There are several useful algorithms related to finding shortest paths in a graph. Today, we will look at the problem of finding the shortest path between two nodes.

**Bellman-Ford Algorithm**

Our first attempt at a backtracking algorithm is:

```plaintext
1 SP(x, y, G)
2   if x = y
3       return 0
4   s <- Inf
5   for z in neighbors(x)
6       s <- min(s, d(x, z) + SP(z, y, G))
7   return s
```

But this quickly runs into trouble, since we don’t keep track of where we’ve already visited. It turns out that our problem is equivalent to a more general one, that of finding the shortest path between one node and all others. Since a shortest path can’t contain a cycle, we must have found the shortest after following \( n = |V| \) edges, so to refine our approach, we keep track of how many edges we have crossed so far:

```plaintext
1 SP (x, y, t, G)
2   if x = y
3       return 0
4   if t = 0
5       return Inf
6   s <- Inf
7   for z in neighbors(x)
8       s <- min(s, d(x, z) + SP(z, y, t-1, G))
9   return s
```

where \( t \) starts at \( n \).
Dynamic programming version

Subproblems can be identified by a pair of starting vertex (x or z) and path length t. We have two base cases (lines 2-3 and 4-5 above): First, the distance from y to itself is 0. Second, the distance from anything to y in more than t steps is Inf. This leads to the following dynamic programming algorithm:

1. SP(x, y, G)
2. N ← size(V)
3. D[v in V, 0 .. N] : Array of distances
4. for t = 0 .. N
5.   D[y,t] ← 0
6. for z in V - {y}
7.   D[z,0] ← Inf
8. for t = 1 .. N
9.   for v in V - {y}
10.  D[t,0] ← Inf
11.  for w in neighbors(v)
12.    D[t,w] ← min(D[t,w], d(w,y) + D[t-1, w])
13. return D[N,x]

Example

Here is how this algorithm works on our example above, when starting from New York:

<table>
<thead>
<tr>
<th>Time</th>
<th>SF</th>
<th>LA</th>
<th>SD</th>
<th>LV</th>
<th>D</th>
<th>NO</th>
<th>Ch</th>
<th>Atl</th>
<th>Ph</th>
<th>NY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>13</td>
<td>8</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>19</td>
<td>13</td>
<td>8</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>∞</td>
<td>24</td>
<td>19</td>
<td>13</td>
<td>13</td>
<td>8</td>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>27</td>
<td>28</td>
<td>19</td>
<td>19</td>
<td>13</td>
<td>13</td>
<td>8</td>
<td>17</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
| 6    | ...| ...| ...| ...| ...| ...| ...| ...  | ...|...

Running time

There are three loops (lines 8, 9, 11), each of which can run no more than |V| times, so the algorithm is at least O(n³). However, we also know that the loop on line 11 only executes twice for each edge, or O(m) times, so we can provide a tighter bound of O(n² + nm) = O(nm).

Floyd-Warshall

Another approach is to branch based on whether or not a node occurs along the shortest path. In other words:
FW(x, y, V[1..n])
if n == 0
    return d(x, y) // Inf, if not connected
else
    return min(
        FW(V[n], y, V[1..n-1]) + FW(x, V[n], V[1..n-1]),
        FW(x, y, V[1..n-1]))

The dynamic programming version of this is:

FW(x, y, G = (V,E))
n <- size(V)
SP[u in V, v in V, 1..n] : Array of distances
SP[u,v,0] <- d(u,v)
for i = 1 .. n
    for u in V
        for v in V
            SP[u, v, i] <- min(
                SP[u, V[i], i-1] + SP[V[i], v, i-1],
                SP[u, v, i-1])
return SP[x,y,n]

Note that here, although the algorithm is always $O(n^3)$ because of the 3 loops on lines 5-7, we are actually computing the shortest paths between all pairs of nodes.