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

1. Create new node root
2. for \( i \leftarrow 1 \) to \( k \) do
3. \( v \leftarrow \text{root}; j \leftarrow 0 \)
4. while \( \text{child}(v, P[j]) \neq \perp \) do
5. \( v \leftarrow \text{child}(v, P[j]); j \leftarrow j + 1 \)
6. \( \text{while } j < |P| \) do
7. Create new node \( u \)
8. \( \text{child}(v, P[j]) \leftarrow u \)
9. \( v \leftarrow u; j \leftarrow j + 1 \)
10. patterns(v) ← \{i|Pi is a suffix of Sv\}
11. return (root, child(), patterns())

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

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

Let \( \text{fail}^*(v) = \{v, \text{fail}(v), \text{fail}^*(\text{fail}(v)), \ldots, \text{root}\} \). These nodes are exactly the trie nodes that represent suffixes of \( S_v \).

Let \( w = \text{parent}(v) \) and \( \text{child}(u, c) = v \). Then \( S_t = S_v c \) and a string \( S \) is a suffix of \( S_t \) iff \( S \) is a suffix of \( S_v \). Thus for any node \( w \):

- If \( w \in \text{fail}^*(v) \setminus \{\text{root}\} \), then \( \text{parent}(w) \in \text{fail}^*(u) \).
- If \( w \in \text{fail}(u) \) and \( \text{child}(u, c) \neq \perp \), then \( \text{child}(w, c) \in \text{fail}^*(v) \).

Therefore, \( \text{fail}(v) = \text{child}(w, c) \), where \( w \) is the first node in \( \text{fail}(u) \) other than \( v \) such that \( \text{child}(w, c) \neq \perp \), or \( \text{fail}(v) = \text{root} \) if no such node \( w \) exists.

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

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 \( \text{child}(\text{fallback}, c) \leftarrow \text{root} \)
3. \( \text{fail}(\text{root}) \leftarrow \text{fallback} \)
4. queue ← \{root\}
5. while queue ≠ ∅ do
6. \( u \leftarrow \text{popfront}(queue) \)
7. \( \text{for } c \in \Sigma \text{ such that child}(u, c) \neq \perp \) do
8. \( v \leftarrow \text{child}(u, c) \)
9. \( w \leftarrow \text{fail}(u) \)
10. \( \text{while child}(w, c) \neq \perp \) do \( w \leftarrow \text{fail}(w) \)
11. \( \text{fail}(v) \leftarrow \text{child}(w, c) \)
12. \( \text{patterns}(v) \leftarrow \text{patterns}(v) \cup \text{patterns}(\text{fail}(v)) \)
13. \( \text{pushback}(queue, v) \)
14. return (fail(), patterns())

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.

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

- Hamming distance allows only the substitution 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. |
|   | New 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:

$$
d_{00} = 0,
\quad d_{0i} = i, \quad 1 \leq i \leq m,
\quad d_{ij} = j, \quad 1 \leq j \leq n,
\quad d_{ij} = \min \left\{ \begin{array}{ll}
  d_{i-1,j} + \delta(A[i], B[j]) \\
  d_{i-1,j} + 1 \\
  d_{i,j-1} + 1
\end{array} \right. \quad 1 \leq i \leq m, 1 \leq j \leq n,
$$

where

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

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

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:**

$d_{00} = 0 = \text{ed}(\epsilon, \epsilon)$

$d_{0i} = i = \text{ed}(A_i, \epsilon)$ (i deletions)

$d_{ij} = j = \text{ed}(\epsilon, B_j)$ (j insertions)

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

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

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

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

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

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 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:

$$
d_{i-1} \quad d_{i-2} \quad \cdots \quad d_{i-j} \\
\vdots \quad \vdots \quad \cdots \quad \vdots \\
C[0] \quad C[1] \quad \cdots \quad C[m]
$$

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

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

**Example 3.3:** $A = \text{ballad}$, $B = \text{handball}$

<table>
<thead>
<tr>
<th>$d$</th>
<th>$a$</th>
<th>$b$</th>
<th>$h$</th>
<th>$a$</th>
<th>$b$</th>
<th>$l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

The time and space complexity is $O(mn)$.

