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 \(g_{i0} = i\) and \(g_{0j} = 0\),

\[
g_{ij} = \Delta v_{1j} + \Delta v_{2j} + \cdots + \Delta v_{ij}
= i + \Delta h_{i1} + \Delta h_{i2} + \cdots + \Delta h_{ij}
\]

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.
**Example 3.16:** ‘–’ means $-1$, ‘=’ means 0 and ‘+’ means $+1$

<table>
<thead>
<tr>
<th></th>
<th>r e m a c h i n e</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>+ + + + + = = + + + + + + + + + + +</td>
</tr>
<tr>
<td>a</td>
<td>+ + + + + = + = - = = + + + + + + + +</td>
</tr>
<tr>
<td>t</td>
<td>+ + + + + = + = + + = + = + + + + + + +</td>
</tr>
<tr>
<td>c</td>
<td>+ + + + + = + = + = + = + = + + + + +</td>
</tr>
<tr>
<td>h</td>
<td>+ + + + + = + = + = + = - = - = - = - =</td>
</tr>
</tbody>
</table>

$0 = 0 = 0 = 0 = 0 = 0 = 0 = 0 = 0 = 0$

$1 = 1 = 1 - 0 + 1 = 1 = 1 = 1 = 1 = 1$

$2 = 2 = 2 - 1 - 0 + 1 + 2 = 2 = 2 = 2$

$3 = 3 = 3 - 2 - 1 = 1 + 2 + 3 = 3 = 3$

$4 = 4 = 4 - 3 - 2 - 1 + 2 + 3 + 4 = 4$

$5 = 5 = 5 - 4 - 3 - 2 - 1 + 2 + 3 + 4$

136
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 \( \Delta h_{\text{out}} = \Delta h_{i,j} \).

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 $Eq = 1$ 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

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

$$= g_{i-1,j-1} + \min\{1 - Eq, \Delta v^{\text{in}} + 1, \Delta h^{\text{in}} + 1\}.$$ 

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 $\Delta h^{\text{out}} = \Delta d - \Delta v^{\text{in}}$. □
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:

  \begin{align*}
  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}
\end{align*}

Then

\begin{align*}
\Delta v &= P_v - M_v \\
\Delta h &= P_h - M_h
\end{align*}

- We replace arithmetic operations ($+, -, \text{min}$) with Boolean (logical) operations ($\land, \lor, \lnot$).
Now the computation rules can be expressed as follows.

**Lemma 3.18:**

\[
P_{v_{\text{out}}} = M_{h_{\text{in}}} \lor \neg (X_{v} \lor P_{h_{\text{in}}}) \\
P_{h_{\text{out}}} = M_{v_{\text{in}}} \lor \neg (X_{h} \lor P_{v_{\text{in}}}) \\
M_{v_{\text{out}}} = P_{h_{\text{in}}} \land X_{v} \\
M_{h_{\text{out}}} = P_{v_{\text{in}}} \land X_{h}
\]

where \(X_{v} = Eq \lor M_{v_{\text{in}}}\) and \(X_{h} = Eq \lor M_{h_{\text{in}}}\).

**Proof.** We show the claim for \(P_{v}\) and \(M_{v}\) only. \(P_{h}\) and \(M_{h}\) are symmetrical.

By Lemma 3.17,

\[
P_{v_{\text{out}}} = (-\Delta d \land M_{h_{\text{in}}}) \lor (\Delta d \land \neg P_{h_{\text{in}}}) \\
M_{v_{\text{out}}} = (-\Delta d \land P_{h_{\text{in}}}) \lor (\Delta d \land 0) = -\Delta d \land P_{h_{\text{in}}}
\]

Because \(\Delta d = - (Eq \lor M_{v_{\text{in}}} \lor M_{h_{\text{in}}}) = -(X_{v} \lor M_{h_{\text{in}}}) = -X_{v} \land \neg M_{h_{\text{in}}},\)

\[
P_{v_{\text{out}}} = ((X_{v} \lor M_{h_{\text{in}}}) \land M_{h_{\text{in}}}) \lor (-X_{v} \land \neg M_{h_{\text{in}}} \land \neg P_{h_{\text{in}}}) \\
= M_{h_{\text{in}}} \lor -(X_{v} \lor M_{h_{\text{in}}} \lor P_{h_{\text{in}}}) = M_{h_{\text{in}}} \lor -(X_{v} \lor P_{h_{\text{in}}}) \\
M_{v_{\text{out}}} = (X_{v} \lor M_{h_{\text{in}}}) \land P_{h_{\text{in}}} = (X_{v} \land P_{h_{\text{in}}}) \lor (M_{h_{\text{in}}} \land P_{h_{\text{in}}}) = X_{v} \lor P_{h_{\text{in}}}
\]

