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.

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

Because $g_{i0} = i$ ja $g_{0j} = 0$,

$$\begin{aligned} g_{ij} &= \Delta v_{1j} + \Delta v_{2j} + \cdots + \Delta v_{ij} \\ &= i + \Delta h_{i1} + \Delta h_{i2} + \cdots + \Delta h_{ij} \end{aligned}$$

Because of diagonal monotonicity, $\Delta d_{ij} \in \{0, 1\}$ and it can be stored in one bit. By the following result, Δh_{ij} and Δv_{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.

Example 3.16: '−' means -1, '=' means 0 and '+' means $+1$

		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

In the standard computation of a cell:

- Input is $g_{i-1,j}$, $g_{i-1,j-1}$, $g_{i,j-1}$ and $\delta(P[i], T[j])$.
- Output is g_{ij}.

In the corresponding bitparallel computation:

- Input is $\Delta v^{\text{in}} = \Delta v_{i,j-1}$, $\Delta h^{\text{in}} = \Delta h_{i-1,j}$ and $Eq_{ij} = 1 - \delta(P[i], T[j])$.
- Output is $\Delta v^{\text{out}} = \Delta v_{i,j}$ and $\Delta h^{\text{out}} = \Delta h_{i,j}$.

$$\begin{tikzcd} g_{i-1,j-1} \arrow[r, "\Delta h^{\text{in}}"] \arrow[d, "\Delta v^{\text{in}}"'] & g_{i-1,j} \arrow[d, "\Delta v^{\text{out}}"] \\ g_{i,j-1} \arrow[r, "\Delta h^{\text{out}}"'] & g_{ij} \end{tikzcd}$$

The algorithm does not compute the Δd values but they are useful in the proofs.

The computation rule is defined by the following result.

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

Proof. We can write the recurrence for g_{ij} as

$$\begin{aligned} g_{ij} &= \min\{g_{i-1,j-1} + \delta(P[i], T[j]), g_{i,j-1} + 1, g_{i-1,j} + 1\} \\ &= g_{i-1,j-1} + \min\{1 - Eq, \Delta v^{\text{in}} + 1, \Delta h^{\text{in}} + 1\}. \end{aligned}$$

Then $\Delta d = g_{ij} - g_{i-1,j-1} = \min\{1 - Eq, \Delta v^{\text{in}} + 1, \Delta h^{\text{in}} + 1\}$ which is 0 if $Eq = 1$ or $\Delta v^{\text{in}} = -1$ or $\Delta h^{\text{in}} = -1$ and 1 otherwise.

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

To enable bitparallel operation, we need two changes:

- The Δv and Δh values are "trits" not bits. We encode each of them with two bits as follows:

$$Pv = \begin{cases} 1 & \text{if } \Delta v = +1 \\ 0 & \text{otherwise} \end{cases} \qquad Mv = \begin{cases} 1 & \text{if } \Delta v = -1 \\ 0 & \text{otherwise} \end{cases}$$

$$Ph = \begin{cases} 1 & \text{if } \Delta h = +1 \\ 0 & \text{otherwise} \end{cases} \qquad Mh = \begin{cases} 1 & \text{if } \Delta h = -1 \\ 0 & \text{otherwise} \end{cases}$$

  Then

$$\begin{aligned} \Delta v &= Pv - Mv \\ \Delta h &= Ph - Mh \end{aligned}$$

- We replace arithmetic operations ($+$, $-$, min) with Boolean (logical) operations ($\wedge$, $\vee$, $\neg$).

Now the computation rules can be expressed as follows.

Lemma 3.18:

$$\begin{aligned} Pv^{\text{out}} &= Mh^{\text{in}} \vee \neg(Xv \vee Ph^{\text{in}}) & \qquad Mv^{\text{out}} &= Ph^{\text{in}} \wedge Xv \\ Ph^{\text{out}} &= Mv^{\text{in}} \vee \neg(Xh \vee Pv^{\text{in}}) & \qquad Mh^{\text{out}} &= Pv^{\text{in}} \wedge Xh \end{aligned}$$

where $Xv = Eq \vee Mv^{\text{in}}$ and $Xh = Eq \vee Mh^{\text{in}}$.

