Shortest Paths

Let $G = (V, E)$ be a directed graph, and let $w_{u,v} \geq 0$ be the weight of edge $(u,v)$. Let $V = \{1, \ldots, n\}$.

Idea for recursive solution to find shortest path between $s$ and $t$, $s, t \in V$: the shortest path between $s$ and $t$ is

- either the shortest of the paths from $s$ to $t$ not passing through $n$
- or the shortest path from $s$ to $n$ concatenated with the shortest path between $n$ and $t$.

Recursive Version

**Algorithm SP-REC**

```python
SP-REC(s, t; V; E; w)
if |V| = 0 then return $w_{s,t}$
else
    k := |V|
    l := SP(s, t; V - {k}; E, w)
    l' := SP(s, k; V - {k}; E, w) + SP(k, t; V - {k}; E, w)
    return min(l, l')
```

Algorithm $SP(s, t; V, E, w)$ finds the length of the shortest of the paths between $s$ and $t$, among those that can pass through the vertices of $V$. (If $(u, v) \notin E$, then assume $w_{u,v} = \infty$ in the implementation.)

Iterative Version — General Idea

Define a $n \times n \times (n + 1)$ matrix $M$.

Design an algorithm that will fill the matrix so that $M[s, t, k]$ is the length of the shortest of the paths between $s$ and $t$, among those that can pass through the vertices $\{1, \ldots, k\}$.

Once the matrix is filled, the length of the shortest path from $s$ to $t$ is in $M[s, t, n]$.

So, once the matrix is filled, we have the length of the shortest paths between any two vertices.

Filling the Matrix

We know what is $M[s, t, 0]$; this is just the length of the edge from $s$ and $t$, if it exists. (Otherwise we set it to $\infty$ by convention.)

The value of $M[s, t, k]$ can then be computed based on precomputed values of $M[\cdot, \cdot, k - 1]$ since

$$M[s, t, k] = \min\{M[s, t, k - 1], M[s, k, k - 1] + M[k, t, k - 1]\}$$

Algorithm

**Algorithm SP-DP**

```python
SP-DP(V, E, w)
create $n \times n \times (n + 1)$ matrix $M$
for u := 1 to n
    for v := 1 to n
        if $(u, v) \in E$ then $M[u, v, 0] := w_{u,v}$
        else $M[u, v, 0] = \infty$
    for k := 1 to n
        for u := 1 to n
            for v := 1 to n
                $M[u, v, k] := \min\{ M[u, v, k - 1], M[u, k, k - 1] + M[k, v, k - 1]\}$
```

Runs in time $O(n^3)$. 
Reconstructing the actual paths

Together with $M$, we also create a matrix $P$ of same dimension, such that $P[u, v, k]$ says which vertex comes before $v$ in the shortest path from $u$ to $v$ among those that can pass through $\{1, \ldots, k\}$.

Once we have the final $M$ and $P$, we can reconstruct the shortest path from $u$ to $v$ backwards. Start from $v$, and look up $v' = P[u, v, n]$. Then look up $v'' = P[u, v', n]$ and so one. Eventually we get to $u$, by way of a shortest path.

Algorithm that also computes $P$

$$\text{SP-DP-2}(V, E, w)$$
create $n \times n \times (n + 1)$ matrices $M$ and $P$
for $u := 1$ to $n$
for $v := 1$ to $n$
if $(u, v) \in E$ then $M[u, v, 0] := w_{u,v}; P[u, v, 0] = u$
else $M[u, v, 0] = \infty; P[u, v, 0] = u$
for $k := 1$ to $n$
for $u := 1$ to $n$
for $v := 1$ to $n$
if $l_1 = M[u, v, k - 1]; l_2 = M[u, k, k - 1] + M[k, v, k - 1]$
if $l_1 \leq l_2$ then
$M[u, v, k] := M[u, v, k - 1]; P[u, v, k] := P[u, v, k - 1]$
else
$M[u, v, k] := M[u, k, k - 1] + M[k, v, k - 1]; P[u, v, k] = k$

Knapsack

You are packing your knapsack for a long hiking. You have a choice of $n$ items to put in. Item $i$ has volume $v_i$ and is worth an advantage $c_i$ if you can take it with you.

The knapsack has volume $B$.

Choose a subset $S$ of elements that fit into the knapsack and maximize $\sum_{i \in S} c_i$.

Dynamic programming algorithm

Construct a $n \times (B + 1)$ table $M[\cdot, \cdot]$.

For every $1 \leq k \leq n$ and $0 \leq B' \leq B$,
\[ M[k, B'] \] contains the cost of an optimum solution for the instance that uses only a subset of the first $k$ elements (of volume $v_1, \ldots, v_k$ and cost $c_1, \ldots, c_k$) and with volume $B'$.

- $M[1, B'] = c_1$ if $B' \geq v_1$; $M[1, B'] = 0$ otherwise.
- for every $k, B' \geq 1$,
\[ M[k, B'] = \max\{M[k - 1, B'], M[k - 1, B' - v_k] + c_k\} \]

Formal description of the problem

Given: $n$ items of “cost” $c_1, c_2, \ldots, c_n$ (positive integers), and of “volume” $v_1, v_2, \ldots, v_n$ (positive integers); a volume value $B$ (for bound).

Find a subset $S$ of the items such that
\[ \sum_{i \in S} v_i \leq B \]
and such that the total cost $\sum_{i \in S} c_i$ is maximized.

Second matrix

The cost of an optimum solution is reported in $M[n, B]$.

To construct an optimum solution, we also build a Boolean matrix $C[\cdot, \cdot]$ of the same size.

