3.1 Bit vector based approximate string matching

In the following we will focus on pairwise string alignments minimizing the *edit distance*. The algorithms covered are:

1. The classical dynamic programming algorithm, discovered and rediscovered many times since the 1960's. It computes a DP table indexed by the positions of the text and the pattern.
2. This can be improved using an idea due to Ukkonen, if we are interested in hits of a pattern in a text with *bounded* edit distance (which is usually the case).
3. The latter can be further speeded up using *bit parallelism* by an algorithm due to Myers. The key idea is to represent the *differences* between the entries in the DP matrix instead of their absolute values.

3.2 The classical algorithm

We want to find all occurences of a query $P = p_1 p_2 \ldots p_m$ that occur with $k \geq 0$ differences (substitutions and indels) in a text $T = t_1 t_2 \ldots t_n$.

The classic approach computes in time $O(mn)$ a $(m+1) \times (n+1)$ dynamic programming matrix $C[0..m, 0..n]$ using the recurrence

$$C[i,j] = \min \left\{ \begin{array}{rcl} C[i-1,j-1] & + & \delta_{ij} \\ C[i-1,j] & + & 1 \\ C[i,j-1] & + & 1 \end{array} \right\},$$

where

$$\delta_{ij} = \begin{cases} 0, & \text{if } p_i = t_j, \\ 1, & \text{otherwise.} \end{cases}$$

Each location j in the last (m-th) row with $C[m,j] \leq k$ is a solution to our query.

The matrix C is initialized:

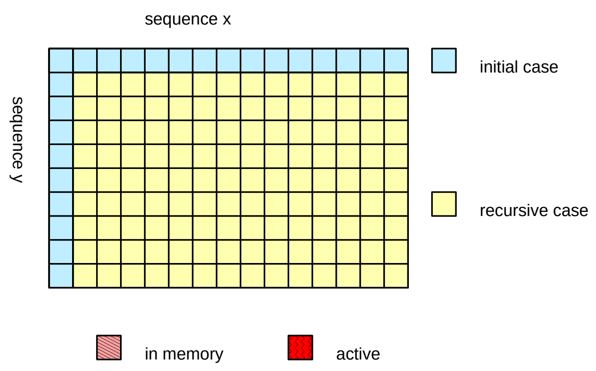
- at the upper boundary by $C[0,j] = 0$, since an occurrence can start anywhere in the text, and
- at the left boundary by $C[i,0] = i$, according to the edit distance.

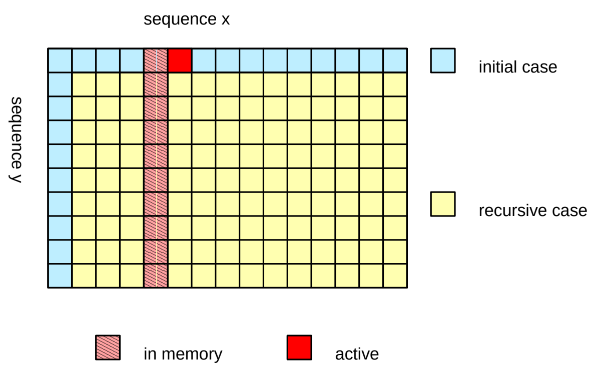
Example. We want to find all occurences with less than 2 differences of the query `annual` in the text `annealing`. The DP matrix looks as follows:

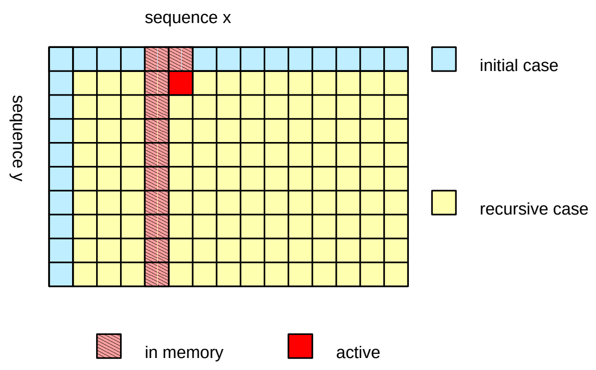
		A	N	N	E	A	L	I	N	G
	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	1	1	1	1
N	2	1	0	1	2	1	1	2	2	2
N	3	2	1	0	1	2	2	2	2	2
U	4	3	2	1	1	2	3	3	3	3
A	5	4	3	2	2	1	2	3	4	4
L	6	5	4	3	3	2	1	2	3	4

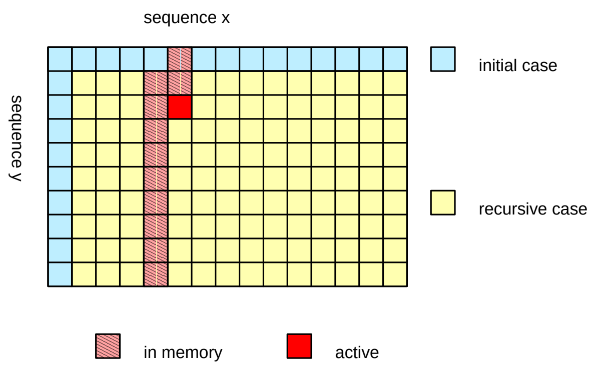
3.3 Computing the score in linear space

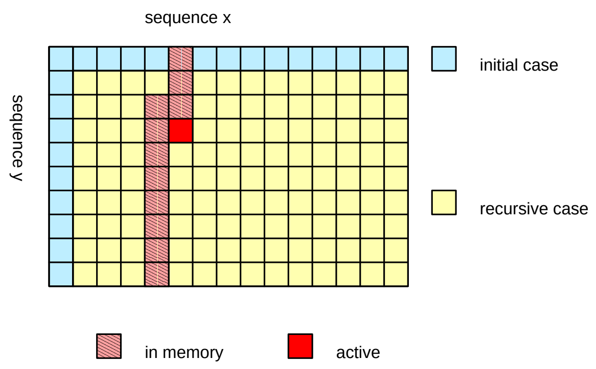
A basic observation is that the *score* computation can be done in $O(m)$ space because computing a column C_j only requires knowledge of the previous column C_{j-1}. Hence we can scan the text from left to right updating our *column vector* and reporting a match every time $C[m,j] \leq k$.

