Example 3.3: $A =$ `ballad`, $B =$ `handball`

d		h	a	n	d	b	a	l	l
	0	1	2	3	4	5	6	7	8
b	1	1	2	3	4	4	5	6	7
a	2	2	1	2	3	4	4	5	6
l	3	3	2	2	3	4	5	4	5
l	4	4	3	3	3	4	5	5	4
a	5	5	4	4	4	4	4	5	5
d	6	6	5	5	4	5	5	5	6

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

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:

$$d_{00} = 0 = ed(\epsilon, \epsilon)$$
$$d_{i0} = i = ed(A_i, \epsilon) \quad (i \text{ deletions})$$
$$d_{0j} = j = ed(\epsilon, B_j) \quad (j \text{ insertions})$$

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

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

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

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

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

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}. □

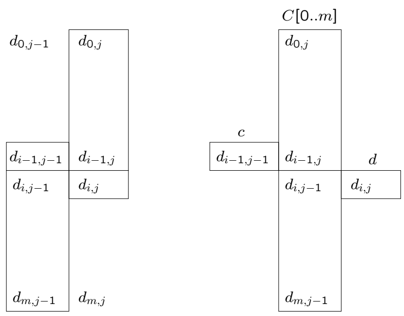
The recurrence gives directly a dynamic programming algorithm for computing the edit distance.

Algorithm 3.4: Edit distance
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$
(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) $d_{ij} \leftarrow \min\{d_{i-1,j-1} + \delta(A[i], B[j]), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
(6) return d_{mn}

The time and space complexity is $\mathcal{O}(mn)$.

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:



Algorithm 3.5: Edit distance in $\mathcal{O}(m)$ space
Input: strings $A[1..m]$ and $B[1..n]$
Output: $ed(A, B)$
(1) for $i \leftarrow 0$ to m do $C[i] \leftarrow i$
(2) for $j \leftarrow 1$ to n do
(3) $c \leftarrow C[0]$; $C[0] \leftarrow j$
(4) for $i \leftarrow 1$ to m do
(5) $d \leftarrow \min\{c + \delta(A[i], B[j]), C[i-1] + 1, C[i] + 1\}$
(6) $c \leftarrow C[i]$
(7) $C[i] \leftarrow d$
(8) return $C[m]$

- Note that because $ed(A, B) = ed(B, A)$ (exercise), we can assume that $m \le n$.

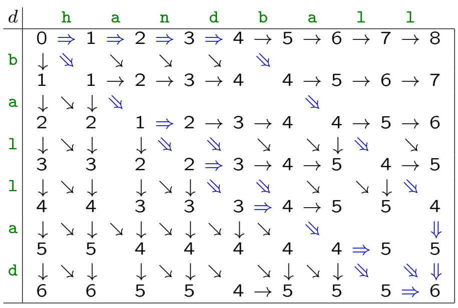
It is also possible to find optimal edit sequences and alignments from the matrix d_{ij}.

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

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

Example 3.6: $A =$ `ballad`, $B =$ `handball`



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

```
IIIINNNNDD   SNISSNIS   SNSSINSI
----ballad   ba-lla-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 $k \ge 0$, report all positions $j \in [1..m]$ such that $ed(P, T(j - \ell...j]) \le k$ for some $\ell \ge 0$.

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:

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

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

$$g_{ij} = \min\{ed(P[1..i], T(j-\ell...j]) \mid 0 \le \ell \le j\} .$$

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

Proof. We use induction with respect to $i + j$.

Basis:
$$g_{00} = 0 = ed(\epsilon, \epsilon)$$
$$g_{0j} = 0 = ed(\epsilon, \epsilon) = ed(\epsilon, T(j-0..j]) \qquad (\text{min at } \ell = 0)$$
$$g_{i0} = i = ed(P[1..i], \epsilon) = ed(P[1..i], T(0-0..0]) \quad (0 \le \ell \le j = 0)$$

Induction step: Essentially the same as in the proof of Theorem 3.2.

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

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 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

