Algorithm 2.31: Construct-AC-Trie
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC trie: $root$, *child*() and *patterns*().

(1) Create new node $root$
(2) **for** $i \leftarrow 1$ **to** k **do**
(3) $\quad v \leftarrow root$; $j \leftarrow 0$
(4) $\quad$ **while** $child(v, P_i[j]) \neq \perp$ **do**
(5) $\quad\quad v \leftarrow child(v, P_i[j])$; $j \leftarrow j + 1$
(6) $\quad$ **while** $j < |P_i|$ **do**
(7) $\quad\quad$ Create new node u
(8) $\quad\quad child(v, P_i[j]) \leftarrow u$
(9) $\quad\quad v \leftarrow u$; $j \leftarrow j + 1$
(10) $\quad patterns(v) \leftarrow \{i\}$
(11) **return** $(root, child(), patterns())$

Lines (3)–(10) perform the standard trie insertion (Algorithm 1.2).

- Line (10) marks v as a representative of P_i.
- The creation of a new node v initializes *patterns*(v) to $\emptyset$ (in addition to initializing *child*(v, c) to $\perp$ for all $c \in \Sigma$).

Algorithm 2.32: Compute-AC-Fail
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC failure function *fail*() and updated *patterns*()

(1) Create new node $fallback$
(2) **for** $c \in \Sigma$ **do** $child(fallback, c) \leftarrow root$
(3) $fail(root) \leftarrow fallback$
(4) $queue \leftarrow \{root\}$
(5) **while** $queue \neq \emptyset$ **do**
(6) $\quad u \leftarrow \text{popfront}(queue)$
(7) $\quad$ **for** $c \in \Sigma$ such that $child(u, c) \neq \perp$ **do**
(8) $\quad\quad v \leftarrow child(u, c)$
(9) $\quad\quad w \leftarrow fail(u)$
(10) $\quad\quad$ **while** $child(w, c) = \perp$ **do** $w \leftarrow fail(w)$
(11) $\quad\quad fail(v) \leftarrow child(w, c)$
(12) $\quad\quad patterns(v) \leftarrow patterns(v) \cup patterns(fail(v))$
(13) $\quad\quad \text{pushback}(queue, v)$
(14) **return** $(fail(), patterns())$

The algorithm does a breath first traversal of the trie. This ensures that correct values of *fail*() and *patterns*() are already computed when needed.

fail(v) is correctly computed on lines (8)–(11):

- Let $fail^*(v) = \{v, fail(v), fail(fail(v)), \ldots, root\}$. These nodes are exactly the trie nodes that represent suffixes of S_v.
- Let $u = parent(v)$ and $child(u, c) = v$. Then $S_v = S_u c$ and a string S is a suffix of S_u iff Sc is suffix of S_v. Thus for any node w
  - If $w \in fail^*(v) \setminus \{root\}$, then $parent(w) \in fail^*(u)$.
  - If $w \in fail^*(u)$ and $child(w, c) \neq \perp$, then $child(w, c) \in fail^*(v)$.
- Therefore, $fail(v) = child(w, c)$, where w is the first node in $fail^*(u)$ other than u such that $child(w, c) \neq \perp$, or $fail(v) = root$ if no such node w exists.

patterns(v) is correctly computed on line (12):

$$\begin{aligned} patterns(v) &= \{i \mid P_i \text{ is a suffix of } S_v\} \\ &= \{i \mid P_i = S_w \text{ and } w \in fail^*(v)\} \\ &= \{i \mid P_i = S_v\} \cup patterns(fail(v)) \end{aligned}$$

Assuming σ is constant:

- The search time is $\mathcal{O}(n)$.
- The space complexity is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The implementation of *patterns*() requires care (exercise).
- The preprocessing time is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The only non-trivial issue is the while-loop on line (10).
  - Let $root, v_1, v_2, \ldots, v_\ell$ be the nodes on the path from root to a node representing a pattern P_i. Let $w_j = fail(v_j)$ for all j. Let *depth*(v) be the depth of a node v (*depth*($root$) = 0).
  - When processing v_j and computing $w_j = fail(v_j)$, we have $depth(w_j) = depth(w_{j-1}) + 1$ before line (10) and $depth(w_j) \leq depth(w_{j-1}) + 1 - t_j$ after line (10), where t_j is the number of rounds in the while-loop.
  - Thus, the total number of rounds in the while-loop when processing the nodes $v_1, v_2, \ldots, v_\ell$ is at most $\ell = |P_i|$, and thus over the whole algorithm at most $||\mathcal{P}||$.

