WS 2021/22

# Algorithms

# Chapter 6
# Dynamic Programming

Martin Dietzfelbinger

February 2022

# Chapter 6: Dynamic Programming (DP)

Algorithm paradigm for optimization problems. Typical:

- Define (many) **“subproblems”** (of an input instance)
- Identify simple **base cases**
- Formulate a version of the property

**Substructures of optimal structures are optimal**

- Find recursion equations for **values** of optimal solutions:

**Bellman’s optimality equations**

- Calculate optimal values (and structures) iteratively

## 6.3 Edit distance

**Problem statement:** Let $A$ be alphabet. (*Example*: Latin alphabet, ASCII alphabet, {A,C,G,T}.) Then $A^*$ denotes the set of all strings (or words) over $A$.

If $x = a_1 \dots a_m \in A^*$ and $y = b_1 \dots b_n \in A^*$ are two strings over $A$, we ask how **similar** (or different) they are.

"`Arbeit`" (*work*) and "`Freizeit`" (*leisure time*) are not identical, but intuitively more similar than "`informatics`" and "`camping`".

How to measure "being more or less similar"?

We define elementary steps ("edit operations"), which change a string:

- Delete a letter: From $uav$ obtain $uv$ ($u, v \in A^*$, $a \in A$). *Example*: `beat` → `bet`.
- Insert a letter: From $uv$ obtain $uav$ ($u, v \in A^*$, $a \in A$). *Example*: `bet` → `beat`.
- Replace a letter: From $uav$ obtain $ubv$ ($u, v \in A^*$, $a, b \in A$).
  *Example*: `beat` → `best`.

The "distance" or **edit distance $d(x,y)$** of $x$ and $y$ is the minimal number of edit operations one needs to transform $x$ into $y$.

Obvious: $d(x,y) = d(y,x)$. (Insertion and deletion are inverses, replacement is self-inverse.)

**Observe:** Transforming is equivalent to aligning:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | r | b | e | i | - | - | - | t | | | |
| F | r | - | e | i | z | e | i | t | | | |
| i | n | f | o | r | m | a | t | - | i | c | s |
| c | - | - | - | - | - | a | m | p | i | n | g |

One aligns strings with some extra dashes below each other, where the two lines without dashes give $x$ and $y$. The **cost** of such an alignment is the number of positions where letters in the two lines differ.

*Claim*: The cost of such an alignment is the same as the number of edit operations in a sequence of operations that transforms one word into the other. (In the examples: $5$ and $10$, resp.) Hence: The **minimal** cost of an alignment is exactly the edit distance. (In the examples: $4$ (!) and $10$, resp.)

The problem "edit distance" is the following:

**Input:** Strings $x = a_1 \ldots a_m$, $y = b_1 \ldots b_n$ from $A^*$.

**Task:** Calculate $d(x, y)$ (and a sequence of edit operations resp. an alignment that transforms $x$ into $y$ of cost $d(x, y)$).

**Strategy: Dynamic programming**

Our example: $x = \texttt{exponentiell}$ und $y = \texttt{polynomiell}$.

Step 1: Identify relevant **subproblems**.

Consider prefixes $x[1..i] = a_1 \ldots a_i$ and $y[1..j] = b_1 \ldots b_j$, and let

$$\boldsymbol{E(i,j)} := d(x[1..i], y[1..j]), \text{ for } 0 \le i \le m,\ 0 \le j \le n$$

In the example: $\boldsymbol{E(7,6)}$ means we are to calculate $d(\texttt{exponen}, \texttt{polyno})$ (because $x[1..7] = \texttt{exponen}$ und $y[1..6] = \texttt{polyno}$).

In order to obtain Bellman equations, we consider relations among subproblems. For this we utilize the idea of aligning words as on slide 3.

Consider (nonempty) prefixes $x[1..i] = a_1 \dots a_i$ and $y[1..j] = b_1 \dots b_j$, for $1 \le i \le m$, $1 \le j \le n$.

