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 $P$
1. top ← min($k + 1$, $n$)
2. for $i \leftarrow 0$ to top do $g_{0,0} \leftarrow i$
3. for $j \leftarrow 1$ to $k$ do $g_{0,j} \leftarrow 0$
4. for $j \leftarrow 1$ to top do $g_{j,0} \leftarrow j$
5. for $i \leftarrow 1$ to top do $g_{i,j} \leftarrow g_{i-1,j-1} + 1$ if $A[i][j][j]$, $g_{i-1,j} + 1$, $g_{i,j-1} + 1$
6. while $g_{top,j} > k$ do $top \leftarrow top - 1$
7. if $top = m$ then output $j$
8. 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 $O(mn)$ 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.

**Myers’ Bitparallel Algorithm**
Another way to speed up the computation is bitparallelism. Instead of the matrix $(g_{ij})$, we store differences between adjacent cells:

- **Vertical delta:** $\Delta v_{ij} = g_{i,j} - g_{i-1,j}$
- **Horizontal delta:** $\Delta h_{ij} = g_{i,j} - g_{i,j-1}$
- **Diagonal delta:** $\Delta d_{ij} = g_{i,j} - g_{i-1,j-1}$

Because $g_{0,0} = 0$, $g_{i,j} = \Delta v_{i,0} + \Delta v_{1,0} + \cdots + \Delta v_{i,j}$, $i + \Delta h_{1,i} + \Delta h_{2,i} + \cdots + \Delta h_{i,i}$

Because of diagonal monotonicity, $\Delta d_{ij} \in \{-1,0,1\}$ and it can be stored in one bit. By the following result, $\Delta h_{ij}$ and $\Delta v_{ij}$ can be stored in two bits.

**Lemma 3.15:** $\Delta h_{ij}, \Delta v_{ij} \in \{-1,0,1\}$ for every $i,j$ that they are defined for.

The proof is left as an exercise.

In the standard computation of a cell:
• **Input** is $\Delta v_{ij}, \Delta h_{ij}, \Delta d_{ij-1}$ and $\delta(P[i], T[j])$.
• **Output** is $g_{ij}$.

In the corresponding bitparallel computation:
• **Input** is $\Delta v^{in} = \Delta v_{i-1,j}$, $\Delta h^{in} = \Delta h_{i-1,j}$ and $E_{gij} = 1 - \delta(P[i], T[j])$.
• **Output** is $\Delta v^{out} = \Delta v_{i,j}$ and $\Delta h^{out} = \Delta h_{i,j}$,
   $g_{i-1,j-1} \xrightarrow{\Delta h^{in}} g_{i-1,j} \xrightarrow{\Delta v^{in}} g_{i,j}$
   $g_{i,j-1} \xrightarrow{\Delta h^{out}} \Delta h^{out} \xrightarrow{\Delta v^{out}} g_{ij}$

The algorithm does not compute the $\Delta d$ values but they are useful in the proofs.

To enable bitparallel operation, we need two changes:
• The $\Delta v$ and $\Delta h$ values are “trits” not bits. We encode each of them with two bits as follows:

<table>
<thead>
<tr>
<th>$Pv$</th>
<th>$Mv$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $\Delta v = +1$</td>
<td>1 if $\Delta v = -1$</td>
</tr>
<tr>
<td>0 otherwise</td>
<td>0 otherwise</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$Ph$</th>
<th>$Mh$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 if $\Delta h = +1$</td>
<td>0 otherwise</td>
</tr>
<tr>
<td>0 otherwise</td>
<td>0 otherwise</td>
</tr>
</tbody>
</table>

Then

$\Delta v = Pv + Mv$

$\Delta h = Ph - Mh$

• We replace arithmetic operations ($+$, $-$, $\min$) with Boolean (logical) operations ($\lor$, $\land$, $\neg$).

The computation rule is defined by the following result.

**Lemma 3.17:** If $Eq = 1$ or $\Delta v^{in} = -1$ or $\Delta h^{in} = -1$, then $\Delta d = 0$, $\Delta v^{out} = \Delta v^{in}$ and $\Delta h^{out} = \Delta h^{in}$. Otherwise $\Delta d = 1$, $\Delta v^{out} = 1 - \Delta h^{in}$ and $\Delta h^{out} = 1 - \Delta v^{in}$.

**Proof.** We can write the recurrence for $g_{ij}$ as $g_{ij} = \min(g_{i-1,j-1} + \delta(P[i], T[j]), g_{i-1,j} + 1, g_{i,j-1} + 1)$ which is 0 if $Eq = 1$ or $\Delta v^{in} = -1$ or $\Delta h^{in} = -1$ and 1 otherwise.

Clearly $\Delta d = \Delta v^{in} + \Delta h^{out} = \Delta h^{in} + \Delta v^{out}$.
Thus $\Delta v^{out} = \Delta d - \Delta h^{in}$ and $\Delta h^{out} = \Delta d - \Delta v^{in}$.

