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

• Time and space complexity is $O(mn)$ on general 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 $i ∈ [1..m]$, and every $j ∈ [1..n]$, $g_{ij} = g_{i-1,j-1} + 1$.
Proof. By definition, $g_{ij} ≤ g_{i-1,j-1} + \delta(P[i], T[j]) ≤ g_{i-1,j-1} + 1$. We show that $g_{ij} ≥ g_{i-1,j-1}$ by induction on $i + j$.
   The induction assumption is that $g_{pq} ≥ g_{q-1,p-1}$ when $p ∈ [1..m]$, $q ∈ [1..n]$ 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 $g_{ij} ≥ g_{i-1,j-1}$.
   2. $g_{ij} = g_{i,j-1} + 1$ and $i + 1$. Then
      \[ g_{ij} = g_{i-1,j-1} + 1 \geq g_{i,j-1} + 1 \geq g_{i-1,j-1} \]
   3. $g_{ij} = g_{i,j-1} + 1$ and $j > 1$. Then
      \[ g_{ij} = g_{i-1,j} + 1 \geq g_{i,j-2} + 1 \geq g_{i-1,j-1} \]
   4. $g_{ij} = g_{i-1,j} + 1$ and $i = 1$. Then $g_{ij} = 0 + 1 > g_{0,j-1}$.
   5. $g_{ij} = g_{i,j-1} + 1$ and $j = 1$. Then $g_{ij} = 1 + 1 = (i - 1) + 2 = g_{i-1,j-1} + 2$, which cannot be true. Thus this case can never happen.

The row on the current column corresponding to the smallest undiscarded diagonal 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. $top ← \min\{k + 1, m\}$
2. for $i ← 0$ to $top$ do $g_{0,i} ← i$
3. for $j ← 1$ to $n$ do $g_{0,j} ← 0$
4. for $j ← 1$ to $n$ do
   (5) for $i ← 1$ to $top$ do
   (6) $g_{ij} ← \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
   (7) while $g_{0,j} > k$ and top < top - 1
   (8) if top = $m$ then output $j$
   (9) else top ← top + 1; $g_{0,j} ← k + 1$

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: $\Delta g_{ij} = g_{ij} - g_{i-1,j}$
Horizontal delta: $\Delta h_{ij} = g_{ij} - g_{i,j-1}$
Diagonal delta: $\Delta d_{ij} = g_{ij} - g_{i-1,j-1}$
Because $g_{00} = g_{0j} = 0$,
\[ g_{ij} = \Delta v_{ij} + \Delta r_{ij} + \Delta t_{ij} \]
Because of diagonal monotonicity, $\Delta d_{ij} ∈ \{0, 1\}$ 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: $\Delta h_{ij}, \Delta v_{ij} ∈ \{-1, 0, 1\}$ for every $i, j$ that they are defined for.

The proof is left as an exercise.

Ukkonen’s Cut-off Heuristic
We can speed up the algorithm using the diagonal monotonicity of the matrix $(g_{ij})$:
A diagonal $d$, $-m < d < n$, 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.

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

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.

Example 3.16: ‘-‘ means 1, ‘=‘ means 0 and ‘+‘ means 1

The proof is left as an exercise.
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 $\Delta v^m = \Delta v_{i-1,j}$, $\Delta h^m = \Delta h_{i-1,j}$ and $E_{\theta j} = 1 - \delta(P[i],T[j])$.
- **Output** is $\Delta v^{out} = \Delta v_{i,j}$ and $\Delta h^{out} = \Delta h_{i,j}$.

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

The computation rule is defined by the following result.

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

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

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

$$= g_{i-1,j} + 1 - \min\{E_{\theta j}, 1 + \Delta a^m + 1, 1 + \Delta h^m\}$$

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

Clearly $\Delta d = \Delta v^m + \Delta a^{out} = \Delta h^m + \Delta h^{out}$

Thus $\Delta v^{out} = \Delta d - \Delta h^{out}$ and $\Delta h^{out} = \Delta d - \Delta v^m$.

Now the computation rules can be expressed as follows.

**Lemma 3.18**: $Pv^{out} = Mh^m \lor -Xv \lor Ph^m$, $Mv^{out} = Ph^m \land Xv$ $Ph^{out} = (Xh \lor \exists v \lor Xv) \land Ph^m$, $Mh^{out} = Ph^m \land Xh$

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

By Lemma 3.17,

$$Pv^{out} = (-\Delta d \land Mh^m) \lor (-\Delta h \land Ph^m)$$

$$Mv^{out} = (-\Delta d \land Ph^m) \lor (-\Delta h \land Mh^m)$$

Because $\Delta d = (-E_{\theta j} \lor Mh^m) = -Xv \lor \exists v \lor Xv$, $Xv \land \exists v \lor Xv$ = $Mh^m \land -Xv \lor \exists v \lor Xv$ $\land Ph^m = (Xh \lor \exists v \lor Xv) \lor Ph^m = Ph^m \land Xh$ $Mh^{out} = (Xh \lor \exists v \lor Xv) \lor Ph^m = (Mh^m \land Ph^m) \lor Xh \lor Ph^m$

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

To obtain a bitparallel algorithm, the columns $Pv_{v,i}$, $Mv_{v,i}$, $Xv_{v,i}$, $Ph_{v,i}$, $Mh_{i,j}$, $H_{i,j}$ and $Eq_{i,j}$ are stored in bitvectors.