All the steps above use just basic laws of Boolean algebra except the last step, where we use the fact that \(M_{h_{\text{in}}}\) and \(P_{h_{\text{in}}}\) cannot be 1 simultaneously. \(\square\)
According to Lemma 3.18, the bit representation of the matrix can be computed as follows.

\[
\begin{align*}
\text{for } i \leftarrow 1 \text{ to } m \text{ do} & \quad P_{v_0} \leftarrow 1; M_{v_0} \leftarrow 0 \\
\text{for } j \leftarrow 1 \text{ to } n \text{ do} & \quad P_{h_{0j}} \leftarrow 0; M_{h_{0j}} \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } m \text{ do} & \quad X_{h_{ij}} \leftarrow \text{Eq}_{ij} \lor M_{h_{i-1,j}} \\
& \quad P_{h_{ij}} \leftarrow M_{v_{i,j-1}} \lor \neg(X_{h_{ij}} \lor P_{v_{i,j-1}}) \\
& \quad M_{h_{ij}} \leftarrow P_{v_{i,j-1}} \land X_{h_{ij}}
\end{align*}
\]

This is not yet bitparallel though.
To obtain a bitparallel algorithm, the columns \( P_{v*j} \), \( M_{v*j} \), \( X_{v*j} \), \( P_{h*j} \), \( M_{h*j} \), \( X_{h*j} \) and \( E_{q*j} \) are stored in bitvectors.

Now the second inner loop can be replaced with the code

\[
\begin{align*}
X_{v*j} & \leftarrow E_{q*j} \lor M_{v*,j-1} \\
P_{v*j} & \leftarrow (M_{h*,j} \ll 1) \lor \neg(X_{v*j} \lor (P_{h*j} \ll 1)) \\
M_{v*j} & \leftarrow (P_{h*j} \ll 1) \land X_{v*j}
\end{align*}
\]

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

\[
\begin{align*}
X_{h*j} & \leftarrow E_{q*j} \lor (M_{h*,j} \ll 1) \\
P_{h*j} & \leftarrow M_{v*,j-1} \lor \neg(X_{h*j} \lor P_{v*,j-1}) \\
M_{h*j} & \leftarrow P_{v*,j-1} \land X_{h*j}
\end{align*}
\]

Now the vector \( M_{h*j} \) is used in computing \( X_{h*j} \) before \( M_{h*j} \) itself is computed! Changing the order does not help, because \( X_{h*j} \) is needed to compute \( M_{h*j} \).

To get out of this dependency loop, we compute \( X_{h*j} \) without \( M_{h*j} \) using only \( E_{q*j} \) and \( P_{v*,j-1} \) which are already available when we compute \( X_{h*j} \).
**Lemma 3.19:** \( X_{h_{ij}} = \exists \ell \in [1, i] : \text{Eq}_{\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx_{i-1,j}}) \).

**Proof.** We use induction on \( i \).

Basis \( i = 1 \): The right-hand side reduces to \( \text{Eq}_{1j} \), because \( \ell = 1 \). By Lemma 3.18, \( X_{h_{1j}} = \text{Eq}_{1j} \lor M_{h_{0j}} \), which is \( \text{Eq}_{1j} \) because \( M_{h_{0j}} = 0 \) for all \( j \).