The analysis when σ is not constant is left as an exercise.

Summary: Exact String Matching

Exact string matching is a fundamental problem in stringology. We have seen several different algorithms for solving the problem.

The properties of the algorithms vary with respect to worst case time complexity, average case time complexity, type of alphabet (ordered/integer) and even space complexity.

The algorithms use a wide range of completely different techniques:

- There exists numerous algorithms for exact string matching but most of them use variations or combinations of the techniques we have seen (study groups).
- Many of the techniques can be adapted to other problems. All of the techniques have some uses in practice.

3. Approximate String Matching

Often in applications we want to search a text for something that is similar to the pattern but not necessarily exactly the same.

To formalize this problem, we have to specify what does "similar" mean. This can be done by defining a similarity or a distance measure.

A natural and popular distance measure for strings is the edit distance, also known as the Levenshtein distance.

Edit distance

The edit distance $ed(A, B)$ of two strings A and B is the minimum number of edit operations needed to change A into B. The allowed edit operations are:

S Substitution of a single character with another character.

I Insertion of a single character.

D Deletion of a single character.

Example 3.1: Let A = `Lewensteinn` and B = `Levenshtein`. Then $ed(A, B) = 3$.

The set of edit operations can be described

with an edit sequence: `NNSNNNINNNND`
or with an alignment:
```
Lewens-teinn
Levenshtein-
```

In the edit sequence, N means No edit.

There are many variations and extension of the edit distance, for example:

- Hamming distance allows only the subtitution operation.
- Damerau–Levenshtein distance adds an edit operation:
  T Transposition swaps two adjacent characters.
- With weighted edit distance, each operation has a cost or weight, which can be other than one.
- Allow insertions and deletions (indels) of factors at a cost that is lower than the sum of character indels.

We will focus on the basic Levenshtein distance.

Levenshtein distance has the following two useful properties, which are not shared by all variations (exercise):

- Levenshtein distance is a metric.
- If $ed(A, B) = k$, there exists an edit sequence and an alignment with k edit operations, but no edit sequence or alignment with less than k edit operations. An edit sequence and an alignment with $ed(A, B)$ edit operations is called optimal.

Computing Edit Distance

Given two strings $A[1..m]$ and $B[1..n]$, define the values d_{ij} with the recurrence:

$$
\begin{aligned}
d_{00} &= 0, \\
d_{i0} &= i, \ \ 1 \le i \le m, \\
d_{0j} &= j, \ \ 1 \le j \le n, \ \text{and} \\
d_{ij} &= \min \begin{cases} d_{i-1,j-1} + \delta(A[i], B[j]) \\ \quad d_{i-1,j} + 1 \\ \quad d_{i,j-1} + 1 \end{cases} \quad 1 \le i \le m, 1 \le j \le n,
\end{aligned}
$$

where

$$\delta(A[i], B[j]) = \begin{cases} 1 & \text{if } A[i] \ne B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}$$

Theorem 3.2: $d_{ij} = ed(A[1..i], B[1..j])$ for all $0 \le i \le m$, $0 \le j \le n$. In particular, $d_{mn} = ed(A, B)$.

Example 3.3: $A =$ `ballad`, $B =$ `handball`

d		h	a	n	d	b	a	l	l
	0	1	2	3	4	5	6	7	8
b	1	1	2	3	4	4	5	6	7
a	2	2	1	2	3	4	4	5	6
l	3	3	2	2	3	4	5	4	5
l	4	4	3	3	3	4	5	5	4
a	5	5	4	4	4	4	4	5	5
d	6	6	5	5	4	5	5	5	6

$ed(A, B) = d_{mn} = d_{6,8} = 6$.

Proof of Theorem 3.2. We use induction with respect to $i + j$. For brevity, write $A_i = A[1..i]$ and $B_j = B[1..j]$.