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

By Lemma 3.17,

$$\begin{aligned} Pv^{\text{out}} &= (\neg\Delta d \wedge Mh^{\text{in}}) \vee (\Delta d \wedge \neg Ph^{\text{in}}) \\ Mv^{\text{out}} &= (\neg\Delta d \wedge Ph^{\text{in}}) \vee (\Delta d \wedge 0) = \neg\Delta d \wedge Ph^{\text{in}} \end{aligned}$$

Because $\Delta d = \neg(Eq \vee Mv^{\text{in}} \vee Mh^{\text{in}}) = \neg(Xv \vee Mh^{\text{in}}) = \neg Xv \wedge \neg Mh^{\text{in}}$,

$$\begin{aligned} Pv^{\text{out}} &= ((Xv \vee Mh^{\text{in}}) \wedge Mh^{\text{in}}) \vee (\neg Xv \wedge \neg Mh^{\text{in}} \wedge \neg Ph^{\text{in}}) \\ &= Mh^{\text{in}} \vee \neg(Xv \vee Mh^{\text{in}} \vee Ph^{\text{in}}) = Mh^{\text{in}} \vee \neg(Xv \vee Ph^{\text{in}}) \\ Mv^{\text{out}} &= (Xv \vee Mh^{\text{in}}) \wedge Ph^{\text{in}} = (Xv \wedge Ph^{\text{in}}) \vee (Mh^{\text{in}} \wedge Ph^{\text{in}}) = Xv \wedge Ph^{\text{in}} \end{aligned}$$

All the steps above use just basic laws of Boolean algebra except the last step, where we use the fact that Mh^{in} and Ph^{in} cannot be 1 simultaneously. $\square$

According to Lemma 3.18, the bit representation of the matrix can be computed as follows.

for $i \leftarrow 1$ to m do
  $Pv_{i0} \leftarrow 1;\ Mv_{i0} \leftarrow 0$
for $j \leftarrow 1$ to n do
  $Ph_{0j} \leftarrow 0;\ Mh_{0j} \leftarrow 0$
  for $i \leftarrow 1$ to m do
    $Xh_{ij} \leftarrow Eq_{ij} \vee Mh_{i-1,j}$
    $Ph_{ij} \leftarrow Mv_{i,j-1} \vee \neg(Xh_{ij} \vee\ Pv_{i,j-1})$
    $Mh_{ij} \leftarrow Pv_{i,j-1} \wedge Xh_{ij}$
  for $i \leftarrow 1$ to m do
    $Xv_{ij} \leftarrow Eq_{ij} \vee Mv_{i,j-1}$
    $Pv_{ij} \leftarrow Mh_{i-1,j} \vee \neg(Xv_{ij} \vee\ Ph_{i-1,j})$
    $Mv_{ij} \leftarrow Ph_{i-1,j} \wedge Xv_{ij}$

This is not yet bitparallel though.

To obtain a bitparallel algorithm, the columns Pv_{*j}, Mv_{*j}, Xv_{*j}, Ph_{*j}, Mh_{*j}, Xh_{*j} and Eq_{*j} are stored in bitvectors.

Now the second inner loop can be replaced with the code

$Xv_{*j} \leftarrow Eq_{*j} \vee Mv_{*,j-1}$
$Pv_{*j} \leftarrow (Mh_{*,j} << 1) \vee \neg(Xv_{*j} \vee\ (Ph_{*j} << 1))$
$Mv_{*j} \leftarrow (Ph_{*j} << 1) \wedge Xv_{*j}$

A similar attempt with the for first inner loop leads to a problem:

$Xh_{*j} \leftarrow Eq_{*j} \vee (Mh_{*j} << 1)$
$Ph_{*j} \leftarrow Mv_{*,j-1} \vee \neg(Xh_{*j} \vee\ Pv_{*,j-1})$
$Mh_{*j} \leftarrow Pv_{*,j-1} \wedge Xh_{*j}$

Now the vector Mh_{*j} is used in computing Xh_{*j} before Mh_{*j} itself is computed! Changing the order does not help, because Xh_{*j} is needed to compute Mh_{*j}.

To get out of this dependency loop, we compute Xh_{*j} without Mh_{*j} using only Eq_{*j} and $Pv_{*,j-1}$ which are already available when we compute Xh_{*j}.

Lemma 3.19: $Xh_{ij} = \exists \ell \in [1,i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1})$.

Proof. We use induction on i.

Basis $i = 1$: The right-hand side reduces to Eq_{1j}, because $\ell = 1$. By Lemma 3.18, $Xh_{1j} = Eq_{1j} \vee Mh_{0j}$, which is Eq_{1j} because $Mh_{0j} = 0$ for all j.