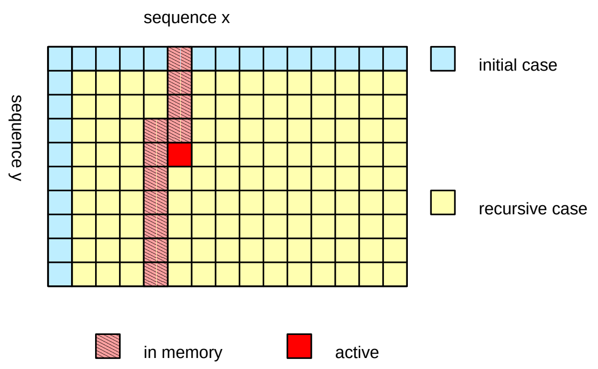


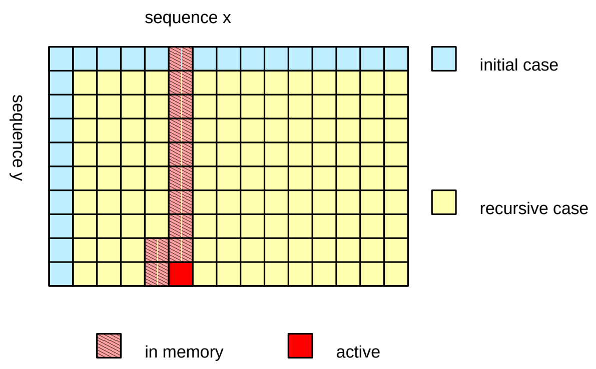












The approximate matches of the pattern can be output on-the-fly when the final value of the column vector is (re-)calculated.

The algorithm still requires in $O(mn)$ time to run, but it uses only $O(m)$ memory.

Note that in order to obtain the actual *alignment* (not just its score), we need to trace back by some other means from the point where the match was found, as the preceeding columns of the DP matrix are no longer available.

3.4 Ukkonen's algorithm

Ukkonen studied the properties of the dynamic programming matrix and came up with a simple twist to the classical algorithm that retains all of its flexibility while reducing the running time to $O(kn)$ (as opposed to $O(mn)$) on average.

The idea is that since the pattern will normally not match the text, the entries of each column read from top to bottom will quickly reach $k + 1$. Let us call an entry of the DP matrix *active* if its value is at most k. Ukkonens algorithm maintains an index *lact* pointing to the ***l**ast **act**ive* cell and updates it accordingly. Due to the properties of *lact* it can avoid working on subsequent cells.

In the exercises you will prove that the value of *lact* can *decrease* in one iteration by more than one, but it can never *increase* by more than one.

3.5 Pseudocode of Ukkonen's algorithm

```
// Preprocessing
for $i \in 0 \ldots m$ do $C_i = i$; od
$lact = k + 1$;
// Searching
for $pos \in 1 \ldots n$ do
    $Cp = 0$; $Cn = 0$;
    for $i \in 1 \ldots lact$ do
        if $p_i = t_{pos}$
          then $Cn = Cp$;
          else
                if $Cp < Cn$ then $Cn = Cp$; fi
                if $C_i < Cn$ then $Cn = C_i$; fi
                $Cn{+}{+}$;
        fi
        $Cp = C_i$; $C_i = Cn$;
    od
    // Updating lact
    while $C_{lact} > k$ do $lact{-}{-}$; od
    if $lact = m$ then report occurrence
                 else $lact{+}{+}$;
    fi
od
```

	$pos - 1$	pos
$i - 1$	Cp	[*was:* Cn]
i	C_i	Cn

3.6 Running time of Ukkonen's algorithm

Ukkonen's algorithm behaves like a standard dynamic programming algorithm, except that it maintains and uses in the main loop the variable *lact*, the last active cell.

The value of *lact* can *decrease* in one iteration by more than one, but it can never *increase* more than one (why?). Thus the total time over the run of the algorithm spent for updating *lact* is $O(n)$. In other words, *lact* is maintained in amortized constant time per column.

One can show that on average the value of *lact* is bounded by $O(k)$. Thus Ukkonen's modification of the classical DP algorithm has an average running time of $O(kn)$.

3.7 Encoding and parallelizing the DP matrix

Next we will look at Myers' bit-parallel algorithm which – in combination with Ukkonens trick – yields a remarkably fast algorithm, which was used e. g. for the overlap computations performed at Celera are the starting point for genome assembly.

For simplicity, we assume that m is smaller than w the length of a machine word.

The main idea of following the bit-vector algorithm is to parallelize the dynamic programming matrix. We will compute the column as a whole in a series of bit-level operations. In order to do so, we need to

1. encode the dynamic programming matrix using bit vectors, and
2. resolve the dependencies (especially *within* the columns).

3.8 Encoding the DP matrix

The binary encoding is done by considering the *differences* between consecutive rows and columns instead of their *absolute* values. We introduce the following nomenclature for these differences ("deltas"):

$$\begin{array}{rrcll} \text{horizontal adjacency property} & \Delta h_{i,j} & = & C_{i,j} - C_{i,j-1} & \in \{-1,0,+1\} \\ \text{vertical adjacency property} & \Delta v_{i,j} & = & C_{i,j} - C_{i-1,j} & \in \{-1,0,+1\} \\ \text{diagonal property} & \Delta d_{i,j} & = & C_{i,j} - C_{i-1,j-1} & \in \{0,+1\} \end{array}$$

Exercise. *Prove that these deltas are indeed within the claimed ranges.*

The delta vectors are encoded as bit-vectors by the following boolean variables:

- $VP_{ij} \equiv (\Delta v_{i,j} = +1)$, the vertical positive delta vector
- $VN_{ij} \equiv (\Delta v_{i,j} = -1)$, the vertical negative delta vector
- $HP_{ij} \equiv (\Delta h_{i,j} = +1)$, the horizontal positive delta vector
- $HN_{ij} \equiv (\Delta h_{i,j} = -1)$, the horizontal negative delta vector
- $D0_{ij} \equiv (\Delta d_{i,j} = 0)$, the diagonal zero delta vector

The deltas and bits are defined such that

$$\begin{aligned} \Delta v_{i,j} &= VP_{i,j} - VN_{i,j} \\ \Delta h_{i,j} &= HP_{i,j} - HN_{i,j} \\ \Delta d_{i,j} &= 1 - D0_{i,j} \,. \end{aligned}$$

It is also clear that these values "encode" the entire DP matrix $C[0..m, 0..n]$ by $C(i,j) = \sum_{r=1}^{i} \Delta v_{r,j}$. Below is our example matrix with $\Delta v_{i,j}$ values.

$\Delta v_{i,j}$:

		A	N	N	E	A	L	I	N	G
	0	0	0	0	0	0	0	0	0	0
A	1	0	1	1	1	0	1	1	1	1
N	1	1	-1	0	1	1	0	1	1	1
N	1	1	1	-1	-1	1	1	0	0	0
U	1	1	1	1	0	0	1	1	1	1
A	1	1	1	1	1	-1	-1	0	1	1
L	1	1	1	1	1	1	-1	-1	-1	0

We denote by $score_j$ the edit distance of a pattern occurrence ending at text position j. The key ideas of Myers' algorithm are as follows:

1. Instead of computing C we compute the Δ values, which in turn are represented as bit-vectors.
2. We compute the matrix column by column as in Ukkonen's version of the DP algorithm.
3. We maintain the value $score_j$ using the fact that $score_0 = m$ and $score_j = score_{j-1} + \Delta h_{m,j}$.

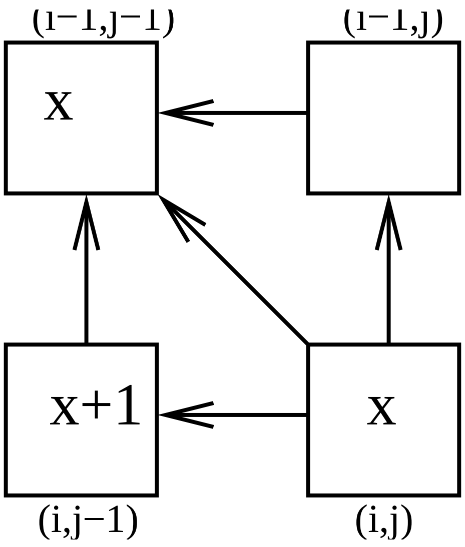
In the following slides we will make some observations about the dependencies of the bit-vectors.

3.9 Observations

Lets have a look at the Δs. The first observation is that

$$HN_{i,j} \Leftrightarrow VP_{i,j-1} \text{ AND } D0_{i,j}\,.$$

Proof. If $HN_{i,j}$ then $\Delta h_{i,j} = -1$ by definition, and hence $\Delta v_{i,j-1} = 1$ and $\Delta d_{i,j} = 0$. This holds true, because otherwise the range of possible values ($\{-1, 0, +1\}$) would be violated.



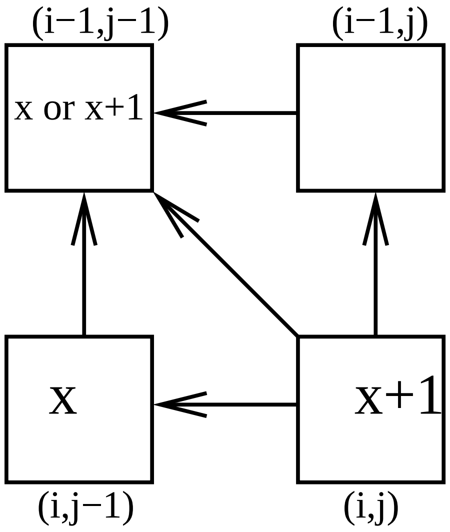
By symmetry we have:

$$VN_{i,j} \Leftrightarrow HP_{i-1,j} \text{ AND } D0_{i,j}\,.$$

The next observation is that

$$HP_{i,j} \Leftrightarrow VN_{i,j-1} \text{ OR NOT } (VP_{i,j-1} \text{ OR } D0_{i,j})\,.$$

Proof. If $HP_{i,j}$ then $VP_{i,j-1}$ cannot hold without violating the ranges. Hence $\Delta v_{i,j-1}$ is -1 or 0. In the first case $VN_{i,j-1}$ is true, whereas in the second case we have neither $VP_{i,j-1}$ nor $D0_{i,j}$



Again by symmetry we have

$$VP_{i,j} \Leftrightarrow HN_{i-1,j} \text{ OR NOT } (HP_{i-1,j} \text{ OR } D0_{i,j})\,.$$

Finally, $D0_{i,j} = 1$ iff $C[i, j]$ and $C[i-1, j-1]$ have the same value. This can be true for three possible reasons, which correspond to the three cases of the DP recurrence:

1. $p_i = t_j$, that is the query at position i matches the text at position j.
2. $C[i, j]$ is obtained by propagating a lower value from the left, that is $C[i, j] = 1 + C[i, j-1]$. Then we have $VN_{i,j-1}$.
3. $C[i, j]$ is obtained by propagating a lower value from above, that is $C[i, j] = 1 + C[i-1, j]$. Then we have $HN_{i,j-1}$.