Basis:

$$
\begin{aligned}
d_{00} &= 0 = ed(\epsilon, \epsilon) \\
d_{i0} &= i = ed(A_i, \epsilon) \quad (i \text{ deletions}) \\
d_{0j} &= j = ed(\epsilon, B_j) \quad (j \text{ insertions})
\end{aligned}
$$

Induction step: We show that the claim holds for d_{ij}, $1 \le i \le m, 1 \le j \le n$. By induction assumption, $d_{pq} = ed(A_p, B_q)$ when $p + q < i + j$.

Let E_{ij} be an optimal edit sequence with the cost $ed(A_i, B_j)$. We have three cases depending on what the last operation symbol in E_{ij} is:

N or S: $E_{ij} = E_{i-1,j-1}\mathrm{N}$ or $E_{ij} = E_{i-1,j-1}\mathrm{S}$ and $ed(A_i, B_j) = ed(A_{i-1}, B_{j-1}) + \delta(A[i], B[j]) = d_{i-1,j-1} + \delta(A[i], B[j])$.

I: $E_{ij} = E_{i,j-1}\mathrm{I}$ and $ed(A_i, B_j) = ed(A_i, B_{j-1}) + 1 = d_{i,j-1} + 1$.

D: $E_{ij} = E_{i-1,j}\mathrm{D}$ and $ed(A_i, B_j) = ed(A_{i-1}, B_j) + 1 = d_{i-1,j} + 1$.

One of the cases above is always true, and since the edit sequence is optimal, it must be one with the minimum cost, which agrees with the definition of d_{ij}. □

The recurrence gives directly a dynamic programming algorithm for computing the edit distance.

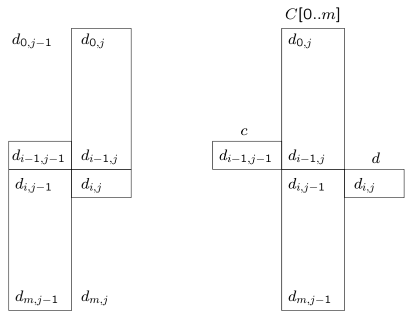
Algorithm 3.4: Edit distance
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$

(1) **for** $i \leftarrow 0$ **to** m **do** $d_{i0} \leftarrow i$
(2) **for** $j \leftarrow 1$ **to** n **do** $d_{0j} \leftarrow j$
(3) **for** $j \leftarrow 1$ **to** n **do**
(4)   **for** $i \leftarrow 1$ **to** m **do**
(5) $d_{ij} \leftarrow \min\{d_{i-1,j-1} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
(6) return d_{mn}

The time and space complexity is $\mathcal{O}(mn)$.

The space complexity can be reduced by noticing that each column of the matrix (d_{ij}) depends only on the previous column. We do not need to store older columns.

A more careful look reveals that, when computing d_{ij}, we only need to store the bottom part of column $j - 1$ and the already computed top part of column j. We store these in an array $C[0..m]$ and variables c and d as shown below:



Algorithm 3.5: Edit distance in $\mathcal{O}(m)$ space
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$

(1) **for** $i \leftarrow 0$ **to** m **do** $C[i] \leftarrow i$
(2) **for** $j \leftarrow 1$ **to** n **do**
(3) $c \leftarrow C[0]$; $C[0] \leftarrow j$
(4)   **for** $i \leftarrow 1$ **to** m **do**
(5) $d \leftarrow \min\{c + \delta(A[i], B[j]), C[i-1] + 1, C[i] + 1\}$
(6) $c \leftarrow C[i]$
(7) $C[i] \leftarrow d$
(8) return $C[m]$

Note that because $ed(A, B) = ed(B, A)$ (exercise), we can always choose A to be the shorter string so that $m \le n$.

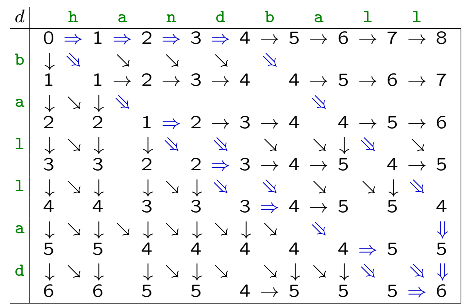
It is also possible to find optimal edit sequences and alignments from the matrix d_{ij}.