Induction step: The induction assumption is that $Xh_{i-1,j}$ is as claimed. Now we have

$$\begin{aligned}
&\exists \ell \in [1,i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge \exists \ell \in [1,i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-2] : Pv_{x,j-1})) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge Xh_{i-1,j}) \qquad \text{(ind. assump.)} \\
&= Eq_{ij} \vee Mh_{i-1,j} \qquad \text{(Lemma 3.18)} \\
&= Xh_{ij} \qquad \text{(Lemma 3.18)}
\end{aligned}$$

□

At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time, let alone a whole vector Xh_{*j}. However, it can be done, but we need more bit operations:

- Let $\veebar$ denote the xor-operation: $0 \veebar 1 = 1 \veebar 0 = 1$ and $0 \veebar 0 = 1 \veebar 1 = 0$.
- 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 $0001 + 0111 = 1000$.

In the following, for a bitvector B, we will 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.

Lemma 3.20: Denote $X = Xh_{*j}$, $E = Eq_{*j}$, $P = Pv_{*,j-1}$ and let $Y = (((E \wedge P) + P) \veebar P) \vee E$. Then $X = Y$.

Proof. By Lemma 3.19, $X[i] = 1$ iff and only if

a) $E[i] = 1$ or
b) $\exists \ell \in [1,i] : E[\ell \ldots i] = 00\cdots01 \wedge P[\ell \ldots i-1] = 11\cdots1$.

and $X[i] = 0$ iff and only if

c) $E_{1\ldots i} = 00\cdots0$ or
d) $\exists \ell \in [1,i] : E[\ell \ldots i] = 00\cdots01 \wedge P[\ell \ldots i-1] \neq 11\cdots1$.

We prove that $Y[i] = X[i]$ in all of these cases:

a) The definition of Y ends with "$\vee E$" which ensures that $Y[i] = 1$ in this case.

b) The following calculation shows that $Y[i] = 1$ in this case:

$$\begin{aligned}
& \qquad\ \ i \quad\ \ \ell \\
E[\ell \ldots i] &= 00\ldots01 \\
P[\ell \ldots i] &= \mathtt{b}1\ldots11 \\
(E \wedge P)[\ell \ldots i] &= 00\ldots01 \\
((E \wedge P) + P)[\ell \ldots i] &= \bar{\mathtt{b}}0\ldots0\mathtt{c} \\
(((E \wedge P) + P) \veebar P)[\ell \ldots i] &= 11\ldots1\bar{\mathtt{c}} \\
Y = ((((E \wedge P) + P) \veebar P) \vee E)[\ell \ldots i] &= 11\ldots11
\end{aligned}$$

where $\mathtt{b}$ is the unknown bit $P[i]$, $\mathtt{c}$ is the possible carry bit coming from the summation of bits $1\ldots,\ell-1$, and $\bar{\mathtt{b}}$ and $\bar{\mathtt{c}}$ are their negations.

c) Because for all bitvectors B, $0 \wedge B = 0$ ja $0 + B = B$, we get $Y = (((0 \wedge P) + P) \veebar P) \vee 0 = (P \veebar P) \vee 0 = 0$.

d) Consider the calculation in case b). A key point there is that the carry bit in the summation travels from position ℓ to i and produces $\bar{\mathtt{b}}$ to position i. The difference in this case is that at least one bit $P[k]$, $\ell \leq k < i$, is zero, which stops the carry at position k. Thus $((E \wedge P) + P)[i] = \mathtt{b}$ and $Y[i] = (b \veebar b) \vee 0 = 0$.

□

As a final detail, we compute the bottom row values g_{mj} using the equalities $g_{m0} = m$ ja $g_{mj} = g_{m,j-1} + \Delta h_{mj}$.

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

for $c \in \Sigma$ do $B[c] \leftarrow 0^m$
for $i \leftarrow 1$ to m do $B[P[i]][i] = 1$
$Pv \leftarrow 1^m;\ Mv \leftarrow 0;\ g \leftarrow m$
for $j \leftarrow 1$ to n do
  $Eq \leftarrow B[T[j]]$
  $Xh \leftarrow (((Eq \wedge Pv) + Pv) \veebar Pv) \vee Eq$
  $Ph \leftarrow Mv \vee \neg(Xh \vee Pv)$
  $Mh \leftarrow Pv \wedge Xh$
  $Xv \leftarrow Eq \vee Mv$
  $Pv \leftarrow (Mh << 1) \vee \neg(Xv \vee (Ph << 1))$
  $Mv \leftarrow (Ph << 1) \wedge Xv$
  $g \leftarrow g + Ph[m] - Mh[m]$
  if $g \leq k$ then output j