So writing this together yields:

$$D0_{i,j} \Leftrightarrow (p_i = t_j) \text{ OR } VN_{i,j-1} \text{ OR } HN_{i-1,j}\,.$$

Taking everything together we have the following equivalences:

$$\begin{aligned}
HN_{i,j} &\Leftrightarrow VP_{i,j-1} \text{ AND } D0_{i,j} && (3.1)\\
VN_{i,j} &\Leftrightarrow HP_{i-1,j} \text{ AND } D0_{i,j} && (3.2)\\
HP_{i,j} &\Leftrightarrow VN_{i,j-1} \text{ OR NOT } (VP_{i,j-1} \text{ OR } D0_{i,j}) && (3.3)\\
VP_{i,j} &\Leftrightarrow HN_{i-1,j} \text{ OR NOT } (HP_{i-1,j} \text{ OR } D0_{i,j}) && (3.4)\\
D0_{i,j} &\Leftrightarrow (p_i = t_j) \text{ OR } VN_{i,j-1} \text{ OR } HN_{i-1,j} && (3.5)
\end{aligned}$$

Can these be used to update the five bit-vectors as the algorithms searches through the text?

3.10 Resolving circular dependencies

In principle the overall strategy is clear: We traverse the text from left to right and keep track of five bit-vectors VN_j, VP_j, HN_j, HP_j, $D0_j$, each containing the bits for $1 \le i \le m$. For moderately sized patterns ($m \le w$) these fit into a machine word. The reduction to linear space works similar as for C.

It is clear that we initalize $VP_0 = 1^m$, $VN_0 = 0^m$, so perhaps we can compute the other bit-vectors using the above observations.

There remains a problem to be solved: $D0_{i,j}$ depends on $HN_{i-1,j}$ which in turn depends on $D0_{i-1,j}$ – a value we have not computed yet. But there is a solution.

Let us have a closer look at $D0_{i,j}$ and expand it.

$$\begin{aligned}
D0_{i,j} &= (p_i = t_j) \text{ OR } VN_{i,j-1} \text{ OR } \underline{HN_{i-1,j}}\\
HN_{i-1,j} &= VP_{i-1,j-1} \text{ AND } D0_{i-1,j}\\
\Rightarrow \quad D0_{i,j} &= (p_i = t_j) \text{ OR } VN_{i,j-1} \text{ OR } (VP_{i-1,j-1} \text{ AND } D0_{i-1,j})
\end{aligned}$$

The formula

$$D0_{i,j} = \underline{(p_i = t_j) \text{ OR } VN_{i,j-1}} \text{ OR } (\underline{VP_{i-1,j-1}} \text{ AND } D0_{i-1,j})$$

is of the form

$$D0_i = X_i \text{ OR } (Y_{i-1} \text{ AND } D0_{i-1})\,,$$

where

$$\begin{aligned}
&X_i := (t_j = p_i) \text{ OR } VN_i\\
\text{and}\quad &Y_i := VP_i\,.
\end{aligned}$$

Here we omitted the index j from notation for clarity. We solve for $D0_i$. Unrolling now the first few terms we get:

$$\begin{aligned}
D0_1 &= X_1\,,\\
D0_2 &= X_2 \text{ OR } (X_1 \text{ AND } Y_1)\,,\\
D0_3 &= X_3 \text{ OR } (X_2 \text{ AND } Y_2) \text{ OR } (X_1 \text{ AND } Y_1 \text{ AND } Y_2)\,.
\end{aligned}$$

In general we have:

$$D0_i = \text{ OR}_{r=1}^{\,i} \,(X_r \text{ AND } Y_r \text{ AND } Y_{r+1} \text{ AND } \ldots \text{ AND } Y_{i-1})\,.$$

To put this in words, let $s < i$ be such that $Y_s \dots Y_{i-1} = 1$ and $Y_{s-1} = 0$. Then $D0_i = 1$ if $X_r = 1$ for some $s \leq r \leq i$. That is, the first consecutive block of 1s in Y that is right of i or i itself must be covered by a 1 in X. Here is an example:

$$\begin{aligned} Y &= \texttt{00011111000011} \\ X &= \texttt{00001010000101} \\ D0 &= \texttt{00111110000111} \end{aligned}$$

A position i in $D0$ is set to 1 if one position to the right in Y is a 1 and in the consecutive runs of 1s from there on or at position i is also an X_r set to 1.

3.11 Computing $D0$

Now we solve the problem to compute $D0$ from X and Y using bit-vector and arithmetic operations.

The first step is to compute the & of X and Y and add Y. The result will be that every $X_r = 1$ that is aligned to a $Y_r = 1$ will be propagated one position to the left of a block of 1s if we then add Y to the result.

Again our example:

$$\begin{aligned} Y &= \texttt{00011111000011} \\ X &= \texttt{00001010000101} \\ X \mathbin{\&} Y &= \texttt{00001010000001} \\ (X \mathbin{\&} Y) + Y &= \texttt{00}\underline{\texttt{1}}\texttt{01001000}\underline{\texttt{1}}\texttt{00} \\ &\dots \\ D0 &= \texttt{00111110000111} \end{aligned}$$

Note the two 1s that got propagated to the end of the runs of 1s in Y.

However, note also the one 1 that is not in the solution but was introduced by adding a 1 in Y to a 0 in $(X \mathbin{\&} Y)$.

As a remedy we *XOR* the term with Y so only the bits that changed during the propagation stay turned on. Again our example:

$$\begin{aligned} Y &= \texttt{00011111000011} \\ X &= \texttt{00001010000101} \\ X \mathbin{\&} Y &= \texttt{00001010000001} \\ (X \mathbin{\&} Y) + Y &= \texttt{00101001000100} \\ ((X \mathbin{\&} Y) + Y) \wedge Y &= \texttt{00110110000111} \\ &\dots \\ D0 &= \texttt{00111110000111} \end{aligned}$$

Now we are almost done. The only thing left to fix is the observation that there may be several X_r bits under the same block of Y. Of those all but the first remain unchanged and hence will not be marked by the *XOR*.