An edit graph is a directed graph, where the nodes are the cells of the edit distance matrix, and the edges are as follows:

- If $A[i] = B[j]$ and $d_{ij} = d_{i-1,j-1}$, there is an edge $(i-1, j-1) \to (i, j)$ labelled with N.
- If $A[i] \ne B[j]$ and $d_{ij} = d_{i-1,j-1} + 1$, there is an edge $(i-1, j-1) \to (i, j)$ labelled with S.
- If $d_{ij} = d_{i,j-1} + 1$, there is an edge $(i, j-1) \to (i, j)$ labelled with I.
- If $d_{ij} = d_{i-1,j} + 1$, there is an edge $(i-1, j) \to (i, j)$ labelled with D.

Any path from $(0, 0)$ to (m, n) is labelled with an optimal edit sequence.

Example 3.6: $A =$ `ballad`, $B =$ `handball`



There are 7 paths from $(0, 0)$ to $(6, 8)$ corresponding to 7 different optimal edit sequences and alignments, including the following three:

```
IIIINNNNDD    SNISSNIS    SNSSINSI
----ballad    ba-lla-d    ball-ad-
handball--    handball    handball
```

Approximate String Matching

Now we are ready to tackle the main problem of this part: approximate string matching.

Problem 3.7: Given a text $T[1..n]$, a pattern $P[1..m]$ and an integer $k \geq 0$, report all positions $j \in [1..m]$ such that $ed(P, T(j-\ell...j]) \leq k$ for some $\ell \geq 0$.

The factor $T(j-\ell...j]$ is called an approximate occurrence of P.

There can be multiple occurrences of different lengths ending at the same position j, but usually it is enough to report just the end positions.
We ask for the end position rather than the start position because that is more natural for the algorithms.

Define the values g_{ij} with the recurrence:

$$g_{0j} = 0, \;\; 0 \leq j \leq n,$$
$$g_{i0} = i, \;\; 1 \leq i \leq m, \text{ and}$$
$$g_{ij} = \min \begin{cases} g_{i-1,j-1} + \delta(P[i], T[j]) \\ g_{i-1,j} + 1 \\ g_{i,j-1} + 1 \end{cases} \quad 1 \leq i \leq m, 1 \leq j \leq n.$$

Theorem 3.8: For all $0 \leq i \leq m$, $0 \leq j \leq n$:

$$g_{ij} = \min\{ed(P[1..i], T(j-\ell...j]) \mid 0 \leq \ell \leq j\} \; .$$

In particular, j is an ending position of an approximate occurrence if and only if $g_{mj} \leq k$.

Proof. We use induction with respect to $i + j$.

Basis:

$g_{00} = 0 = ed(\epsilon, \epsilon)$
$g_{0j} = 0 = ed(\epsilon, \epsilon) = ed(\epsilon, T(j-0..j])$ (min at $\ell = 0$)
$g_{i0} = i = ed(P[1..i], \epsilon) = ed(P[1..i], T(0-0..0])$ ($0 \leq \ell \leq j = 0$)

Induction step: Essentially the same as in the proof of Theorem 3.2.

Example 3.9: $P =$ match, $T =$ remachine, $k = 1$

g		r	e	m	a	c	h	i	n	e
	0	0	0	0	0	0	0	0	0	0
m	1	1	1	0	1	1	1	1	1	1
a	2	2	2	1	0	1	2	2	2	2
t	3	3	3	2	1	1	2	3	3	3
c	4	4	4	3	2	1	2	3	4	4
h	5	5	5	4	3	2	1	2	3	4

One occurrence ending at position 6.

Algorithm 3.10: Approximate string matching
Input: text $T[1..n]$, pattern $P[1..m]$, and integer k
Output: end positions of all approximate occurrences of P

for $i \leftarrow 0$ to m do $g_{i0} \leftarrow i$
for $j \leftarrow 1$ to n do $g_{0j} \leftarrow 0$
for $j \leftarrow 1$ to n do
    for $i \leftarrow 1$ to m do
        $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
    if $g_{mj} \leq k$ then output j

- Time and space complexity is $\mathcal{O}(mn)$ on ordered alphabet.
- The space complexity can be reduced to $\mathcal{O}(m)$ by storing only one column as in Algorithm 3.5.