For every $k$ and $B'$, $C[k, B'] = \text{True}$ iff there is an optimum solution that packs a subset of the first $k$ items in volume $B'$ so that item $k$ is included in the solution.
Example

Consider an instance with 9 items and a bag of size 15. The costs and the volumes of the items are as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Volume</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

Solution

Optimum: 23
Item 9 Cost 8 Volume 5
Item 7 Cost 7 Volume 2
Item 5 Cost 4 Volume 3
Item 4 Cost 4 Volume 4

Approximate string matching

When you run a spell checker on a text, and it finds a word not in the dictionary, it normally proposes a choice of possible corrections.

If it finds stell it will suggest:

- tell, swell, stull, still, steel, steal, stall, spell, smell, shell, and sell.

Edit distance

How do you decide the “closeness” between two strings?

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of “errors” (edit operations) needed to transform $x$ into $y$.

Possible operations are:

- **insert a character.**
  
  \[ \text{insert}(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n. \]

- **delete a character.**
  
  \[ \text{delete}(x, i) = x_1 x_2 \cdots x_{i-1} x_{i+1} \cdots x_n. \]

- **modify a character.**
  
  \[ \text{modify}(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n. \]
Example

$x = aabab$ and $y = babb$.

One possibility

\begin{align*}
  a & \quad a & \quad b & \quad a & \quad b & \quad x \\
  b & \quad a & \quad b & \quad a & \quad b & \quad x' = \text{insert}(x,0,b) \\
  b & \quad a & \quad b & \quad a & \quad b & \quad x'' = \text{delete}(x',2) \\
  b & \quad a & \quad b & \quad b & \quad y = \text{delete}(x'',4)
\end{align*}

And also

\begin{align*}
  a & \quad a & \quad b & \quad a & \quad b & \quad x \quad = \text{delete}(x,1) \\
  b & \quad a & \quad b & \quad x' = \text{delete}(x',1) \\
  b & \quad a & \quad b & \quad b & \quad y = \text{insert}(x'',3,b)
\end{align*}

Can you do better?

Computing edit distance

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

- **put** $y_m$ at the end: $x \rightarrow x_1 \cdots x_n y_m$ and then transform $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- **delete** $x_n$: $x \rightarrow x_1 \cdots x_{n-1}$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- **change** $x_n$ into $y_m$ (if they are different): $x \rightarrow x_1 \cdots x_{n-1} y_m$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

Dynamic programming table

Given strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$.

Define $(n + 1) \times (m + 1)$ matrix $M[i,j]$.

For every $0 \leq i \leq n$ and $0 \leq j \leq m$, $M[i,j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Recursive definition of the table

- $M[0,j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the $j$ characters $y_1, \ldots, y_j$.
- $M[i,0] = i$ for similar reasons.

- For $i, j \geq 1$,

\[
M[i,j] = \min\{M[i-1,j] + 1, \quad M[i,j-1], \quad M[i-1,j-1] + \text{change}(x_i, y_j)\}
\]

where $\text{change}(x_i, y_j) = 1$ if $x_i \neq y_j$ and $\text{change}(x_i, y_j) = 0$ o/w.
Consider again \( x = aabab \) and \( y = babb \).

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>( a )</th>
<th>( b )</th>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( a )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( b )</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( a )</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( b )</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( a )</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Optimum solution?

---

### Longest common subsequence

A subsequence of a string is obtained by taking a string and possibly deleting elements. If \( x_1 \cdots x_n \) is a string and \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \) is a strictly increasing sequence of indices, then \( x_{i_1}x_{i_2}\cdots x_{i_k} \) is a subsequence of \( x \).

E.g. art is a subsequence of algorithm.

Given strings \( x \) and \( y \) we want to find the longest string that is a subsequence of both.

E.g. art is the longest common subsequence of algorithm and parachute.

---

### Definition of the matrix

For every \( 0 \leq i \leq n \) and \( 0 \leq j \leq m \), \( M[i,j] \) contains the length of the l.c.s. between \( x_1 \cdots x_i \) and \( y_1 \cdots y_j \).

- \( M[i,0] = 0 \)
- \( M[0,j] = 0 \)
- and

\[
M[i,j] = \max \{ M[i-1,j], M[i,j-1] + \text{eq}(x_i, y_j) \}
\]

where \( \text{eq}(x_i, y_j) = 1 \) if \( x_i = y_j \), \( \text{eq}(x_i, y_j) = 0 \) a/w.

---

### Example

Let's consider again \( x = aabab \) and \( y = babb \).

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>( a )</th>
<th>( b )</th>
<th>( a )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( a )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( b )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( c )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The length of the l.c.s. of \( x = x_1 \cdots x_n \) and \( y = y_1 \cdots y_m \) is either

- The length of the l.c.s. of \( x_1 \cdots x_{n-1} \) and \( y_1 \cdots y_m \) or;
- The length of the l.c.s. of \( x_1 \cdots x_n \) and \( y_1 \cdots y_{m-1} \) or;
- \( 1 + \) the length of the l.c.s. of \( x_1 \cdots x_{n-1} \) and \( y_1 \cdots y_{m-1} \), if \( x_n = y_m \).

---

### Reducing to a smaller subproblem

The table has \( \Theta(nm) \) entries, each one computable in constant time.

One can construct an auxiliary table \( Op[i,j] \) such that \( Op[i,j] \) specifies what is the first operation to do in order to optimally transform \( x_1 \cdots x_i \) into \( y_1 \cdots y_j \).

---

### Algorithm and running time

The table has \( \Theta(nm) \) entries, each one computable in constant time.