To fix this and to account for the case $X_1 = 1$ we *OR* the final result with X. Again our example:

$$\begin{aligned} Y &= \texttt{00011111000011} \\ X &= \texttt{00001010000101} \\ X \mathbin{\&} Y &= \texttt{00001010000001} \\ (X \mathbin{\&} Y) + Y &= \texttt{00101001000100} \\ ((X \mathbin{\&} Y) + Y) \wedge Y &= \texttt{00110110000111} \\ (((X \mathbin{\&} Y) + Y) \wedge Y)) \mid X &= \texttt{00111110000111} \\ = D0 &= \texttt{00111110000111} \end{aligned}$$

Now we can substitute X back by $(p_i = t_j)$ OR VN and Y by VP.

3.12 Preprocessing the alphabet

The only thing left now in the computation of $D0$ is the expression $p_i = t_j$.

Of course we cannot check for all i whether p_i is equal t_j for all i. This would take time $O(m)$ and defeat our goal. However, we can preprocess the query pattern and the alphabet Σ. We use $\mid \Sigma \mid$ many bit-vectors $B[\alpha] \mid \forall \alpha \in \Sigma$ with the property that $B[\alpha]_i = 1$ if $p_i = \alpha$. These vectors can easily be precomputed in time $O(\mid \Sigma \mid m)$.

Now we have everything together.

3.13 Myers' bit-vector algorithm

```
// Preprocessing
for $c \in \Sigma$ do $B[c] = 0^m$ od
for $j \in 1 \ldots m$ do $B[p_j] = B[p_j] \mid 0^{m-j}10^{j-1}$ od
$VP = 1^m;\ VN = 0^m;$
$score = m;$
```

```
// Searching
for $pos \in 1 \ldots n$ do
    $X = B[t_{pos}] \mid VN;$
    $D0 = ((VP + (X \,\&\, VP)) \wedge VP) \mid X;$
    $HN = VP \,\&\, D0;$
    $HP = VN \mid \sim (VP \mid D0);$
    $X = HP \ll 1;$
    $VN = X \,\&\, D0;$
    $VP = (HN \ll 1) \mid \sim (X \mid D0);$
    // Scoring and output
    if $HP \,\&\, 10^{m-1} \neq 0^m$
      then $score$ += 1;
      else if $HN \,\&\, 10^{m-1} \neq 0^m$
            then $score$ −= 1;
          fi
    fi
    if $score \leq k$ report occurrence at $pos$ fi;
od
```

Note that this algorithm easily computes the *edit distance* if we add in line 7 the following code fragment: $\mid 0^{m-1}1$. This because in this case there is a horizontal increment in the 0-th row. In the case of reporting all occurrences each column starts with 0.

3.14 The example

If we try to find `annual` in `annealing` we first run the preprocessing resulting in:

a	0	1	0	0	0	1
l	1	0	0	0	0	0
n	0	0	0	1	1	0
u	0	0	1	0	0	0
*	0	0	0	0	0	0

in addition

$$\begin{array}{rcl} VN & = & 000000 \\ VP & = & 111111 \\ score & & 6 \end{array}$$

We read the first letter of the text, an a.

$$\begin{array}{rcl} Reading & a & 010001 \\ D0 & = & 111111 \\ HN & = & 111111 \\ HP & = & 000000 \\ VN & = & 000000 \\ VP & = & 111110 \\ score & & 5 \end{array}$$

We read the second letter of the text, an n.

$$\begin{array}{rcl} Reading & n & 000110 \\ D0 & = & 111110 \\ HN & = & 111110 \\ HP & = & 000001 \\ VN & = & 000010 \\ VP & = & 111101 \\ score & & 4 \end{array}$$

We read the third letter of the text, an n.

$$\begin{array}{rcl} Reading & n & 000110 \\ D0 & = & 111110 \\ HN & = & 111100 \\ HP & = & 000010 \\ VN & = & 000100 \\ VP & = & 111001 \\ score & & 3 \end{array}$$

We read the fourth letter of the text, an e.

$$\begin{array}{rcl} Reading & e & 000000 \\ D0 & = & 000100 \\ HN & = & 000000 \\ HP & = & 000110 \\ VN & = & 000100 \\ VP & = & 110001 \\ score & & 3 \end{array}$$

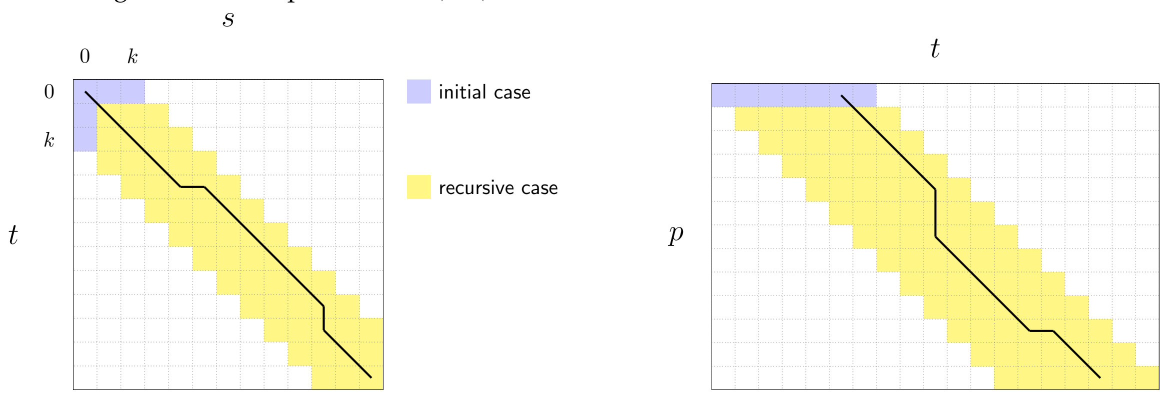
We read the fifth letter of the text an a.

Reading	*e*	010001
*D*0	=	110111
HN	=	110011
HP	=	001100
VN	=	010000
VP	=	100110
score		2

and so on..... In the exercises you will extend the algorithm to handle queries larger than w.