If $x[1..i]$ and $y[1..j]$ are aligned optimally, with cost $E(i,j)$, there are three possibilities for the last position in the alignment:

| Case 1 (copy/replace) | Case 2 (delete) | Case 3 (insert) |
|---|---|---|
| $x[1..i-1]\ a_i$ | $x[1..i-1]\ a_i$ | $x[1..i]$ - |
| $y[1..j-1]\ b_j$ | $y[1..j]$ - | $y[1..j-1]\ b_j$ |

| Case 1 (copy/replace) | Case 2 (delete) | Case 3 (insert) |
|---|---|---|
| $x[1..i-1]$ $a_i$ | $x[1..i-1]$ $a_i$ | $x[1..i]$ - |
| $y[1..j-1]$ $b_j$ | $y[1..j]$ - | $y[1..j-1]$ $b_j$ |

Case 1: There is a replacement in the last position, with cost

$$\text{diff}(a_i, b_j) := [a_i \neq b_j] = \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{if } a_i = b_j \end{cases}.$$

$x[1..i-1]$ und $y[1..j-1]$ must be optimally aligned ("optimal substructure").

$$\Rightarrow \quad E(i,j) = E(i-1, j-1) + \text{diff}(a_i, b_j).$$

Case 2: There is a deletion in the last position, with cost 1.
$x[1..i-1]$ and $y[1..j]$ must be optimally aligned.

$$\Rightarrow \quad E(i,j) = E(i-1, j) + 1.$$

Case 3: There is an insertion in the last position, with cost 1.
$x[1..i]$ and $y[1..j-1]$ must be optimally aligned.

$$\Rightarrow \quad E(i,j) = E(i, j-1) + 1.$$

Among the three cases the one/s with the smallest cost is/are relevant.
Altogether, we obtain:

**Bellman optimality equations** for edit distance:

$$E(i,j) = \min\{E(i-1,j-1) + \mathsf{diff}(a_i,b_j), E(i-1,j)+1, E(i,j-1)+1\}.$$

**Base cases:** Empty prefixes $\varepsilon = x[1..0]$ and $\varepsilon = y[1..0]$.

Obvious: $E(i,0) = d(x[1..i],\varepsilon) = i$, for $0 \le i \le m$, and
$E(0,j) = d(\varepsilon, y[1..j]) = j$, for $0 \le j \le n$.

(One has to delete/insert all letters.)

The numbers $E(i,j)$ are computed iteratively, entering them into a matrix $\texttt{E}[0..m,0..n]$. This is already the "dynamic programming algorithm" for edit distance.

**Initialization:**

$\texttt{E}[i,0] \leftarrow i$, for $i = 0, \ldots, m$;
$\texttt{E}[0,j] \leftarrow j$, for $j = 0, \ldots, n$.

Then we fill in the matrix (i.,e.) row by row, following the Bellman equations:

**for** i **from** $1$ **to** $m$ **do**
  **for** j **from** $1$ **to** $n$ **do**
    $\texttt{E[i,j]} \leftarrow \min\{\texttt{E[i-1,j-1]} + \text{diff}(a_{\texttt{i}}, b_{\texttt{j}}), \texttt{E[i-1,j]} + 1, \texttt{E[i,j-1]} + 1\}$;
**return** $\texttt{E}[m,n]$.

**Time:** $O(m \cdot n)$.

**Remark:** For doing the calculation by hand it saves time to identify the pairs $(i,j)$ with $\text{diff}(a_i, b_j) = 0$, or $a_i = b_j$, beforehand (red, underlined).

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | | | | | | | | | — | | |
| x | 2 | | | | | | | | | | | |
| p | 3 | — | | | | | | | | | | |
| o | 4 | | — | | | | — | | | | | |
| n | 5 | | | | | — | | | | | | |
| e | 6 | | | | | | | | | — | | |
| n | 7 | | | | | — | | | | | | |
| t | 8 | | | | | | | | | | | |
| i | 9 | | | | | | | | — | | | |
| e | 10 | | | | | | | | | — | | |
| l | 11 | | | — | | | | | | | — | — |
| l | 12 | | | — | | | | | | | — | — |

Base values $E(i,0)$ and $E(0,j)$, markers for $\text{diff}(a_i, b_j) = 0$.