Now the second inner loop can be replaced with the code

$$Xv_{v,i} \leftarrow Eq_{v,i} \lor Mh_{i-1,j}$$

$$Ph_{v,i} \leftarrow (Mh_{i-1,j} < 1) \lor (-Xv_{v,i} \lor (Ph_{v,i} < 1))$$

The similar process can be applied with the for first inner loop leads to a problem:

Xh_{0,i} \leftarrow Eq_{v,i} \lor (Mh_{i-1,j} < 1)

Ph_{v,i} \leftarrow (Mh_{i-1,j} < 1) \lor (-Xh_{0,i} \lor (Ph_{v,i} < 1))

Now the vector $H_{i,j}$ is used in computing $Xh_{0,i}$ before $Mh_{i,j}$ itself is computed! Changing the order does not help, because $Xh_{0,i}$ needed to compute $Mh_{i,j}$.

To get out of this dependency loop, we compute $Xh_{0,i}$ without $Mh_{i,j}$ using only $Eq_{v,i}$ and $Pv_{v,i-1}$ which are already available when we compute $Xh_{0,i}$.

At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time, let alone a whole vector $Xh_{0,i}$. However, it can be done, but we need more bit operations.

- Let $\forall$ denote the xor-operation: $0 \lor 1 = 1 \land 0 = 1 \lor 0 = 0$. 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.

The opposite of the bits reflects the interpretation as an integer.
Lemma 3.20: Denote $X = X_{h^j}$, $E = E_{q^j}$, $P = P_{v^j}$, and let $Y = (((E \land P) + P) \land P) \land P \lor E$. Then $X = Y$.

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

- $E[i] = 1$ or
- $E[i] = 0$, $E[i] = 00 \cdots 01$ and $P[i] = 01 \cdots 1$.

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

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

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

- $E[\ldots i] = 00 \cdots 01$ and $P[\ldots i] = 01 \cdots 1$.

□

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 $P$
(1) for $c \in \Sigma$ do $B[c] \leftarrow 0$
(2) for $i \leftarrow 1$ to $n$ do $B[P[i]] \leftarrow 1$
(3) $Pv \leftarrow 1^m$; $Me \leftarrow 0$; $g \leftarrow m$
(4) for $j \leftarrow 1$ to $n$ do
(5) $Eq \leftarrow B[P[j]]$
(6) $Xh \leftarrow (((Eq \land Pv) + P) \lor P) \lor Eq$
(7) $Ph \leftarrow Me \lor \neg(Xh \lor P)$
(8) $Mh \leftarrow P \lor Xh$
(9) $Xe \leftarrow Eq \lor Me$
(10) $Pe \leftarrow (Mh < k < 1) \lnot(Xe \lor (Ph < 1))$
(11) $Mv \leftarrow (Ph < k < 1) \lor Xv$
(12) $g \leftarrow g + |Ph[m] - Mh[m]|$
(13) if $g \leq k$ then output $j$

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

- There are at most 3 different columns. Thus there exists a deterministic automaton with 3 states and 3 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 $O(\log n)$ characters at a time and $O(n \log n)$ matrix cells at a time using lookup tables of size $O(n)$. This gives time complexity $O(n \log n / \log^2 n)$.
- A practical variant uses smaller lookup tables to compute multiple entries of a column at a time.

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 $P_1 P_2 \ldots P_{k+1} = P$ be a partitioning of the pattern $P$ into $k + 1$ nonempty factors. Any string $S$ with $cd(S, P) \leq k$ contains $P$, 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.

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.

The Karp–Rabin algorithm is a filtering algorithm for exact string matching. The property we are looking for in that case is having the same fingerprint as the pattern.

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 $P_i$. 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 $O(n m n)$ 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 \( \frac{n}{r} \).
- The expected total verification time is at most \( O\left(\frac{m^2(k+1)n}{\sigma}\right) \leq O\left(\frac{m^3n}{\sigma^2}\right) \).

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 \( k \leq m/(3 \log_2 m + 1) - 1 \).

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 \( \Omega(\frac{n(k + \log_2 m)}{m}) \), 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 \( O(mn) \). The time complexity can be improved to \( O(kn) \) using diagonal monotonicity, and to \( O(n\lceil m/w \rceil) \) using bitparallelism.
- Filtering algorithms can improve average case time complexity and are the fastest in practice when \( k \) is not too large. The partitioning into \( k+1 \) factors is a simple but effective filtering technique.

More advanced techniques have been developed but are not covered here (except in study groups).

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