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

\[
\exists \ell \in [1, i] : \text{Eq}_{\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx_{i-1,j}})
\]

\[
= \text{Eq}_{ij} \lor \exists \ell \in [1, i - 1] : \text{Eq}_{\ell j} \land (\forall x \in [\ell, i - 1] : P_{vx_{i-1,j}})
\]

\[
= \text{Eq}_{ij} \lor (P_{v_{i-1,j-1}} \land \exists \ell \in [1, i - 1] : \text{Eq}_{\ell j} \land (\forall x \in [\ell, i - 2] : P_{vx_{i-2,j}}))
\]

\[
= \text{Eq}_{ij} \lor (P_{v_{i-1,j-1}} \land X_{h_{i-1,j}}) \quad (\text{ind. assump.})
\]

\[
= \text{Eq}_{ij} \lor M_{h_{i-1,j}} \quad (\text{Lemma 3.18})
\]

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

\( \square \)
At first sight, we cannot use Lemma 3.19 to compute even a single bit in constant time, let alone a whole vector $X_{h+j}$. However, it can be done, but we need more bit operations:

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

- 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

$$B = B[1..m] = B[m]B[m−1]...B[1]$$

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 $Y = (((E \land P) + P) \lor P) \lor E$. Then $X = Y$.

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

a) $E[i] = 1$ or

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

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

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

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

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:

\[
\begin{align*}
E[\ell \ldots i] &= 00 \ldots 01 \\
P[\ell \ldots i] &= b1 \ldots 11 \\
(E \land P)[\ell \ldots i] &= 00 \ldots 01 \\
((E \land P) + P)[\ell \ldots i] &= 00 \ldots 01 \\
(((E \land P) + P) \lor P)[\ell \ldots i] &= b1 \ldots 11 \\
Y &= (((E \land P) + P) \lor E)[\ell \ldots i] = 11 \ldots 11
\end{align*}
\]

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 $\overline{b}$ and $\overline{c}$ are their negations.

c) Because for all bitvectors $B$, $0 \land B = 0$ ja $0 + B = B$, we get

\[
Y = (((0 \land P) + P) \lor P) \lor 0 = (P \lor P) \lor 0 = 0.
\]

d) Consider the calculation in case b). A key point there is that the carry bit in the summation travels from position $\ell$ to $i$ and produces $\overline{b}$ 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

\[
((E \land P) + P)[i] = b \text{ and } Y[i] = (b \lor b) \lor 0 = 0.
\]
As a final detail, we compute the bottom row values $g_{mj}$ using the equalities

$$g_{m0} = m \text{ ja } g_{mj} = g_{m,j-1} + \Delta h_{mj}.$$ 

**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 $P$

1. for $c \in \Sigma$ do $B[c] \leftarrow 0^m$  
2. for $i \leftarrow 1$ to $m$ do $B[P[i]][i] = 1$  
3. $Pv \leftarrow 1^m$; $Mv \leftarrow 0$; $g \leftarrow m$  
4. for $j \leftarrow 1$ to $n$ do  
   5. $Eq \leftarrow B[T[j]]$  
   6. $Xh \leftarrow (((Eq \land Pv) + Pv) \lor Pv) \lor Eq$  
   7. $Ph \leftarrow Mv \lor \neg(Xh \lor Pv)$  
   8. $Mh \leftarrow Pv \land Xh$  
   9. $Xv \leftarrow Eq \lor Mv$  
   10. $Pv \leftarrow (Mh << 1) \lor \neg(Xv \lor (Ph << 1))$  
   11. $Mv \leftarrow (Ph << 1) \land Xv$  
   12. $g \leftarrow g + Ph[m] - Mh[m]$  
   13. if $g \leq k$ then output $j$
On an integer alphabet, when \( m \leq w \):

- Pattern preprocessing time is \( O(m + \sigma) \).
- Search time is \( O(n) \).

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

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

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

**Example 3.22:**
\( P = \text{pattern}, \; k = 3 \)
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 $O(n)$ 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 $O(\log_\sigma n)$ characters at a time and $O(\log_\sigma^2 n)$ matrix cells at a time using lookup tables of size $O(n)$. This gives time complexity $O(mn/\log_\sigma^2 n)$.

- 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.

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 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 $ed(P, S) \leq k$ 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 $O(n)$ 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 $O(mn)$ 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 \( n/\sigma^r \).

- The expected total verification time is at most

\[
\mathcal{O}\left( \frac{m^2(k + 1)n}{\sigma^r} \right) \leq \mathcal{O}\left( \frac{m^3n}{\sigma^r} \right).
\]

This is \( \mathcal{O}(n) \) if \( r \geq 3 \log_\sigma m \).

- The condition \( r \geq 3 \log_\sigma m \) is satisfied when \((k + 1) \leq m/(3 \log_\sigma m + 1)\).

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