Example 3.3: $A = \text{ballad}, B = \text{handball}$

<table>
<thead>
<tr>
<th>d</th>
<th>b</th>
<th>a</th>
<th>h</th>
<th>n</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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:**

- $d_0 = 0 = d(e, \epsilon)$
- $d_0 = i = ed(A, \epsilon)$ (i deletions)
- $d_0 = j = ed(\epsilon, B_j)$ (j insertions)

**Induction step:** We show that the claim holds for $d_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$.

By induction assumption, $d_{ij} = ed(A_i, B_j)$ 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:** $E_{ij} = E_{i-1,j}N$ or $E_{ij} = E_{i,j-1}S$ and $ed(A_i, B_j) = ed(A_{i-1}, B_j-1) + S(A[i], B[j])$.
- **I:** $E_{ij} = E_{i,j-1}I$ and $ed(A_i, B_j) = ed(A_{i-1}, B_j) + 1 = d_{i-1,j} + 1$.
- **D:** $E_{ij} = E_{i-1,j}D$ and $ed(A_i, B_j) = ed(A_{i-1}, B_j) + 1 = d_{i,j-1} + 1$.

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 $d_{ij}$. $\square$

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 $j$ and the already computed top part of column $i$. We store these in an array $C[0..m]$ and variables $c$ and $d$ as shown below:

```
<table>
<thead>
<tr>
<th>d_{i-1,j}</th>
<th>d_{i,j}</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_{i-1,j}</td>
<td>d_{i,j}</td>
<td>c</td>
</tr>
<tr>
<td>d_{i-1,j}</td>
<td>d_{i,j}</td>
<td>c</td>
</tr>
</tbody>
</table>
```

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

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

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 $k \geq 0$, report all positions $j \in [1..n]$ such that $ed(T, P) \leq k$ for some $i \geq 0$.

The factor $T(i-1) \cdots (i-j)$ 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.

Example 3.6: $A = \text{ballad}, B = \text{handball}$

<table>
<thead>
<tr>
<th>d</th>
<th>b</th>
<th>a</th>
<th>l</th>
<th>n</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>l</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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

```
IIIINNNN  SSSNNN  SNNNI
-----ballad  ba-lla-d  ball-ad-
handball--  hand-ball  handball
```
Define the values \( g_{ij} \) with the recurrence:
\[
g_{0j} = 0, \quad 0 \leq j \leq n,
\]
\[
g_{i0} = i, \quad 1 \leq i \leq m,
\]
\[
g_{ij} = \min\{g_{i-1,j-1} + \delta(P[i], T[j]) \mid 0 \leq \ell \leq j\} + 1.
\]

**Theorem 3.8:** For all \( 0 \leq i \leq m, \ 0 \leq j \leq n \):
\[
g_{ij} = \min\{\delta(P[i], T(j - \ell...j)) \mid 0 \leq \ell \leq j\}.
\]

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

**Example 3.9:** \( P = \text{match}, \ T = \text{remachine}, \ k = 1 \)

<table>
<thead>
<tr>
<th>( g )</th>
<th>( \text{r e m a c h i n e} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

One occurrence ending at position 6.

**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 non-decreasing.

**Example 3.11:** Diagonals -3 and 2.

<table>
<thead>
<tr>
<th>( g )</th>
<th>( \text{m a t c h} )</th>
<th>( \text{h} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

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:** \( P = \text{match}, \ T = \text{remachine}, \ k = 1 \)

<table>
<thead>
<tr>
<th>( g )</th>
<th>( \text{r e m a c h i n e} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Proof.** We use induction with respect to \( i + j \).

**Basis:**
\[
g_{00} = 0 = \text{ed}(\varepsilon, \varepsilon), \quad g_{0j} = 0 = \text{ed}(\varepsilon, \text{T}(j - 0...j)) \quad (\text{min at } \ell = 0)
\]
\[
g_{ij} = 1 = \text{ed}(P[i], T(0 - 0...0)) \quad (0 \leq \ell \leq j = 0)
\]

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

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

- Time and space complexity is \( O(mn) \) on ordered alphabet.

- The space complexity can be reduced to \( O(m) \) by storing only one column as in Algorithm 3.5.

More specifically, we have the following property:

**Lemma 3.12:** For every \( i \in [1..m] \) and every \( j \in [1..n] \),
\[
g_{ij} = g_{i,j-1} \text{ or } g_{i-1,j+1} + 1.
\]

**Proof.** By definition, \( g_{ij} \geq g_{i,j-1} + \delta(P[i], T[j]) \geq g_{i,j-1} + 1 \).

We show that \( g_{ij} \geq g_{i,j-1} + 1 \) by induction on \( i + j \).