```
for i ← 0 to m do g_i0 ← i
for j ← 1 to n do g_0j ← 0
for j ← 1 to n do
    for i ← 1 to m do
        g_ij ← min{g_{i-1,j-1} + δ(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1}
    if q_mj ≤ k then output j
```

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

Ukkonen's Cut-off Heuristic

We can speed up the algorithm using the diagonal monotonicity of the matrix (g_{ij}):

> A diagonal d, $-m \le d \le n$, 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
	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 3.12: 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, $g_{ij} \le g_{i-1,j-1} + \delta(P[i], T[j]) \le g_{i-1,j-1} + 1$. We show that $g_{ij} \ge g_{i-1,j-1}$ by induction on $i + j$.

The induction assumption is that $g_{pq} \ge g_{p-1,q-1}$ 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 $g_{ij} \ge g_{i-1,j-1}$.
2. $g_{ij} = g_{i-1,j} + 1$ and $i > 1$. Then
$$g_{ij} = g_{i-1,j} + 1 \overset{\text{ind. assump.}}{\ge} g_{i-2,j-1} + 1 \overset{\text{definition}}{\ge} g_{i-1,j-1}$$
3. $g_{ij} = g_{i,j-1} + 1$ and $j > 1$. Then
$$g_{ij} = g_{i,j-1} + 1 \overset{\text{ind. assump.}}{\ge} g_{i-1,j-2} + 1 \overset{\text{definition}}{\ge} g_{i-1,j-1}$$
4. $g_{ij} = g_{i-1,j} + 1$ and $i = 1$. Then $g_{ij} = 0 + 1 > 0 = g_{i-1,j-1}$.
5. $g_{ij} = g_{i,j-1} + 1$ 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. □

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.

Example 3.13: $P = \texttt{match}$, $T = \texttt{remachine}$, $k = 1$

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	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 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

```
for i ← 0 to min(k, m) do g_i0 ← i
for j ← 1 to n do g_0j ← 0
top ← min(k + 1, m)
for j ← 1 to n do
    for i ← 1 to top do
        g_ij ← min{g_{i-1,j-1} + δ(A[i], B[j]), g_{i-1,j} + 1, g_{i,j-1} + 1}
    while g_{top,j} > k do top ← top − 1
    if top = m then output j
    else top ← top + 1
```

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

- The worst case time complexity is still $\mathcal{O}(mn)$ 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 $\Delta v^{\mathsf{in}} = \Delta v_{i,j-1}$, $\Delta h^{\mathsf{in}} = \Delta h_{i,j-1}$ and $Eq_{ij} = 1 - \delta(P[i], T[j])$.
- Output is $\Delta v^{\mathsf{out}} = \Delta v_{i,j}$ and $\Delta h^{\mathsf{out}} = \Delta h_{i,j}$.

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

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 $\Delta v^{\mathsf{in}} = -1$ or $\Delta h^{\mathsf{in}} = -1$, then $\Delta d = 0$, $\Delta v^{\mathsf{out}} = -\Delta h^{\mathsf{in}}$ and $\Delta h^{\mathsf{out}} = -\Delta v^{\mathsf{in}}$.
Otherwise $\Delta d = 1$, $\Delta v^{\mathsf{out}} = 1 - \Delta h^{\mathsf{in}}$ and $\Delta h^{\mathsf{out}} = 1 - \Delta v^{\mathsf{in}}$.

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

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

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

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

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 (∧, ∨, ¬).

Now the computation rules can be expressed as follows.

Lemma 3.18:

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

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

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

By Lemma 3.17,

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

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

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

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.
□

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

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

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

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

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

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:

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

where b is the unknown bit $P[i]$, c is the possible carry bit coming from the summation of bits $1 \ldots, \ell - 1$, and $\bar{\text{b}}$ and $\bar{\text{c}}$ 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 $\bar{\text{b}}$ 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 $((E \wedge P) + P)[i] =$ 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 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 ∈ 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
```


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

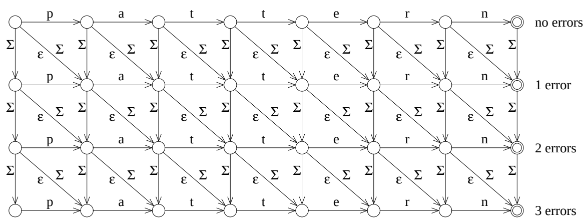
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

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 based on bitparallel simulation of a nondeterministic automaton.

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



- The algorithm of Wu and Manber uses a bit vector for each row. It can be seen as an extension of Shift-And. The search time complexity is $\mathcal{O}(kn \lceil m/w \rceil)$.
- The algorithm of Baeza-Yates and Navarro uses a bit vector for each diagonal, packed into one long bitvector. The search time complexity is $\mathcal{O}(n \lceil km/w \rceil)$.