Approximate String Matching

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

Problem 2.7: 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*:

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

Theorem 2.8: 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:

$$
\begin{aligned}
&g_{00} = 0 = ed(\epsilon, \epsilon)\\
&g_{0j} = 0 = ed(\epsilon, \epsilon) = ed(\epsilon, T(j-0..j]) \qquad (\text{min at } \ell = 0)\\
&g_{i0} = i = ed(P[1..i], \epsilon) = ed(P[1..i], T(0-0..0]) \quad (0 \leq \ell \leq j = 0)
\end{aligned}
$$

Induction step: Essentially the same as in the proof of Theorem 2.2.

Example 2.9: $P = \texttt{match}$, $T = \texttt{remachine}$, $k = 1$

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 2.10: 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* $i \leftarrow 0$ *to* m *do* $g_{i0} \leftarrow i$
(2) *for* $j \leftarrow 1$ *to* n *do* $g_{0j} \leftarrow 0$
(3) *for* $j \leftarrow 1$ *to* n *do*
(4)   *for* $i \leftarrow 1$ *to* m *do*
(5) $g_{ij} \leftarrow \min\{g_{i-1,j-1} + \delta(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1\}$
(6)   *if* $q_{mj} \leq k$ *then* output j

- Time and space complexity is $\mathcal{O}(mn)$.
- The space complexity can be reduced to $\mathcal{O}(m)$ by storing only one column as in Algorithm 2.5.

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 increasing*.

Example 2.11: 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

More specifically, we have the following property.

Lemma 2.12: 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. $i = 1$. Then $g_{ij} \geq 0 = g_{i-1,j-1}$.

$g_{ij} = g_{i,j-1} + 1$ and $j = 1$ is not possible because $g_{i,1} \leq g_{i-1,0} + 1 < g_{i,0} + 1$. $\square$

We can reduce computation using diagonal monotonicity:

- Whenever the value on a column 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.

Example 2.13: $P = \texttt{match}$, $T = \texttt{remachine}$, $k = 1$

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					1	1	2	3		
c						1	2	3		
h							1	2		

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

Algorithm 2.14: 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

for $i \leftarrow 0$ *to* m *do* $g_{i0} \leftarrow i$
for $j \leftarrow 1$ *to* n *do* $g_{0j} \leftarrow 0$
$top \leftarrow \min(k+1, m)$
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 $top \leftarrow top + 1$

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

- The worst case time complexity is still $\mathcal{O}(mn)$.
- 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: $\Delta d_{ij} = g_{ij} - g_{i-1,j}$

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

The proof is left as an exercise.

Example 2.16: '−' 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 $\Delta v^{\text{in}} = \Delta v_{i,j-1}$, $\Delta h^{\text{in}} = \Delta h_{i,j-1}$ 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 computation rule is defined by the following result.

Lemma 2.17: 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 logical operations ($\wedge$, $\vee$, $\neg$).

Now the computation rules can be expressed as follows.

Lemma 2.18:

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

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 2.17,

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

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

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

In the last step, we used the fact that Mh^{in} and Ph^{in} cannot be 1 simultaneously. $\square$

According to Lemma 2.18, 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 bit*parallel* 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 2.19: $Xh_{ij} = \exists \ell \in [1, i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1})$.

Proof. We use induction on i.

Basis $i = 1$: The right-hand side reduces to Eq_{1j}, because $\ell = 1$. By Lemma 2.18, $Xh_{1j} = Eq_{1j} \vee Mh_{0j}$, 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

$$\begin{aligned}
&\exists \ell \in [1, i] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee \exists \ell \in [1, i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-1] : Pv_{x,j-1}) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge \exists \ell \in [1, i-1] : Eq_{\ell j} \wedge (\forall x \in [\ell, i-2] : Pv_{x,j-1})) \\
&= Eq_{ij} \vee (Pv_{i-1,j-1} \wedge Xh_{i-1,j}) \qquad \text{(ind. assump.)} \\
&= Eq_{ij} \vee Mh_{i-1,j} \qquad \text{(Lemma 2.18)} \\
&= Xh_{ij} \qquad \text{(Lemma 2.18)}
\end{aligned}$$

□

At first sight, we cannot use Lemma 2.19 to compute even a single bit in constant time, not to mention 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 2.20: Denote $X = Xh_{*j}$, $E = Eq_{*j}$, $P = Pv_{*,j-1}$ ja olkoon $Y = (((E \wedge P) + P) \veebar P) \vee E$. Then $X = Y$.

Proof. By Lemma 2.19, $X[i] = 1$ iff and only if

a) $E[i] = 1$ or

b) $\exists \ell \in [1, i] : E[\ell \ldots i] = 00 \cdots 01 \wedge P[\ell \ldots i-1] = 11 \cdots 1$.

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

c) $E_{1\ldots i} = 00 \cdots 0$ or

d) $\exists \ell \in [1, i] : E[\ell \ldots i] = 00 \cdots 01 \wedge P[\ell \ldots i-1] \neq 11 \cdots 1$.

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:

$$
\begin{array}{rl}
 & \quad i \qquad \ell \\
E[\ell \ldots i] = & \texttt{00...01} \\
P[\ell \ldots i] = & \texttt{b1...11} \\
(E \wedge P)[\ell \ldots i] = & \texttt{00...01} \\
((E \wedge P) + P)[\ell \ldots i] = & \bar{\texttt{b}}\texttt{0...0c} \\
(((E \wedge P) + P) \veebar P)[\ell \ldots i] = & \texttt{11...1}\bar{\texttt{c}} \\
Y = ((((E \wedge P) + P) \veebar P) \vee E)[\ell \ldots i] = & \texttt{11...11}
\end{array}
$$

where $\texttt{b}$ is the unknown bit $P[i]$, $\texttt{c}$ is the possible carry bit coming from the summation of bits $1 \ldots, \ell - 1$, and $\bar{\texttt{b}}$ and $\bar{\texttt{c}}$ 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 $\bar{\texttt{b}}$ to position i. The difference in this case is that at least one bit $P[k]$, $\ell \le k < i$, is zero, which stops the carry at position k. Thus $((E \wedge P) + P)[i] = \texttt{b}$ and $Y[i] = 0$.

□

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 2.21: 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 ∈ Σ do B[c] ← 0^m
for i ← 1 to m do B[P[i]][i] = 1
Pv ← 1^m; Mv ← 0; g ← m
for j ← 1 to n do
    Eq ← B[T[j]]
    Xh ← (((Eq ∧ Pv) + Pv) ⊻ Pv) ∨ Eq;
    Ph ← Mv ∨ ¬(Xh ∨ Pv)
    Mh ← Pv ∧ Xh;
    Xv ← Eq ∨ Mv
    Pv ← (Mh << 1) ∨ ¬(Xv ∨ (Ph << 1))
    Mv ← (Ph << 1) ∧ Xv
    g ← g + Ph[m] − Mh[m]
    if g ≤ k then output j
```