3.15 Banded Myers' bit vector algorithm

- Often only a *small band* of the DP matrix needs to be computed, e. g.:
  - Global alignment with up to k errors (left)



  - Verification of a potential match returned by a filter (right)

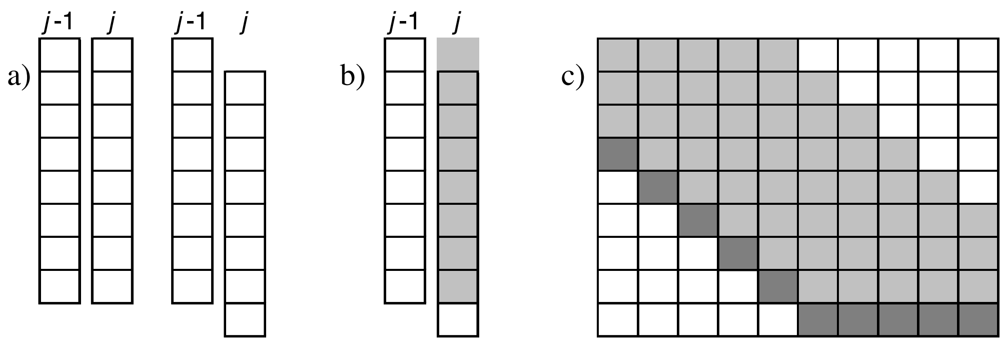
Myers' bit-vector algorithm is efficient if the number of DP rows is less or equal to the machine word length (typically 64). For larger matrices, the bitwise operations must be emulated using multiple words at the expense of running time.

Banded Myers' bit vector algorithm:

- Adaption of Myers' algorithm to calculate a banded alignment (Hyyrö 2003)
- Very efficient alternative if the band width is less than the machine word length

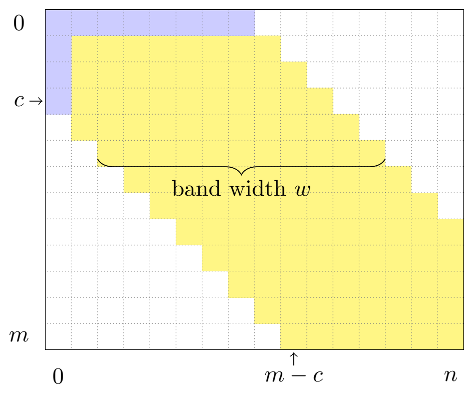
Algorithm outline:

- Calculate *VP* and *VN* for column *j* from column *j* − 1
- Right-shift *VP* and *VN* (see b)
- Use either *D*0 or *HP*, *HN* to track *score* (dark cells in c)



3.16 Preprocessing and Searching

- Given a text t of length n and a pattern p of length m
- We consider a band of the DP matrix:
  - consisting of w consecutive diagonals
  - where the leftmost diagonal is the main diagonal shifted by c to left



Now, we don't encode the whole DP column but the intersection of a column and the band (plus one diagonal left of the band). Thus we store only w vertical deltas in VP and VN. The lowest bits encode the differences between the rightmost and the left adjacent diagonal.

The pattern bit mask computation remains the same.

```
(1) // Preprocessing
(2) for $i \in \Sigma$ do $B[i] = 0^m$ od
(3) for $j \in 1 \ldots m$ do $B[p_j] = B[p_j] \mid 0^{m-j}10^{j-1}$; od
(4) $VP = 1^w$; $VN = 0^w$;
(5) $score = c$;
```

Instead of shifting HP/HN to the left, we shift $D0$ to the right. Pattern bit masks must be shifted accordingly.

```
// Original
for $pos \in 1 \ldots n$ do


    $X = B[t_{pos}] \mid VN$;
    $D0 = ((VP + (X \& VP)) \wedge VP) \mid X$;
    $HN = VP \& D0$;
    $HP = VN \mid \sim (VP \mid D0)$;
    $X = HP \ll 1$;
    $VN = X \& D0$;
    $VP = (HN \ll 1) \mid \sim (X \mid D0)$;
    // Scoring and output
    ...
od
```

```
// Banded
for $pos \in 1 \ldots n$ do
    // Use shifted pattern mask
    $B = (B[t_{pos}]0^w \gg (pos + c)) \& 1^w$;
    $X = B \mid VN$;
    $D0 = ((VP + (X \& VP)) \wedge VP) \mid X$;
    $HN = VP \& D0$;
    $HP = VN \mid \sim (VP \mid D0)$;
    $X = D0 \gg 1$;
    $VN = X \& HP$;
    $VP = HN \mid \sim (X \mid HP)$;
    // Scoring and output
    ...
od
```

The score value is tracked along the left diagonal and along the last row, beginning with $score = c$.
Bit w in $D0$ is used to track the diagonal differences and bit $(w-1) - \big(pos - (m - c + 1)\big)$ in HP/HN is used to track the horizontal differences.

```
(1)  // **Scoring and output**
(2)  if $pos \leq m - c$
(3)    then
(4)        $score \mathrel{+}= 1 - \big(\big(D0 \gg (w-1)\big) \,\&\, 1\big)$;
(5)    else
(6)        $s = (w-2) - \big(pos - (m - c + 1)\big)$;
(7)        $score \mathrel{+}= (HP \gg s) \,\&\, 1$;
(8)        $score \mathrel{-}= (HN \gg s) \,\&\, 1$;
(9)  fi
(10) if $pos \geq m - c \;\wedge\; score \leq k$ report occurrence at $pos$ fi;
```

3.17 Edit distance

In the beginning bit vectors partially encode cells outside the DP matrix. Depending on the search mode (approximate search or edit distance computation) we initialize VP/VN according to the scheme below and zero the pattern masks.

Approximate search

-5	-5	-5	-5	-5	-5
-4	-4	-4	-4	-4	-4
-3	-3	-3	-3	-3	-3
-2	-2	-2	-2	-2	-2
-1	-1	-1	-1	-1	-1
0	0	0	0	0	0
1					
2					
3					
4					
5					
6					

$VP = 1^w, VN = 0^w$

Edit distance computation

0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
0	1	2	3	4	5
1					
2					
3					
4					
5					
6					

$VP = 1^{c+1}0^{w-c-1}, VN = 0^w$

4 Fast filtering algorithms

This exposition is based on

1. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, chapter 6.5, pages 162ff.

