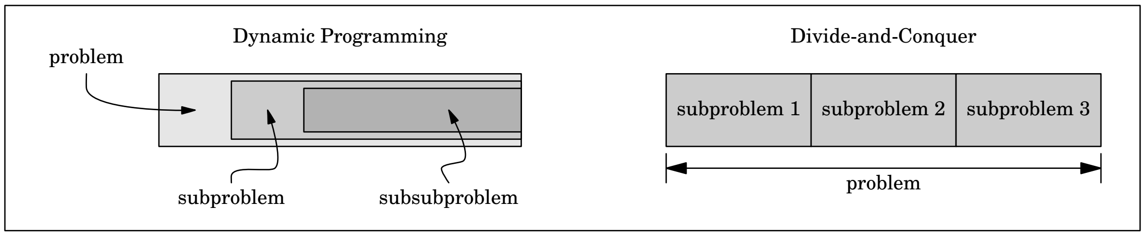

Figure 6.1. Schematic diagrams for dynammic programming and divide-and-conquer.

6.2 Shortest-Path Problems

Given a directed graph $G = (V, E, w)$ with real-valued edge weights, it is very natural to ask what the shortest (i.e., lightest) path between two vertices is.

Input: A weighted, directed graph $G = (V, E, w)$ with real-valued edge weights, a start vertex u, and an end vertex v

Output: The minimal weight of a path from u to v. That is, the value of

$$\delta(u,v) = \begin{cases} \min\left\{ w(p) : \text{ paths } u \overset{p}{\rightsquigarrow} v \right\} & \text{if such a path exists} \\ \infty & \text{otherwise,} \end{cases}$$

where the weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

(Often we will also want an example of a path which achieves this minimal weight.)

Shortest paths exhibit an optimal-substructure property:

Proposition 6.1 (Theorem 24.1 of CLRS). *Let $G = (V, E, w)$ be a weighted, directed graph with real-valued edge weights. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from v_0 to v_k. Let i, j be indices with $0 \le i \le j \le k$, and let p_{ij} be the subpath $\langle v_i, v_{i+1}, \ldots, v_j \rangle$. Then p_{ij} is a shortest path from v_i to v_j.*

Proof. We are given that

$$p : v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

is a shortest path. If there were a shorter path from v_i to v_j, say $v_i \overset{p'_{ij}}{\rightsquigarrow} v_j$, then we could patch it in to obtain a shorter path from v_0 to v_k:

$$p' : v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \ ,$$

which is impossible. □

6.2.1 All Pairs

In recitation, we saw Dijkstra's algorithm (a greedy algorithm) for finding all shortest paths from a single source, for graphs with nonnegative edge weights. In this lecture, we will solve the problem of finding the shortest paths between all pairs of vertices. This information is useful in many contexts, such as routing tables for courier services, airlines, navigation software, Internet traffic, etc.

The simplest way to solve the all-pairs shortest path problem is to run Dijkstra's algorithm $|V|$ times, once with each vertex as the source. This would take time $|V| \cdot T_{\text{DIJKSTRA}}$, which, depending on the implementation of the min-priority queue data structure, would be

Linear array:	$O\left(V^3 + VE\right) = O\left(V^3\right)$
Binary min-heap:	$O\left(VE \lg V\right)$
Fibonacci heap:	$O\left(V^2 \lg V + VE\right)$.

However, Dijkstra's algorithm only works if the edge weights are nonnegative. If we wanted to allow negative edge weights, we could instead use the slower Bellman–Ford algorithm once per vertex:

$$O\left(V^2E\right) \xrightarrow{\text{dense graph}} O\left(V^4\right).$$

We had better be able to beat this!

6.2.2 Formulating the All-Pairs Problem

When we solved the single-source shortest paths problem, the shortest paths were represented on the actual graph, which was possible because subpaths of shortest paths are shortest paths and because there was only one source. This time, since we are considering all possible source vertices at once, it is difficult to conceive of a graphical representation. Instead, we will use matrices.

Let $n = |V|$, and arbitrarily label the vertices $1, 2, \ldots, n$. We will format the output of the all-pairs algorithm as an $n \times n$ distance matrix $D = \left(d_{ij}\right)$, where

$$d_{ij} = \delta(i,j) = \begin{cases} \text{weight of a shortest path from } i \text{ to } j & \text{if a path exists} \\ \infty & \text{if no path exists,} \end{cases}$$

together with an $n \times n$ predecessor matrix $\Pi = \left(\pi_{ij}\right)$, where

$$\pi_{ij} = \begin{cases} \text{NIL}, & \text{if } i = j \text{ or there is no path from } i \text{ to } j \\ \text{predecessor of } j \text{ in our shortest path } i \rightsquigarrow j & \text{otherwise.} \end{cases}$$

Thus, the ith row of Π represents the sinlge-source shortest paths starting from vertex i, and the ith row of D gives the weights of these paths. We can also define the weight matrix $W = \left(w_{ij}\right)$, where

$$w_{ij} = \begin{cases} \text{weight of edge } (i,j) & \text{if edge exists} \\ \infty & \text{otherwise.} \end{cases}$$

Because G is a directed graph, the matrices D, Π and W are not necessarily symmetric. The existence of a path from i to j does not tell us anything about the existence of a path from j to i.