On an integer alphabet, when $m \leq w$:

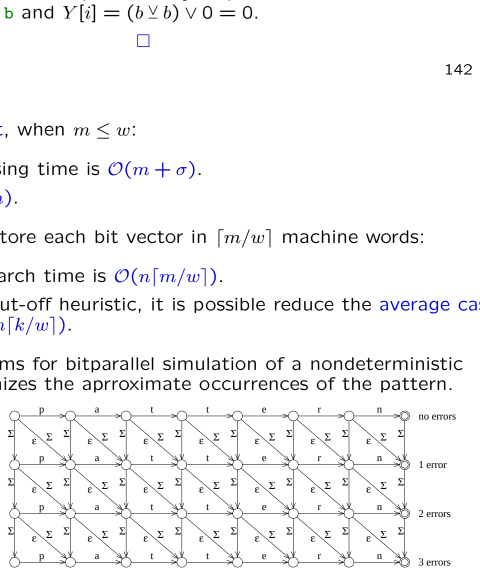
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

When $m > w$, we can store each bit vector in $\lceil m/w \rceil$ machine words:

- The worst case search time is $\mathcal{O}(n\lceil m/w \rceil)$.
- Using Ukkonen's cut-off heuristic, it is possible reduce the average case search time to $\mathcal{O}(n\lceil k/w \rceil)$.

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

Example 3.22:
$P = \mathtt{pattern}$, $k = 3$



Another way to utilize Lemma 3.15 ($\Delta h_{ij}, \Delta v_{ij} \in \{-1, 0, 1\}$) is to use precomputed tables to process multiple matrix cells at a time.

- There are at most 3^m different columns. Thus there exists a deterministic automaton with 3^m states and $\sigma 3^m$ transitions that can find all approximate occurrences in $\mathcal{O}(n)$ 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 $\mathcal{O}(\log_\sigma n)$ characters at a time and $\mathcal{O}(\log_\sigma^2 n)$ matrix cells at a time using lookup tables of size $\mathcal{O}(n)$. This gives time complexity $\mathcal{O}(mn/\log_\sigma^2 n)$.
- 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 following lemma shows the property used by the Baeza-Yates–Perleberg algorithm and proves that it satisfies the first condition.

Lemma 3.23: Let $P_1 P_2 \dots P_{k+1} = P$ be a partitioning of the pattern P into $k+1$ nonempty factors. Any string S with $ed(P,S) \le k$ contains P_i as a factor for some $i \in [1..k+1]$.

Proof. Each single symbol edit operation can change at most one of the pattern factors P_i. Thus any set of at most k edit operations leaves at least one of the factors untouched. □

The algorithm has two phases:

Filtration: Search the text T for exact occurrences of the pattern factors P_i. Using the Aho–Corasick algorithm this takes $\mathcal{O}(n)$ time for a constant alphabet.

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

The worst case time complexity is $\mathcal{O}(m^2 n)$, which can be reduced to $\mathcal{O}(mn)$ 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 $r = \lfloor m/(k+1) \rfloor$.
- The expected number of exact occurrences of a random string of length r in a random text of length n is at most n/σ^r.
- The expected total verification time is at most
$$\mathcal{O}\left(\frac{m^2(k+1)n}{\sigma^r}\right) \le \mathcal{O}\left(\frac{m^3 n}{\sigma^r}\right) .$$
This is $\mathcal{O}(n)$ if $r \ge 3\log_\sigma m$.
- The condition $r \ge 3\log_\sigma m$ is satisfied when $(k+1) \le m/(3\log_\sigma m + 1)$.

Theorem 3.24: The average case time complexity of the Baeza-Yates–Perleberg algorithm is $\mathcal{O}(n)$ when $k \le m/(3\log_\sigma m + 1) - 1$.

Many variations of the algorithm have been suggested:

- The filtration can be done with a different multiple exact string matching algorithm.
- The verification time can be reduced using a technique called hierarchical verification.
- The pattern can be partitioned into fewer than $k+1$ pieces, which are searched allowing a small number of errors.

A lower bound on the average case time complexity is $\Omega(n(k + \log_\sigma m)/m)$, and there exists a filtering algorithm matching this bound.

Summary: Approximate String Matching

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

- Basic dynamic programming time complexity is $\mathcal{O}(mn)$. The time complexity can be improved to $\mathcal{O}(kn)$ using diagonal monotonicity, and to $\mathcal{O}(n\lceil m/w \rceil)$ 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.