We present the hierarchical filtering approach called *PEX* of Navarro and Baeza-Yates.

4.1 Filtering algorithms

The idea behind filtering algorithms is that it might be easier to check that a text position does *not* match a pattern string than to verify that it does.

Filtering algorithms *filter out* portions of the text that cannot possibly contain a match, leaving positions that could match.

The potential match positions then need to be *verified* with another algorithm like for example the bit-parallel algorithm of Myers (BPM).

Filtering algorithms are very sensitive to the *error level* $\alpha := k/m$ since this normally affects the amount of text that can be discarded from further consideration. (m = pattern length, k = errors.)

If most of the text has to be verified, the additional filtering steps are an overhead compared to the strategy of just verifying the pattern in the first place.

On the other hand, if large portions of the text can be discarded quickly, then the filtering results in a faster search.

Filtering algorithms can improve the average-case performance (sometimes dramatically), but not the worst-case performance.

Assume that we want to find all occurrences of a pattern $P = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$ that have an edit distance of at most k.

If we *divide* the pattern into $k + 1$ pieces $P = p^1, \ldots, p^{k+1}$, then at least *one* of the pattern pieces match *without error*.

There is a more general version of this principle first formalized by Myers in 1994:

Lemma 1. *Let Occ match P with k errors,* $P = p^1, \ldots, p^j$ *be a concatenation of subpatterns, and* $a_1, \ldots, a_j$ *be nonnegative integers such that* $A = \sum_{i=1}^{j} a_i$*. Then, for some* $i \in 1, \ldots, j$*, Occ includes a substring that matches* p^i *with at most* $\lfloor a_i k/A \rfloor$ *errors.*

Proof: *Exercise.*

So the basic procedure is:

1. *Divide:* Divide the pattern into $k + 1$ pieces of approximately the same length.
2. *Search:* Search all the pieces simultaneously with a multi-pattern string matching algorithm. According to the above lemma, each possible occurrence will match at least one of the pattern pieces.
3. *Verify:* For each found pattern piece, check the neighborhood with a verification algorithm that is able to detect an occurrence of the whole pattern with edit distance at most k. Since we allow indels, if $p_{i_1} \ldots p_{i_2}$ matches the text $t_j \ldots t_{j+i_2-i_1}$, then the verification has to consider the text area $t_{j-(i_1-1)-k} \ldots t_{j+(m-i_1)+k}$, which is of length $m + 2k$.

4.2 An example

Say we want to find the pattern *annual* in the texts

$t_1 = any\_annealing$ and

$t_2 = an\_unusual\_example\_with\_numerous\_verifications$

with at most 2 errors.

1. *Divide:* We divide the pattern *annual* into $p^1 = an$, $p^2 = nu$, and $p^3 = al$. One of these subpattern has to match with 0 errors.

2. *Search:* We search for all subpatterns:

```
1: searching for an:  in t_1:  find positions 1, 5
                      in t_2:  find position  1
2: searching for nu:  in t_1:  find no positions
                      in t_2:  find positions 5, 25
3: searching for al:  in t_1:  find position 9
                      in t_2:  find position 9
```

3. *Verification:* We have to verify 3 positions in t_1, and 4 positions in t_2, to find 3 occurrences in t_1 and *none* in t_2.

4.3 Hierarchical verification

The toy example makes clear that *many* verifications can be triggered that are unsuccesssful and that many subpatterns can trigger the *same* verification. Repeated verfications can be avoided by carefully sorting the occurrences of the pattern.

It was shown by Baeza-Yates and Navarro that the running time is dominated by the multipattern search for error levels $\alpha = k/m$ below $1/(3\log_{|\Sigma|} m)$. In this region, the search cost is about $O(kn\frac{\log_{|\Sigma|} m}{m})$. For higher error levels, the cost for verifications starts to dominate, and the filter efficiency deteriorates abruptly.

Baeza-Yates and Navarro introduced the idea of hierarchical verification to reduce the verification costs, which we will explain next. Then we will work out more details of the three steps.

Navarro and Baeza-Yates use Lemma **??** for a *hierarchical verification*. The idea is that, since the verification cost is high, we pay too much for verifying the *whole* pattern *each* time a small piece matches. We could possibly reject the occurrence with a cheaper test for a shorter pattern.

So, instead of directly dividing the pattern into $k+1$ pieces, we do it hierarchically. We split the pattern first in two pieces and search for each piece with $\lfloor k/2 \rfloor$ errors, following Lemma **??**. The halves are then recursively split and searched until the error rate reaches zero, i. e. we can search for exact matches.

With hierarchical verification the area of applicability of the filtering algorithm grows to $\alpha < 1/\log_{|\Sigma|} m$, an error level three times as high as for the naive paritioning and verification. In practice, the filtering algorithm pays off for $\alpha < 1/3$ for medium long patterns.

Example. Say we want to find the pattern P = aaabbbcccddd in the text T = xxxbbbxxxxxx with at most $k = 3$ differences. The pattern is split into four pieces p^1 = aaa, p^2 = bbb, p^3 = ccc, p^4 = ddd. We search with $k = 0$ errors in level 2 and find *bbb*.

```
level 0            aaabbbcccddd           with k=3 errors
                   /          \
level 1       aaabbb          cccddd      with k=1 errors
              /    \          /    \
level 2     aaa    bbb      ccc    ddd    with k=0 errors
```

Now instead of verifying the complete pattern in the complete text (at level 0) with $k = 3$ errors, we only have to check a slightly bigger pattern (aaabbb) at level 1 with one error. This is much cheaper. In this example we can decide that the occurrence bbb cannot be extended to a match.

```
level 0            aaabbbcccddd           with k=3 errors
                   /          \
level 1       AAABBB          cccddd      with k=1 errors
              /    \          /    \
level 2     aaa    BBB      ccc    ddd    with k=0 errors
```

4.4 The PEX algorithm

Divide: Split pattern into $k + 1$ pieces, such that each piece has equal probability of occurring in the text. If no other information is available, the uniform distribution is assumed and hence the pattern is divided in pieces of equal length.