The induction assumption is that \( g_{ij} \geq g_{i,j-1} + 1 \) when \( p \in [1..m], q \in [1..n] \) and \( p + q < i + j \). At least one of the following holds:

1. \( g_{ij} = g_{i,j-1} + 1 \) and \( j \leq 1 \). Then \( g_{ij} \geq g_{i-1,j-1} \).
2. \( g_{ij} = g_{i,j-1} + 1 \) and \( i \leq 1 \). Then \( g_{ij} \geq g_{i-1,j-1} \).
3. \( g_{ij} = g_{i,j-1} + 1 \) and \( j > 1 \). Then \( g_{ij} \geq g_{i-1,j-2} + 1 \).
4. \( g_{ij} = g_{i,j-1} + 1 \) and \( i > 1 \). Then \( g_{ij} = g_{0,j-1} + 1 \).

We have \( g_{ij} = g_{i,j-1} + 1 \) and the smallest undiscarded diagonal is kept in a variable \( \text{top} \).

**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. for \( i \leftarrow 0 \) to \( \min(k, m) \) do \( g_{0i} \leftarrow i \)
2. for \( j \leftarrow 1 \) to \( n \) do \( g_{0j} \leftarrow 0 \)
3. \( \text{top} \leftarrow \min(k + 1, m) \)
4. for \( j \leftarrow 1 \) to \( n \) do
5. while \( g_{\text{top}, j} > k \) do \( \text{top} \leftarrow \text{top} - 1 \)
6. if \( \text{top} = m \) then output \( j \)
7. \( \text{else} \ \text{top} \leftarrow \text{top} + 1 \)
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 \(O(lnn)\). 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

<table>
<thead>
<tr>
<th>m</th>
<th>0 = 0 = 0 = 0 = 0 = 0 = 0 = 0 = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>+ + + + + + + + + + + + + + + + + + + + + + + +</td>
</tr>
<tr>
<td>t</td>
<td>2 = 2 = 2 = 2 = 2 = 2 = 2 = 2 = 2</td>
</tr>
<tr>
<td>c</td>
<td>3 = 3 = 3 = 3 = 3 = 3 = 3 = 3 = 3</td>
</tr>
<tr>
<td>h</td>
<td>+ + + + + + + + + + + + + + + + + + + + + + + + +</td>
</tr>
</tbody>
</table>

The computation rule is defined by the following result.

**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.

**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 \(g_{ij} = i\) ja \(g_{ij} = 0\), \(g_{ij} = \Delta v_{ij} + \Delta v_{ij} + \ldots + \Delta v_{ij}\) \(= i + \Delta h_{ij} + \Delta h_{ij} + \ldots + \Delta h_{ij}\)

Because of diagonal monotonicity, \(\Delta d_{ij} \in \{0, 1\}\) 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 \(g_{i-1,j}, g_{i-1,j-1}, g_{i,j-1}\) and \(s(P[i], T[j])\).
- **Output** is \(g_{ij}\).

In the corresponding bitparallel computation:

- **Input** is \(\Delta v^e = \Delta v_{i-1,j-1}, \Delta h^e = \Delta h_{i-1,j-1}\) and \(E_{g_{ij}} = 1 - \delta(P[i], T[j])\).
- **Output** is \(\Delta v^{out} = \Delta v_{ij}\) and \(\Delta h^{out} = \Delta h_{ij}\).

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:
  - \(P_v = \begin{cases} 1 & \text{if } \Delta v = +1 \\ 0 & \text{otherwise} \end{cases}\)
  - \(M_v = \begin{cases} 1 & \text{if } \Delta v = -1 \\ 0 & \text{otherwise} \end{cases}\)
  - \(P_h = \begin{cases} 1 & \text{if } \Delta h = +1 \\ 0 & \text{otherwise} \end{cases}\)
  - \(M_h = \begin{cases} 1 & \text{if } \Delta h = -1 \\ 0 & \text{otherwise} \end{cases}\)

- We replace arithmetic operations (+, −, min) with Boolean (logical) operations (∨, ∧, ¬).

Now the computation rules can be expressed as follows.

**Lemma 3.18:**

\[
P^e_{v} = M_h^e \vee (X_v \lor P_h^e) \quad M^e_{v} = P_h^e \land X_v
\]

\[
P^e_{h} = M_v^e \vee (X_h \lor P_v^e) \quad M^e_{h} = P_v^e \land X_h
\]

Where \(X_v = Eq \lor M_v^e\) and \(X_h = Eq \lor M_h^e\).