6.3 The Floyd–Warshall Algorithm

The Floyd–Warshall algorithm solves the all-pairs shortest path problem in $\Theta(V^3)$ time. It allows negative edge weights, but assumes that there are no cycles with negative total weight.[1] The Floyd–Warshall algorithm uses dynamic programming based on the following subproblem:

> What are D and Π if we require all paths to have intermediate vertices[2] taken from the set $\{1, \ldots, k\}$, rather than the full $\{1, \ldots, n\}$?

[1] If there were a negative-weight cycle, then there would exist paths of arbitrarily low weight, so some vertex pairs would have a distance of $-\infty$. In particular, the distance from a vertex to itself would not necessarily be zero, so our algorithms would need some emendations.

[2] An *intermediate vertex* in a path $p = \langle v_0, \ldots, v_n \rangle$ is one of the vertices $v_1, \ldots, v_{n-1}$, i.e., not an endpoint.

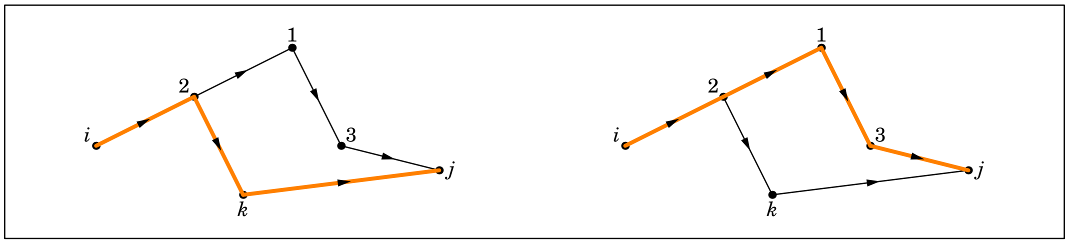

Figure 6.2. Either p passes through vertex k, or p doesn't pass through vertex k.

As k increases from 0 to n, we build up the solution to the all-pairs shortest path problem. The base case $k = 0$ (where no intermediate vertices are allowed) is easy:

$$D^{(0)} = \left(d_{ij}^{(0)}\right) \qquad \text{and} \qquad \Pi^{(0)} = \left(\pi_{ij}^{(0)}\right),$$

where

$$d_{ij}^{(0)} = \begin{Bmatrix} 0 & \text{if } i = j \\ w_{ij} & \text{otherwise} \end{Bmatrix} \qquad \text{and} \qquad \pi_{ij}^{(0)} = \begin{Bmatrix} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and there is an edge from } i \text{ to } j \end{Bmatrix}.$$

In general, let $D^{(k)}$ and $\Pi^{(k)}$ be the corresponding matrices when paths are required to have all their intermediate vertices taken from the set $\{1, \ldots, k\}$. (Thus $D = D^{(n)}$ and $\Pi = \Pi^{(n)}$.) What we need is a way of computing $D^{(k)}$ and $\Pi^{(k)}$ from $D^{(k-1)}$ and $\Pi^{(k-1)}$. The following observation provides just that.

Observation. Let p be a shortest path from i to j when we require all intermediate vertices to come from $\{1, \ldots, k\}$. Then either k is an intermediate vertex or k is not an intermediate vertex. Thus, one of the following:

1. k is not an intermediate vertex of p. Thus, p is also a shortest path from i to j when we require all intermediate vertices to come from $\{1, \ldots, k-1\}$.

2. k is an intermediate vertex for p. Thus, we can decompose p as

$$p : \; i \overset{p_{ik}}{\rightsquigarrow} k \overset{p_{kj}}{\rightsquigarrow} j \,,$$

   where p_{ik} and p_{kj} are shortest paths when we require all intermediate vertices to come from $\{1, \ldots, k-1\}$, by (a slightly modified version of) Proposition 6.1.

This tells us precisely that

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)},\, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

Thus we may compute D by the following procedure:

```
Algorithm: FLOYD–WARSHALL($W$)
1  $n \leftarrow |W.rows|$
2  $D^{(0)} \leftarrow W$
3  $\Pi^{(0)} \leftarrow$ an $n \times n$ matrix with all entries initially NIL
4  for $k \leftarrow 1$ to $n$ do
5      Let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ and $\Pi^{(k)} = \left(\pi_{ij}^{(k)}\right)$ be new $n \times n$ matrices
6      for $i \leftarrow 1$ to $n$ do
7          for $j \leftarrow 1$ to $n$ do
8              $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
9              if $d_{ij}^{(k)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ then
10                 $\pi_{ij}^{(k)} \leftarrow \pi_{ij}^{(k-1)}$
11             else
12                 $\pi_{ij}^{(k)} \leftarrow \pi_{kj}^{(k-1)}$
```

Being in a triply nested loop, lines 8–12 are executed n^3 times. Thus, the algorithm runs in $\Theta\left(n^3\right) = \Theta\left(V^3\right)$ time. Note that while the algorithm as written requires $\Theta\left(V^3\right)$ space, reasonable implementations will only require $\Theta\left(V^2\right)$ space: we only ever need to memorize four matrices at a time, namely $D^{(k)}$, $D^{(k-1)}$, $\Pi^{(k)}$, $\Pi^{(k-1)}$. In fact, we can get away with memorizing even less—see Exercise 25.2-4 of CLRS.