Build Tree: Build a tree of the pattern for the hierarchical verification. If $k + 1$ is not a power of 2, we try to keep the binary tree as balanced as possible.

Each node has two members $from$ and to indicating the first and the last position of the pattern piece represented by it. The member *err* holds the number of allowed errors. A pointer *myParent* leads to its parent in the tree. (There are no child pointers, since we traverse the tree only from the leafs to the root.) An internal variable $left$ holds the number of pattern pieces in the left subtree. *idx* is the next leaf index to assign. *plen* is the length of a pattern piece.

Algorithm CreateTree generates a hierarchical verification tree for a single pattern. (Lines **??** and **??** are justified by Lemma **??**.)

```
(1)  CreateTree( $p = p_i p_{i+1} \dots p_j$, $k$, $myParent$, $idx$, $plen$ )
(2)  // Note: the initial call is: CreateTree ( $p$, $k$, $nil$, 0, $\lfloor m/(k+1) \rfloor$ )
(3)  Create new node $node$
(4)  $from(node) = i$
(5)  $to(node) = j$
(6)  $left = \lceil (k+1)/2 \rceil$
(7)  $parent(node) = myParent$
(8)  $err(node) = k$
(9)  if $k = 0$
(10)   then $leaf_{idx} = node$
(11)   else
(12)       $lk = \lfloor (left \cdot k)/(k+1) \rfloor$
(13)       CreateTree( $p_i \dots p_{i+left \cdot plen - 1}$, $lk$, $node$, $idx$, $plen$ )
(14)       $rk = \lfloor ((k + 1 - left) \cdot k)/(k+1) \rfloor$
(15)       CreateTree( $p_{i+left \cdot plen} \dots p_j$, $rk$, $node$, $idx + left$, $plen$ )
(16) fi
```

Example: Find the pattern $P =$ `annual` in the text $T =$ `annual_CPM_anniversary` with at most $k = 2$ errors. First we build the tree with $k + 1 = 3$ leaves. Below we write at each node n_i the variables $(from, to, error)$.

```
                   "annual" n4=(1,6,2)
                   /                  \
         "annu" n3=(1,4,1)             \
          /         \                   \
"an" n0=(1,2,0)   "nu" n1=(3,4,0)   "al" n2=(5,6,0)
    |                 |                 |
leaf 0            leaf 1            leaf 2
```

Search: After constructing the tree, we have $k + 1$ leafs $leaf_i$. The $k + 1$ subpatterns

$$\{ \, p_{from(n)}, \dots, p_{to(n)}, \; n = leaf_i, \; i \in \{0, \dots, k\} \, \}$$

are sent as input to a multi-pattern search algorithm (e. g. Aho-Corasick, Wu-Manbers, or SBOM). This algorithm gives as output a list of pairs (pos, i) where *pos* is the text position that matched and i is the number of the piece that matched.

The PEX algorithm performs verifications on its way upward in the tree, checking the presence of longer and longer pieces of the pattern, as specified by the nodes.

```
(1) Search phase of algorithm PEX
(2) for $(pos, i) \in$ output of multi-pattern search do
(3)     $n = leaf_i$; $in = from(n)$; $n = parent(n)$;
(4)     $cand = true$;
```

```
(5)   while $cand = true$ and $n \neq nil$ do
(6)         $p_1 = pos - (in - from(n)) - err(n)$;
(7)         $p_2 = pos + (to(n) - in) + err(n)$;
(8)         verify text $t_{p_1} \dots t_{p_2}$ for pattern piece $p_{from(n)} \dots p_{to(n)}$
(9)            allowing $err(n)$ errors;
(10)        if pattern piece was not found
(11)          then $cand = false$;
(12)          else $n = parent(n)$;
(13)        fi
(14)  od
(15)  if $cand = true$
(16)    then report the positions where the whole $p$ was found;
(17)  fi
(18) od
```

We search for `annual` in `annual_CPM_anniversary`. We constructed the tree for `annual`. A multi-pattern search algorithm finds: $(1,1)$, $(12,1)$, $(3,2)$, $(5,3)$. (Note that leaf i corresponds to pattern p^{i+1}). For each of these positions we do the hierarchical verification:

```
Initialization for (1,1);
n=n0; in=1; n=n3; cand=true;
While loop;
  a) p1=1-(1-1)-1=0;  p2=1+(4-1)+1=5;
  verify pattern annu in text annua with 1 error => found !
  b) p1=1-(1-1)-2=-1; p2=1+(6-1)+2=8;
  verify pattern annual in text annual_C => found !
  c) report end positions (6,7,8)

Initialization for (3,2);
n=n1; in=3; n=n3; cand=true;
While loop;
  a) p1=3-(3-1)-1=0;  p2=3+(4-3)+1=5;
  verify pattern annu in text annua with 1 error => found !
  b) p1=3-(3-1)-2=-1; p2=3+(6-3)+2=8;
  verify pattern annual in text annual_C => found !
  c) report end positions (6,7,8)

Initialization for (12,1);
n=n0; in=1; n=n3; cand=true;
While loop;
  a)  p1=12-(1-1)-1=11; p2=12+(4-1)+1=16;
  verify pattern annu in text _anniv with 1 error => found !
  b)  p1=12-(1-1)-2=10; p2=12+(6-1)+2=19;
  verify pattern annual in text M_annivers => NOT found !
```

Note that sorting of the leaf matches would avoid verifying the match three times.

4.5 Summary

- Filtering algorithms prevent a large portion of the text from being looked at.
- The larger $\alpha = k/m$, the less efficient filtering algorithms become.
- Filtering algorithms based on the pigeonhole principle need an exact, multi-pattern search algorithm and a verification capable approximate string matching algorithm.
- The PEX algorithm starts verification from short exact matches and considers longer and longer substrings of the pattern as the verification proceeds upward in the tree.