**Proof.** We show the claim for \(P_v\) and \(M_v\) only. \(Ph\) and \(Mh\) are symmetrical.

By Lemma 3.17, \(P_v^{out} = (\neg \Delta d \land M_h^e) \lor (\Delta d \land \neg P_h^e)\)

\[M_v^{out} = (\neg \Delta d \land P_h^e) \lor (\Delta d \land \neg M_h^e)\]

Because \(\Delta d = (\neg Eq \lor M_v^e \lor M_h^e) = (\neg X_v \land M_h^e)\)

\[\neg X_v \land \neg M_h^e\]

\[= M^e_{h} \lor \neg (X_v \lor P_h^e)\]

\[M^e_{v} \lor (X_v \lor M^e_{h}) \land P_h^e = X_v \lor P_h^e\]

All the steps above use just basic laws of Boolean algebra except the last step, where we use the fact that \(M^e_{h} \land P_h^e\) cannot be 1 simultaneously.

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

\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do } \quad P_{v_{ij}} \leftarrow 1, M_{v_{ij}} \leftarrow 0
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do } \quad P_{h_{ij}} \leftarrow 0, M_{h_{ij}} \leftarrow 0
\]

\[
\text{for } i \leftarrow 1 \text{ to } m \text{ do } \quad X_{h_{ij}} \leftarrow Eq_{ij} \lor M_{h_{ij-1}}
\]

\[
\text{for } j \leftarrow 1 \text{ to } n \text{ do } \quad X_{v_{ij}} \leftarrow Eq_{ij} \lor M_{v_{ij-1}}
\]

This is not yet bitparallel though.
To obtain a bitparallel algorithm, the columns \( P_v^*, M_v^*, X_v^*, P_h^*, M_h^*, X_h^* \) and \( E_q^* \) are stored in bitvectors.

Now the second inner loop can be replaced with the code

\[
X_v^* \leftarrow E_q^* \lor X_v^* \quad P_v^* \leftarrow (M_h^* \ll 1) \lor (X_v^* \lor (P_h^* \ll 1))
\]

\[
M_v^* \leftarrow (P_h^* \ll 1) \land X_v^*
\]

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

\[
X_h^* \leftarrow E_q^* \lor (M_h^* \ll 1)
\]

\[
P_h^* \leftarrow (M_h^* \ll 1) \lor (X_v^* \lor P_v^* \ll 1)
\]

\[
M_h^* \leftarrow P_v^* \land X_h^*
\]

Now the vector \( M_h^* \) is used in computing \( X_h^* \) before \( M_h^* \) itself is computed! Changing the order does not help, because \( X_h^* \) is needed to compute \( M_h^* \).

To get out of this dependency loop, we compute \( X_h^* \) without \( M_h^* \) using only \( E_q^* \) and \( P_v^* \), which are already available when we compute \( X_h^* \).

\[\text{Lemma 3.19: } X_{hij} = \exists \ell \in [1, i] : E_{q\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx,j-1}).\]

\[\text{Proof. We use induction on } i.\]

\[\text{Basis } i = 1: \text{ The equality holds trivially.}\]

\[\text{Induction step: The induction assumption is that } X_{h_{i-1,j}} \text{ is as claimed. Now we have:}\]

\[
\exists \ell \in [1, i] : E_{q\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx,j-1}) = E_{q\ell j} \lor \exists \ell \in [1, i - 1] : E_{q\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx,j-1}) = E_{q\ell j} \lor (P_{v_{i-1,j}} \land X_{h_{i-1,j}}) \quad \text{(ind. assump.)}
\]

\[
= E_{q\ell j} \lor M_{h_{i-1,j}} \quad \text{(Lemma 3.18)}
\]

\[
= X_{hij} \quad \text{(Lemma 3.18)}
\]

\[\square\]

At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time, not to mention a whole vector \( B[1..m] \). An example is shown in all of these cases:

\[\text{b) The following calculation shows that } Y[i] = 1 \text{ in this case:}\]

\[
E[\ell...i] = 00...01 \quad P[\ell...i] = 01...11
\]

\[
((E \land P) \lor P)[\ell...i] = 00...00 \quad ((E \land P) \lor P)[\ell...i] = 11...11
\]

\[
Y = (((E \land P) \lor P) \lor P)[\ell...i] = 11...11
\]