Now the computation rules can be expressed as follows.

**Lemma 3.18:** $Pv^{out} = Mv^{in} \lor \neg(Xv \lor Ph^{in})$ $Mv^{out} = Ph^{in} \land Xv$

$Ph^{out} = Mv^{in} \lor \neg(Xh \lor Ph^{in})$ $Mh^{out} = Ph^{in} \land Xh$

where $Xv = Eq \lor Mh^{in}$ and $Xh = Eq \lor Mh^{in}$.

**Proof.** We show the claim for $Pv$ and $Mv$ only. $Ph$ and $Mh$ are symmetrical.

By Lemma 3.17,

$Pv^{out} = (\neg \Delta d \land Mh^{in}) \lor (\Delta d \land \neg Ph^{in})$

$Mv^{out} = (\neg \Delta d \land Ph^{in}) \lor (\Delta d \land 0)$

Because $\Delta d = Eq \lor Mh^{in} \land Mb^{in}$, $\neg(Xv \lor Mb^{in}) = Eq \land \neg Mb^{in}$,

$Pv^{out} = (\neg Eq \lor Mh^{in} \land Mb^{in}) \lor (\neg Eq \land \neg Mb^{in} \land \neg Ph^{in})$

$Mv^{out} = (\neg Eq \lor Mb^{in} \land Mb^{in}) \lor (\neg Eq \land Mb^{in} \land Ph^{in})$

All the steps above use just basic laws of Boolean algebra except the last step, where we use the fact that $Mb^{in}$ and $Ph^{in}$ cannot be 1 simultaneously.
According to Lemma 3.18, the bit representation of the matrix can be computed as follows.

\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do } \\
P_{vi0} \leftarrow 1; \ldots \text{ do } \\
X_{vij} \leftarrow E_{qij} \lor M_{vi,j-1} \\
P_{vij} \leftarrow M_{hi-1,j} \lor \neg(X_{vij} \lor P_{hi-1,j}) \\
M_{vij} \leftarrow P_{hi-1,j} \land X_{vij} \\
\text{for } i \leftarrow 1 \text{ to } m \text{ do } \\
X_{hij} \leftarrow E_{qij} \lor M_{hij-1} \\
P_{hij} \leftarrow M_{hij-1} \land \neg(X_{hij} \lor P_{hij-1}) \\
M_{hij} \leftarrow P_{hij-1} \land X_{hij}
\]

This is not yet bitparallel though.

**Lemma 3.19:** \( X_{hij} = \exists \in \{1, \ldots, n\} : E_{qij} \land (\forall x \in \{1, \ldots, n\} : P_{xh(i-1)}). \)

**Proof.** We use induction on \( i \).

Basis \( i = 1 \): The right-hand side reduces to \( E_{qij} \), because \( \ell = 1 \). By Lemma 3.18, \( X_{hij} = E_{qij} \lor M_{hij} \), which is \( E_{qij} \) because \( M_{hij} = 0 \) for all \( j \).

Induction step: The induction assumption is that \( X_{h(i-1),j} \) is as claimed. Now we have

\[
\exists \in \{1, \ldots, n\} : E_{qij} \land (\forall x \in \{1, \ldots, n\} : P_{xh(i-1)}) \\
= E_{qij} \land (\forall x \in \{1, \ldots, n\} : P_{xh(i-1)}) \\
= E_{qij} \land (P_{xh(i-1)} \land \exists x \in \{1, \ldots, n\} : E_{qij} \land (\forall x \in \{1, \ldots, n\} : P_{xh(i-1)})) \\
= E_{qij} (P_{xh(i-1)} \land X_{h(i-1),j}) \text{ (ind. assum.)} \\
= E_{qij} (P_{xh(i-1)} \land \neg \exists x \in \{1, \ldots, n\} : E_{qij} \land (\forall x \in \{1, \ldots, n\} : P_{xh(i-1)})) \\
= X_{hij} \quad \text{(Lemma 3.18)}
\]

At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time; let alone a whole vector \( X_{hij} \). However, it can be done, but we need more bit operations:

- Let \( \land \) denote the xor-operation: \( 0 \land 1 = 1 \land 0 = 1 \land 1 = 0 \).
- A bitvector is interpreted as an integer and we use addition as a bit operation. The carry mechanism in addition plays a key role. For example \( 0010 + 0111 = 1001 \).

In the following, for a bitvector \( B \), we write

\[
B = B[1..m] = B[m]B[m-1] \ldots B[1]
\]

The reverse order of the bits reflects the interpretation as an integer.

As a final detail, we compute the bottom row values \( g_{nij} \) using the equalities \( g_{nij} = m \lor g_{nij} = g_{n(i-1),j} + \Delta h_{nij} \).