**Algorithm 3.4:** Edit distance

Input: strings $A[1..m]$ and $B[1..n]$

Output: $\text{ed}(A, B)$

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. $d_{ij} \leftarrow \min\{d_{i-1,j} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
6. return $d_{mn}$

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

**Example 3.6:** $A = \text{ballad}$, $B = \text{handball}$

<table>
<thead>
<tr>
<th>$d$</th>
<th>$a$</th>
<th>$b$</th>
<th>$h$</th>
<th>$a$</th>
<th>$b$</th>
<th>$l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- IIIINRDQ, SKRSSIS, SRRSNISI
- ----ballad ba-lia-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.

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 $P$

1. if $i \leq 0$ to $m$ do $g_0 \leftarrow i$
2. for $j \leftarrow 1$ to $m$ do $g_0 \leftarrow j$
3. for $i \leftarrow 1$ to $n$ do
   4. for $i \leftarrow 1$ to $m$ do
      5. $g_{ij} \leftarrow \min\{g_{ij-1} + \delta(A[i],B[j]),g_{ij-1} + 1,g_{ij-1} + 1\}$
5. if $g_{ij} \leq k$ then output $j$

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

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

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 $i$ and $j$.

The induction assumption is that $g_{ij} \geq g_{i-1,j-1}$ 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

\[
g_{ij} = g_{i-1,j} + 1 \geq g_{i-2,j} + 1 \geq g_{i-1,j} - 1
\]

3. $g_{ij} = g_{i-1,j} + 1$ and $j > 1$. Then

\[
g_{ij} = g_{i-1,j} + 1 \geq g_{i-1,j-2} + 1 \geq g_{i-1,j-1}
\]

4. $g_{ij} = g_{i-1,j} + 1$ and $i = 1$. Then $g_{ij} = 0 + 1 > g_{i-1,j-1}$.

5. $g_{ij} = g_{i-1,j} + 1$ 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.

Define the values $g_0$ with the recurrence:

\[
g_0 = 0, \quad 0 \leq j \leq n;
\]

\[
g_0 = i, \quad 1 \leq i \leq m, \quad g_0 = 0
\]

\[
g_0 = \min\{g_{i-1,j} + \delta(P[i],T[j])\}
\]

\[
1 \leq i \leq m, 1 \leq j \leq n.
\]

\[
g_0 = g_{i-1,j} + 1
\]

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

\[
g_0 = \min\{ed(P[1..i],T[j-\ell..j])\} \quad \text{if} \quad 0 \leq \ell \leq j
\]

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

Example 3.9: $P = \text{match}$, $T = \text{remachine}$, $k = 1$

```
g r e m a c h i n e
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.

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
m \`3 3 3 2 1 1 2 3 3 3
a \`2 2 2 1 0 1 2 2 2 2
t \`1 1 1 0 1 1 1 1 1 1
c \`4 4 4 3 2 1 2 3 4 4
h \`5 5 5 4 3 2 1 2 3 4
```

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 $n$.
- We keep track of the smallest undiscarded diagonal $d$. Each column is computed only up to diagonal $d+1$.

Example 3.13: $P = \text{strict}$, $T = \text{datastructure}$, $k = 1$

```
g d a t a s t r u c t u r e
s \`0 0 0 0 0 0 0 0 0 0
r \`0 0 0 0 0 0 0 0 0 0
```

We can reduce computation using diagonal monotonicity:
The position of the smallest undiscarded diagonal on the current column is kept in a variable \( \text{top} \).

**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 \( P \)

1. \( \text{top} \leftarrow \min(k + 1, m) \)
2. for \( i \leftarrow 0 \) to \( \text{top} \) do \( g_{0,i} \leftarrow i \)
3. for \( j \leftarrow 1 \) to \( n \) do \( g_{0,j} \leftarrow 0 \)
4. for \( j \leftarrow 1 \) to \( n \) do
5.   for \( i \leftarrow 1 \) to \( \text{top} \) do
6.     \( g_{i,j} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\} \)
7.     while \( g_{\text{top},j} > k \) do \( \text{top} \leftarrow \text{top} - 1 \)
8.     if \( \text{top} = m \) then output \( j \)
9.     else \( \text{top} \leftarrow \text{top} + 1; g_{\text{top},j} \leftarrow k + 1 \)

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

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

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