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **8** | 9 | 10 |
| x | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 10 |
| p | 3 | **2** | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 |
| o | 4 | 3 | | | | | | | | | | |
| n | 5 | | | | | | | | | | | |
| e | 6 | | | | | | | | | | | |
| n | 7 | | | | | | | | | | | |
| t | 8 | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | |
| e | 10 | | | | | | | | | | | |
| l | 11 | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | |

Filled in row by row up to $E(4, 1)$.

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **8** | 9 | 10 |
| x | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 10 |
| p | 3 | **2** | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 |
| o | 4 | 3 | **2** | | | | | | | | | |
| n | 5 | | | | | | | | | | | |
| e | 6 | | | | | | | | | | | |
| n | 7 | | | | | | | | | | | |
| t | 8 | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | |
| e | 10 | | | | | | | | | | | |
| l | 11 | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | |

$E(4,2) = \min\{2 + \underline{0}, 3 + 1, 3 + 1\} = 2.$

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **8** | 9 | 10 |
| x | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 10 |
| p | 3 | **2** | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 |
| o | 4 | 3 | **2** | 3 | | | | | | | | |
| n | 5 | | | | | | | | | | | |
| e | 6 | | | | | | | | | | | |
| n | 7 | | | | | | | | | | | |
| t | 8 | | | | | | | | | | | |
| i | 9 | | | | | | | | | | | |
| e | 10 | | | | | | | | | | | |
| l | 11 | | | | | | | | | | | |
| l | 12 | | | | | | | | | | | |

$E(4,3) = \min\{3+1, 3+1, 2+1\} = 3.$

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **8** | 9 | 10 |
| x | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 10 |
| p | 3 | **2** | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 |
| o | 4 | 3 | **2** | 3 | 4 | 5 | **5** | 6 | 7 | 8 | 9 | 10 |
| n | 5 | 4 | 3 | 3 | 4 | **4** | 5 | 6 | 7 | 8 | 9 | 10 |
| e | 6 | 5 | 4 | 4 | 4 | 5 | 5 | 6 | 7 | **7** | 8 | 9 |
| n | 7 | 6 | 5 | 5 | 5 | **4** | 5 | 6 | 7 | 8 | 8 | 9 |
| t | 8 | 7 | 6 | 6 | 6 | 5 | 5 | 6 | 7 | 8 | 9 | 9 |
| i | 9 | 8 | 7 | 7 | 7 | 6 | 6 | 6 | **6** | 7 | 8 | 9 |
| e | 10 | 9 | 8 | 8 | 8 | 7 | 7 | 7 | 7 | **6** | 7 | 8 |
| l | 11 | 10 | 9 | **8** | 9 | 8 | 8 | 8 | 8 | 7 | **6** | **7** |
| l | 12 | 11 | 10 | **9** | 9 | 9 | 9 | 9 | 9 | 8 | **7** | **6** |

Matrix is filled completely. Result: $d(x, y) = E(12, 11) = 6$, lower right corner!

## Discovering an optimal edit sequence in time $O(m+n)$

Given $x$ and $y$, one wants not only $d(x,y)$, but also the sequence of operations that transforms $x$ into $y$ in $d(x,y)$ steps.
If we have the matrix $\texttt{E}[0..m, 0..n]$, this is easy:

We walk a path from $(m,n)$ to $(0,0)$. From $(i,j)$ we go to

$$\begin{array}{ll} (i-1,j-1), & \text{if } E(i,j) = E(i-1,j-1) + \text{diff}(a_i,b_j), \\ (i-1,j), & \text{if } E(i,j) = E(i-1,j) + 1, \\ (i,j-1), & \text{if } E(i,j) = E(i,j-1) + 1. \end{array}$$

If more than one of the equations hold, we can choose arbitrarily.
The path we find ends in $(0,0)$. This path, read backwards, from $(0,0)$ to $(m,n)$,represents a transformation of $x$ into $y$, where letters are left alone, replaced, inserted, or deleted. One reads $x = x[1..m]$ from left to right and generates $y[1..n]$ from left to right.