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

\[\text{c) Because for all bitvectors } B, 0 \land B = 0 \text{ and } B = B, \text{ we get}\]

\[
Y = (((E \land P) \lor P) \lor P)[\ell...i] = 0 = (P \lor P) \lor P = 0
\]

\[\text{d) Consider the calculation in case b). A key point there is the carry in the summation travels from position } \ell \text{ to } i \text{ and produces } 1 \text{ to position } k. \text{ The difference in this case is that at least one bit } P[k], \ell \leq k < i \text{, is zero, which stops the carry at position } k. \text{ Thus } ((E \land P) \lor P)[i] = b \text{ and } Y[i] = 0.\]

\[\square\]

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

- Pattern preprocessing time is \( O(m + s) \).
- 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 to reduce the average case search time to \( O(n/k|w|) \).

\[\text{Lemma 3.19: } X_{hij} = \exists \ell \in [1, i] : E_{q\ell j} \land (\forall z \in [\ell, i - 1] : P_{vz,j-2}).\]

\[\text{Proof. We use induction on } i.\]

\[\text{Basis } i = 1: \text{ The right-hand side reduces to } E_{q1j}, \text{ because } \ell = 1. \text{ By Lemma 3.18, } X_{h1j} = E_{q1j} \land M_{h0j}, \text{ which is } E_{q1j}, \text{ because } M_{h0j} = 0 \text{ for all } j.\]

\[\text{Induction step: The induction assumption is that } X_{h_{i-1,j}} \text{ is as claimed. Now we have:}\]

\[
\exists \ell \in [1, i] : E_{q\ell j} \land (\forall z \in [\ell, i - 1] : P_{vz,j-1}) = E_{q\ell j} \lor \exists \ell \in [1, i - 1] : E_{q\ell j} \land (\forall z \in [\ell, i - 1] : P_{vz,j-1}) = E_{q\ell j} \lor (P_{v_{i-1,j}} \land X_{h_{i-1,j}}) \quad \text{(ind. assump.)}
\]

\[
= E_{q\ell j} \lor M_{h_{i-1,j}} \quad \text{(Lemma 3.18)}
\]

\[
= X_{hij} \quad \text{(Lemma 3.18)}
\]

\[\square\]

\[\text{Lemma 3.20: Denote } X = X_{h11}, E = E_{q11}, P = P_{v11-1} \text{ and let}\]

\[Y = (((E \land P) \lor P) \lor P) \lor E. \text{ Then } X = Y.\]

\[\text{Proof. By Lemma 3.19, } X[i] = 1 \text{ iff and only if}\]

\[a) \quad E[i] = 1 \text{ or}\]

\[b) \quad \exists \ell \in [1, i] : E[\ell...i] = 00...01 \land P[\ell...i - 1] = 11...1.\]

\[\text{and } Y[i] = 0 \text{ iff and only if}\]

\[c) \quad E_{i-1,j} = 00...00 \text{ or}\]

\[d) \quad \exists \ell \in [1, i] : E[\ell...i] = 00...01 \land P[\ell...i - 1] \neq 11...1.\]

\[\text{We prove that } Y[i] = X[i] \text{ in all of these cases:}\]

\[a) \quad \text{The definition of } Y \text{ ends with } \lor \land \text{ which ensures that } Y[i] = 1 \text{ in this case.}\]

As a final detail, we compute the bottom row values \( g_{mj} \) using the equalities

\[g_{m0} = m \text{ and } g_{mj} = g_{m,j-1} + \Delta h_{mj}.\]

\[\text{Algorithm 3.21: Myers’ bitparallel algorithm}\]

\[\text{Input: text } T[1..n], \text{ pattern } P[1..m], \text{ and integer } k\]

\[\text{Output: end positions of all approximate occurrences of } P\]

\[\text{for } c \in X \text{ do } B[c] \leftarrow 0^m\]

\[\text{for } i \leftarrow 1 \text{ to } m \text{ do } B[P[i]] \leftarrow 1\]

\[\text{for } j \leftarrow 1 \text{ to } n \text{ do}\]

\[E \leftarrow B[T[j]]\]

\[X \leftarrow (((E \land P) \lor P) \lor P) \lor Eq\]

\[Pb \leftarrow Mv \lor \neg (X \lor (Ph \ll 1))\]

\[Mh \leftarrow P \lor Xh\]

\[Xv \leftarrow Eq \lor Mv\]

\[Mv \leftarrow (Mh \ll 1) \land Xh\]

\[g \leftarrow g + Pb[m] - Mh[m]\]

\[\text{if } g \leq k \text{ then output } j\]

\[\square\]

There are also algorithms based on bitparallel simulation of a nondeterministic automaton.

\[\text{Example 3.22: } P = \text{pattern}, k = 3\]