Ukkonen's Cut-off Heuristic

We can speed up the algorithm using the diagonal monotonicity of the matrix (g_{ij}):

A diagonal d, $-m \leq d \leq n$, consists of the cells g_{ij} with $j - i = d$.
Every diagonal in (g_{ij}) is monotonically non-decreasing.

Example 3.11: Diagonals -3 and 2.

g		r	e	m	a	c	h	i	n	e
	0	0	0	0	0	0	0	0	0	0
m	1	1	1	0	1	1	1	1	1	1
a	2	2	2	1	0	1	2	2	2	2
t	3	3	3	2	1	1	2	3	3	3
c	4	4	4	3	2	1	2	3	4	4
h	5	5	5	4	3	2	1	2	3	4

Lemma 3.12: For every $i \in [1..m]$ and every $j \in [1..n]$, $g_{ij} = g_{i-1,j-1}$ or $g_{ij} = g_{i-1,j-1} + 1$.

Proof. By definition, $g_{ij} \leq g_{i-1,j-1} + \delta(P[i], T[j]) \leq g_{i-1,j-1} + 1$. We show that $g_{ij} \geq g_{i-1,j-1}$ by induction on $i + j$.

The induction assumption is that $g_{pq} \geq g_{p-1,q-1}$ when $p \in [1..m]$, $q \in [1..n]$ and $p + q < i + j$. At least one of the following holds:

1. $g_{ij} = g_{i-1,j-1} + \delta(P[i], T[j])$. Then $g_{ij} \geq g_{i-1,j-1}$.
2. $g_{ij} = g_{i-1,j} + 1$ and $i > 1$. Then
$$g_{ij} = g_{i-1,j} + 1 \overset{\text{ind. assump.}}{\geq} g_{i-2,j-1} + 1 \overset{\text{definition}}{\geq} g_{i-1,j-1}$$
3. $g_{ij} = g_{i,j-1} + 1$ and $j > 1$. Then
$$g_{ij} = g_{i,j-1} + 1 \overset{\text{ind. assump.}}{\geq} g_{i-1,j-2} + 1 \overset{\text{definition}}{\geq} g_{i-1,j-1}$$
4. $g_{ij} = g_{i-1,j} + 1$ and $i = 1$. Then $g_{ij} = 0 + 1 > 0 = g_{i-1,j-1}$.
5. $g_{ij} = g_{i,j-1} + 1$ and $j = 1$. Then $g_{ij} = i + 1 = (i-1) + 2 = g_{i-1,j-1} + 2$, which cannot be true. Thus this case can never happen. □

We can reduce computation using diagonal monotonicity:

- Whenever the value on a diagonal d grows larger than k, we can discard d from consideration, because we are only interested in values at most k on the row m.
- We keep track of the smallest undiscarded diagonal d. Each column is computed only up to diagonal $d + 1$.

Example 3.13: $P =$ strict, $T =$ datastructure, $k = 1$

g		d	a	t	a	s	t	r	u	c	t	u	r	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	1	1	1	1	1	0	1	1	1	1	1	1	1	1
t	2	2	2	1	2	1	0	1	2	2	1	2	2	2
r				2	2	2	1	0	1	2	2	2		
i							2	1	1	2	3	3		
c								2	2	1	2	3		
t										2	1	2		

The position of the smallest undiscarded diagonal on the current column is kept in a variable top.

Algorithm 3.14: Ukkonen's cut-off algorithm
Input: text $T[1..n]$, pattern $P[1..m]$, and integer k
Output: end positions of all approximate occurrences of P

$top \leftarrow \min(k+1, m)$
for $i \leftarrow 0$ to top do $g_{i0} \leftarrow i$
for $j \leftarrow 1$ to n do $g_{0j} \leftarrow 0$
for $j \leftarrow 1$ to n do
    for $i \leftarrow 1$ to top do
        $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
    while $g_{top,j} > k$ do $top \leftarrow top - 1$
    if $top = m$ then output j
    else $top \leftarrow top + 1$; $g_{top,j} \leftarrow k + 1$

The time complexity is proportional to the computed area in the matrix (g_{ij}).

- The worst case time complexity is still $\mathcal{O}(mn)$ on ordered alphabet.
- The average case time complexity is $\mathcal{O}(kn)$. The proof is not trivial.

There are many other algorithms based on diagonal monotonicity. Some of them achieve $\mathcal{O}(kn)$ worst case time complexity.