(1) $(i-1,j-1) \stackrel{b}{\to} (i,j)$ corresponds to copying $a_i$ ($b=0$) or replacing it by $b_j$ ($b=1$);

(2) $(i-1,j) \stackrel{1}{\to} (i,j)$ corresponds to deleting $a_i$;

(3) $(i,j-1) \stackrel{1}{\to} (i,j)$ corresponds to inserting $b_j$.

| $E$ | | p | o | l | y | n | o | m | i | e | l | l |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| e | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 8 | 9 | 10 |
| x | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 9 | 10 |
| p | 3 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 |
| o | 4 | 3 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | 9 | 10 |
| n | 5 | 4 | 3 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| e | 6 | 5 | 4 | 4 | 4 | 5 | 5 | 6 | 7 | 7 | 8 | 9 |
| n | 7 | 6 | 5 | 5 | 5 | 4 | 5 | 6 | 7 | 8 | 8 | 9 |
| t | 8 | 7 | 6 | 6 | 6 | 5 | 5 | 6 | 7 | 8 | 9 | 9 |
| i | 9 | 8 | 7 | 7 | 7 | 6 | 6 | 6 | 6 | 7 | 8 | 9 |
| e | 10 | 9 | 8 | 8 | 8 | 7 | 7 | 7 | 7 | 6 | 7 | 8 |
| l | 11 | 10 | 9 | 8 | 9 | 8 | 8 | 8 | 8 | 7 | 6 | 7 |
| l | 12 | 11 | 10 | 9 | 9 | 9 | 9 | 9 | 9 | 8 | 7 | 6 |

In the example: $(0,0) \xrightarrow{1} (1,0) \xrightarrow{1} (2,0) \xrightarrow{0} (3,1) \xrightarrow{0} (4,2) \xrightarrow{1} (5,3) \xrightarrow{1} (6,4) \xrightarrow{0} (7,5) \xrightarrow{1} (8,6) \xrightarrow{1} (8,7) \xrightarrow{0} (9,8) \xrightarrow{0} (10,9) \xrightarrow{0} (11,10) \xrightarrow{0} (12,11)$, cost 6.

This yields the following alignment/transformation, where the six "chargeable" positions are marked in red:

```
e x p o n e n t - i e l l
-------------------------
- - p o l y n o m i e l l
```

One more example:

$x =$ `speziell` (*special*) and $y =$ `beliebig` (*arbitrary*).

Filling in the matrix gives:

| $E$ | | b | e | l | i | e | b | i | g |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| s | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| p | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| e | 3 | 3 | **2** | 3 | 4 | **4** | 5 | 6 | 7 |
| z | 4 | 4 | 3 | 3 | 4 | 5 | 5 | 6 | 7 |
| i | 5 | 5 | 4 | 4 | **3** | 4 | 5 | **5** | 6 |
| e | 6 | 6 | **5** | 5 | 4 | **3** | 4 | 5 | 6 |
| l | 7 | 7 | 6 | **5** | 5 | 4 | 4 | 5 | 6 |
| l | 8 | 8 | 7 | **6** | 6 | 5 | 5 | 5 | 6 |

Red, underlined: the positions $(i, j)$ with $a_i = b_j$.

Retracing yields the following sequence of operations (one possibility among others):

$(0,0) \to (1,1) \to (2,1) \to (3,2) \to (4,3) \to (5,4) \to (6,5) \to (6,6) \to (7,7) \to (8,8)$.

This yields the following alignment:

$$\frac{\texttt{s p e z i e - l l}}{\texttt{b - e l i e b i g}}$$

There are other alignments with cost 6, e.g.

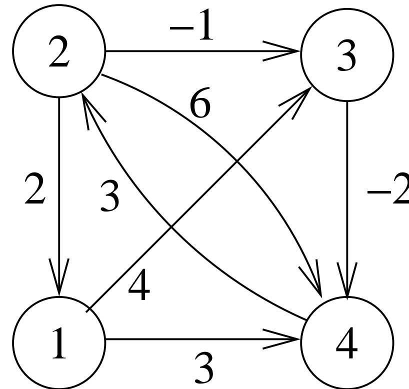
$$\frac{\texttt{s p e z i e l l -}}{\texttt{- b e l i e b i g}} \quad .$$

## 6.6 The All-Pairs-Shortest-Paths problem

Central example for dynamic programming: The "**APSP problem**".