**Algorithm 3.21:** Myers’ bitparallel algorithm

**Input:** text \( T[1..n] \), pattern \( P[1..m] \), and integer \( k \)

**Output:** end positions of all approximate occurrences of \( P \)

\[
\text{for } c \in \Sigma \text{ do } B[c] \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } m \text{ do } B[P[i]] \leftarrow 1 \\
Pc \leftarrow \ldots; \text{ Me } \leftarrow \ldots; g \leftarrow \ldots \\
\text{for } j \leftarrow 1 \text{ to } n \text{ do } \\
E_{qij} \leftarrow B[P[i]] \\
X_{hij} \leftarrow ((E_{qij} \lor P_{c}) \lor P_{e}) \lor E_{qij} \\
P_{hij} \leftarrow P_{c} \lor (X_{hij} \lor P_{e}) \\
M_{hij} \leftarrow P_{c} \lor X_{hij} \\
X_{vij} \leftarrow E_{qij} \lor Me \\
P_{vij} \leftarrow (M_{hij} < 1) \lor (-X_{vij} \lor (P_{hij} < 1)) \\
M_{vij} \leftarrow (P_{hij} < 1) \land X_{vij} \\
g \leftarrow g + Pa[m] - Ma[n] \\
\text{if } g \leq k \text{ then output } j
\]

On an integer alphabet, when \( m \leq w \):

- Pattern preprocessing time is \( O(m + \sigma) \).
- Search time is \( O(n) \).

When \( m > w \), we can store each bit vector in \( [m/w] \) machine words:

- The worst case search time is \( O(n/[m/w]) \).
- Using Ukkonen’s cut-off heuristic, it is possible reduce the average case search time to \( O(n/[m/w]) \).

There are also algorithms for bitparallel simulation of a nondeterministic automaton that recognizes the approximate occurrences of the pattern.

**Example 3.22:**

\[
\begin{array}{cccc}
| & | & | & | \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
\end{array}
\]

\[
P = \text{pattern, } k = 3
\]

\[
\begin{array}{cccc}
| & | & | & | \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
\end{array}
\]
Another way to utilize Lemma 3.15 (\(\Delta_{ij}, \Delta_{vij} \in \{-1, 0, 1\}\)) is to use precomputed tables to process multiple matrix cells at a time.

- There are at most 3\(^n\) different columns. Thus there exists a deterministic automaton with 3\(^n\) states and 3\(^3n\) transitions that can find all approximate occurrences in \(O(n)\) time. However, the space and constructions time of the automaton can be too big to be practical.
- There is a super-alphabet algorithm that processes \(O(\log_2 n)\) characters at a time and \(O(\log_2^2 n)\) matrix cells at a time using lookup tables of size \(O(n)\). This gives time complexity \(O(mn/\log_2^2 n)\).
- A practical variant uses smaller lookup tables to compute multiple entries of a column at a time.

### Baeza-Yates–Perleberg Filtering Algorithm

A filtering algorithm for approximate string matching searches the text for factors having some property that satisfies the following conditions:

1. Every approximate occurrence of the pattern has this property.
2. Strings having this property are reasonably rare.
3. Text factors having this property can be found quickly.

Each text factor with the property is a potential occurrence, which is then verified for whether it is an actual approximate occurrence.

Filtering algorithms can achieve linear or even sublinear average case time complexity.

The algorithm has two phases:

**Filtration:** Search the text \(T\) for exact occurrences of the pattern factors \(P\).

Using the Aho–Corasick algorithm this takes \(O(n)\) time for a constant alphabet.

**Verification:** An area of length \(O(m)\) surrounding each potential occurrence found in the filtration phase is searched using the standard dynamic programming algorithm in \(O(m^2)\) time.

The worst case time complexity is \(O(m^2 n)\), which can be reduced to \(O(mn^2)\) by combining any overlapping areas to be searched.

Let us analyze the average case time complexity of the verification phase.

- The best pattern partitioning is as even as possible. Then each pattern factor has length at least \(r = \left\lceil m/(k + 1) \right\rceil\).
- The expected number of exact occurrences of a random string of length \(r\) in a random text of length \(n\) is at most \(n/r^2\).
- The expected total verification time is at most
  \[
  O\left(\frac{m^2(k+1)n}{\sigma^2}\right) \leq O\left(\frac{m^3n}{\sigma^2}\right).
  \]
  This is \(O(n)\) if \(r \geq 3\log_2 m\).
- The condition \(r \geq 3\log_2 m\) is satisfied when \((k + 1) \leq m/(3\log_2 m + 1)\).

**Theorem 3.24:** The average case time complexity of the Baeza-Yates–Perleberg algorithm is \(O(n)\) when \(k \leq m/(3\log_2 m + 1) - 1\).

### Summary: Approximate String Matching

We have seen two main types of algorithms for approximate string matching:

- Basic dynamic programming time complexity is \(O(mn)\). The time complexity can be improved to \(O(nm)\) using diagonal monotonicity, and to \(O(n(m/w))\) using bitparallelism.
- Filtering algorithms can improve average case time complexity and are the fastest in practice when \(k\) is not too large.

Similar techniques can be useful for other variants of edit distance but not always straightforwardly.