**"shortest paths between all nodes".**

**Input:**




Directed graph $G = (V, E, c)$
with $V = \{1, \ldots, n\}$ and $E \subseteq \{(v, w) \mid 1 \leq v, w \leq n, v \neq w\}$.

$c\colon E \to \mathbb{R} \cup \{+\infty\}$: **Weight/cost/length function**.

**Cost/Length** of a walk $p = (v = v_0, v_1, \ldots, v_r = w)$ is

$$c(p) = \sum_{1 \le s \le r} c(v_{s-1}, v_s).$$

**Task:** For all $(v, w)$, $1 \le v, w \le n$, find a walk from $v$ to $w$ with minimal cost/length $c(p)$ ("shortest path").

Here: **Floyd-Warshall algorithm**.

We demand:
No **cycles** have negative total weight:

**(∗)** $(v = v_0, v_1, \ldots, v_r = v)$ cycle $\Rightarrow \sum_{1 \le s \le r} c(v_{s-1}, v_s) \ge 0.$

Reason: If there is a negative cycle from $v$ to $v$, and some walk from $v$ to $w$, then there are walks from $v$ to $w$ with negative length of arbitrarily large absolute value – the question for a "shortest path" does not make sense.

**Consequences:** (1) If $p$ is a walk from $v$ to $w$, then there is a (simple) path $p'$ from $v$ to $w$ with $c(p') \leq c(p)$.

**Reason:** Cut out $(u, \ldots, u)$-segments from $p$, repeatedly, until no repetition remains. The walk cannot get longer by this operation.)

(2) If there is a walk from $v$ to $w$ at all, then there also is one with minimal length (a **“shortest path”**).

**Reason:** Because of (1) it suffices to consider simple paths (with at most $n-1$ edges); of these there are only finitely many.

But: There may be several “shortest paths” from $v$ to $w$. (Simple paths and maybe walks with cycles of length 0.)

First: Determine **length $S(v,w)$** of a shortest path from $v$ to $w$, for all $v, w \in V$. ($\infty$, if no path exists.)

**Output:** matrix $S = (S(v,w))_{1 \leq v,w \leq n}$.

W. l. o. g.: $E = V \times V - \{(v,v) \mid v \in V\}$

Edges $(v,w)$ that are not present $(v,w)$ are represented by $c(v,w) = \infty$.

**First step**: Identify suitable **subproblems**.

Given $k$, $0 \le k \le n$, consider paths from $v$ to $w$ that **in between** (excepting start and end node) visit only nodes in $\{1,\ldots,k\}$.

$k = 0$: Cannot visit any node on the way;
only edges $(v,w)$ with $c(v,w) < \infty$, or paths $(v,v)$ of length $0$.

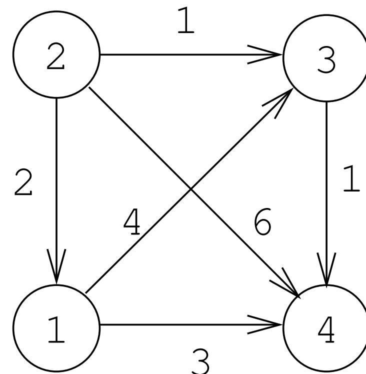
$k = 1$: Possible paths: $(v,w)$ ($(v)$ if $v = w$) and $(v,1,w)$.

$k = 2$: Possible paths:$(v,w)$ ($(v)$ if $v = w$), $(v,1,w)$, $(v,2,w)$, $(v,1,2,w)$, and $(v,2,1,w)$.

Etc.

$\boldsymbol{S(v,w,k)}$ := length of a shortest path from $v$ to $w$ using only nodes in $\{1,\ldots,k\}$ underway.
( $=\infty$, if there is no such path.)

*Example*:

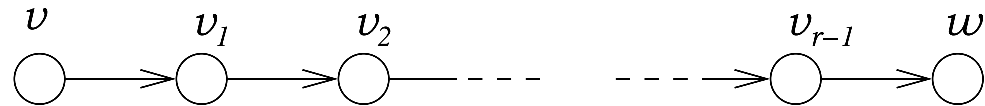



$S(2,4,0) = 6$
$S(2,4,1) = 5$
$S(2,4,2) = 5$
$S(2,4,3) = 2$

monotonically decreasing

"Bellman equations":

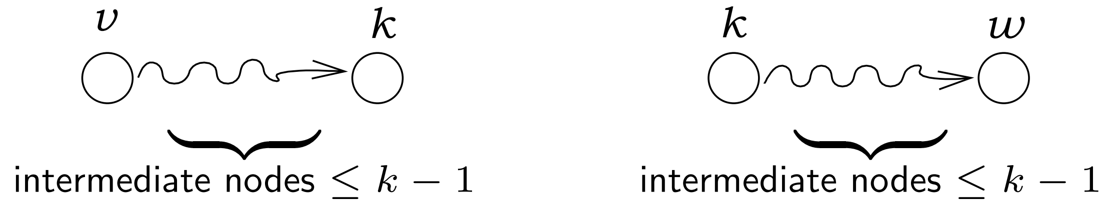
Consider path $p = (v_0, \ldots, v_r)$ from $v$ to $w$; intermediate nodes $v_1, \ldots, v_{r-1}$ are from $\{1, \ldots, k\}$:




Assume this path is optimal. Node $k$ appears once or not at all.

1) If $k$ is not on the path $p$,
then $p$ is optimal for intermediate nodes $1, \ldots, k-1$.

2) If $k$ is not on the path $p$, then $p$ has two parts




both optimal with respect to $\{1, \ldots, k-1\}$ (otherwise one could replace the part by something cheaper, contradicting the optimality of $p$).

1) and 2) express that substructures (subpaths) of an optimal solution (shortest path) have to be optimal.

**Bellman optimality equations** for the Floyd-Warshall algorithm:

$$S(v,w,k) = \min\{S(v,w,k-1), S(v,k,k-1) + S(k,w,k-1)\},$$
$$\text{for } 1 \le v,w \le n, 1 \le k \le n.$$

**Base cases:** $S(v,v,0) = 0.$
$S(v,w,0) = c(v,w)$, for $v \ne w$.

We describe an **iterative** algorithm.
Intermediate results are stored in an array $\texttt{S}[1..n,1..n,0..n]$, initialized by

$$\texttt{S}[v,w,0] \leftarrow c(v,w), \quad 1 \le v,w \le n, \quad v \ne w;$$
$$\texttt{S}[v,v,0] \leftarrow 0, \quad 1 \le v \le n.$$

The algorithm fills $\texttt{S}$ according to growing $k$:

```
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w,k] ← min{S[v,w,k-1], S[v,k,k-1] + S[k,w,k-1]}.
```

**Correctness:** Follows from previous considerations:

The algorithms calculates all values $S(v, w, k)$ according to the Bellman equations.

**Running time:** There are three nested loops: $\Theta(n^3)$.

We are wasting space since for the $k$-th execution of the loop we only need the $(k-1)$-components of `S`.

Easy: Need only two matrices. An "old" one ($(k-1)$-version) and a "new" one ($k$-version), between which we can switch back and forth.

Then space is $O(n^2)$, time is $O(n^3)$.

Further improvement: We have:

$$(*) \qquad \begin{array}{l} S(v,k,k) = S(v,k,k-1) \text{ and} \\ S(k,w,k) = S(k,w,k-1), \text{ für } 1 \le v,w,k \le n, \end{array}$$

since node $k$ cannot occur in the interior of a path from $v$ to $k$ or from $k$ to $w$.

So one does not need an “old” and a “new” array, but only one: S[$1$..$n$,$1$..$n$].

```
S[v,v] ← 0;
S[v,w] ← c(v,w), für 1 ≤ v,w ≤ n, v ≠ w;
for k from 1 to n do
  for v from 1 to n do
    for w from 1 to n do
      S[v,w] ← min{S[v,w], S[v,k] + S[k,w]}.
```

We do not only want to know the **cost** of the shortest paths, but we also want to be able to calculate such a path for given $v, w$.

For this: Auxiliary array `I[`$1..n$`,`$1..n$`]`, in which throughout rounds $k = 0, 1, \ldots, n$ for each pair $(v, w)$ the following information is kept:
What is the node with the largest number that has been used for a path from $v$ to $w$ of length $S(v, w, k)$?

Adding this feature gives the complete Floyd-Warshall algorithm.

**Algorithm Floyd-Warshall**(C[1..$n$,1..$n$])

**Input:** C[1..$n$,1..$n$]: matrix of edge costs/lengths

**Output:** S[1..$n$,1..$n$], with $S(v,w)$ the cost of a shortest path from $v$ to $w$

I[1..$n$,1..$n$]: minimal maximal nodes on shortest $(v,w)$ path

```
(1)   for v from 1 to n do
(2)     for w from 1 to n do
(3)       if v = w then S[v,w] ← 0; I[v,w] ← 0
(4)         else S[v,w] ← C[v,w];
(5)       if S[v,w] < ∞ then I[v,w] ← 0 else I[v,w] ← −1;
(6)   for k from 1 to n do
(7)     for v from 1 to n do
(8)       for w from 1 to n do
(9)         if S[v,k] + S[k,w] < S[v,w] then
(10)          S[v,w] ← S[v,k] + S[k,w]; I[v,w] ← k;
(11)  return S[1..n,1..n], I[1..n,1..n].
```

**Correctness:** Loop invariant:
After the $k$-th execution of the loop we have $\mathtt{S}[v,w] = S(v,w,k)$.

(Proof by induction on $k$, using the Bellman equations and $(*)$.)

And: $\mathtt{I}[v,w]$ is the smallest $\ell$ such that among the

**shortest paths from $v$ to $w$ over only nodes in $\{1,\ldots,k\}$**

there is one with **maximal** node $\ell$.

(Proof by induction on $k$.)

Then a shortest path from $v$ to $w$ can be printed using a simple recursive procedure "`printPathInner`", on the basis of $\mathtt{I}[1..n,1..n]$.

**Algorithm printPathInner**$(v,\ w)$
**Global:** $\texttt{I}[1..n,1..n]$: matrix of minimal maximal nodes on shortest paths
**Input:** $v, w \in V$
**Output:** over print function

(1) **if** $v \neq w$ **then**
(2) $\quad$ $\texttt{k} \leftarrow \texttt{I}[v,w]$;
(3) $\quad$ **if** $\texttt{k} > 0$ **then**
(4) $\quad\quad$ **printPathInner**$(u, \texttt{k})$; **print**$(\texttt{k})$; **printPathInner**$(\texttt{k}, w)$;

**Algorithm printPath**$(v,\ w)$
**Global:** $\texttt{I}[1..n,1..n]$: matrix of minimal maximal nodes on shortest paths
**Input:** $v, w \in V$
**Output:** over print function

(1) **if** $v = w$
(2) $\quad$ **then print**$(v,$ "length 0"$)$
(3) $\quad$ **else if** $\texttt{I}[v,w] < 0$ **then print**$($"no path"$)$
(4) $\quad$ **else print**$(v)$; **printPathInner**$(v, w)$; **print**$(w)$;

Exercise: Carry this out on an example. Why is the output correct?

## Theorem

Floyd and Warshall's algorithm solves the All Pairs Shortest Paths problem in time $O(n^3)$ and space $O(n^2)$.

The result is a data structure of size $O(n^2)$ (a matrix), that enables calculating a shortest path from $v$ to $w$, for any given pair $(v, w)$ of nodes. The running time for such a call is $O(\#(\text{edges on the path}))$.

**That's it!**
**Thanks for coming to the class, and**
**all the best for the exam!**