3 Bit-parallel string matching

We will discuss

- Shift-Or algorithm
- Ukkonens algorithm for k-differences matching
- Myers' bit vector algorithm

This exposition has been developed by C. Gröpl, G. Klau, D. Weese, and K. Reinert. It is based on the following sources, which are all recommended reading:

1. Navarro and Raffinot (2002) *Flexible Pattern Matching in Strings*, sections 2.2.2 and 6.4.
2. G. Myers (1999) *A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming*, Journal of the ACM, 46(3): 495-415
3. Heikki Hyyrö (2001) *Explaining and Extending the Bit-parallel Approximate String Matching Algorithm of Myers*, Technical report, University of Tampere, Finland.
4. Heikki Hyyrö (2003) *A bit-vector algorithm for computing Levenshtein and Damerau edit distances*, Nordic Journal of Computing, 10: 29-39

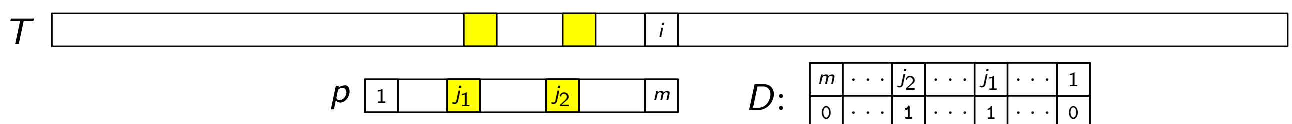
3.1 Shift-And and Shift-Or

The *Shift-And* and the *Shift-Or* algorithms are closely related. We start with the Shift-And algorithm, which is a bit easier to explain. The Shift-Or algorithm is just a clever implementation trick.

The algorithms maintain the set of all prefixes of p that match a suffix of the text read. They use *bit-parallelism* to update this set for each new text character. The set is represented by a *bit mask* $D = d_m, \ldots, d_1$. Note that the indices run left-to-right in the pattern, but bits are usually numbered LSB-to-MSB, i. e. right-to-left.

We keep the following *invariant*:

There is a 1 *at the j-th position of D if and only if* $p_1, \ldots, p_j$ *is a suffix of* $t_1, \ldots, t_i$.



If the size of p is less than the word length, this array will fit into a computer register. (Modern PCs have 64 bit architectures.)

When reading the next text character t_{i+1}, we have to compute the new set D'. In the Shift-And algorithm, we say that a position of the bit mask is *active* if it is 1.

We use the following fact:

Observation. Position $j+1$ in the set D' will be active if and only if

1. the position j was active in D, that is, $p_1, \ldots, p_j$ was a suffix of $t_1, \ldots, t_i$, and
2. t_{i+1} matches p_{j+1}.

It remains to find a way how to perform this update using bit-level operations.

3.2 Preprocessing

In a preprocessing step, the algorithm builds a table B which stores a bit mask $b_m, \ldots, b_1$ for each character $\alpha \in \Sigma$. The mask $B[\alpha]$ has the j-th bit set if $p_j = \alpha$.

Example. $\Sigma = \{a, c, g, t\}$,

p	=	tcaa	p^{rev}	=	aact
$B[a]^{rev}$	=	0011	$B[a]$	=	1100
$B[c]^{rev}$	=	0100	$B[c]$	=	0010
$B[g]^{rev}$	=	0000	$B[g]$	=	0000
$B[t]^{rev}$	=	1000	$B[t]$	=	0001

The computation of B takes $O(m + |\Sigma|)$ time using the following algorithm, if we can we assume that operations on $B[\alpha]$ can be done in constant time.

Preprocessing:
Input: pattern p of length m
Output: vector of bit masks B

for $\alpha \in \Sigma$ **do** $B[\alpha] = 0^m$;
for $j \in 1 \ldots m$ **do** $B[p_j] = B[p_j] \mid 0^{m-j}10^{j-1}$;

3.3 Searching

Initially we set $D = 0^m$ and for each new character t_{pos} we update D using the formula

$$D' = \left((D \ll 1) \mid 0^{m-1}1\right) \,\&\, B[t_{pos}] \ .$$

This update maintains our invariant using the above observation.

The shift operation marks all positions as potential prefix-suffix matches that were such matches in the previous step (notice that this includes the empty string ε). In addition, to stay a match, the character t_{pos} has to match p at those positions. This is achieved by applying an &-operation with the corresponding bitmask $B[t_{pos}]$.

3.4 Shift-And Pseudocode

1 **Input:** text T of length n and pattern p of length m
2 **Output:** all occurences of p in T
3 *Preprocessing:*
4 **for** $c \in \Sigma$ **do** $B[c] = 0^m$;
5 **for** $j \in 1 \ldots m$ **do** $B[p_j] = B[p_j] \mid 0^{m-j}10^{j-1}$;
6 *Searching:*
7 $D = 0^m$;
8 **for** $pos \in 1 \ldots n$ **do**
9 $\quad D = \left((D \ll 1) \mid 0^{m-1}1\right) \,\&\, B[t_{pos}]$;
10 $\quad$ **if** $D \,\&\, 10^{m-1} \neq 0^m$ **then output** "p occurs at position $pos - m + 1$";

The running time for the searching phase is $O(n)$, assuming that operations on D can be done in constant time.

Thus the overall running time is $O(m + n + |\Sigma|)$. If m and the alphabet size is constant, the running time is $O(n)$.

3.5 Shift-And versus Shift-Or

So what is the Shift-Or algorithm about? It is just an implementation trick to avoid a bit operation, namely the "$\mid 0^{m-1}1$" in line 10.

In the Shift-Or algorithm we complement all bit masks of B and use a complemented bit mask D. Now the $\ll$ operator will introduce a 0 to the right of D' and the new suffix stemming from the empty string is already in D'. Obviously we have to use a bit | instead of an & and report a match whenever $d_m = 0$. Thus the recursion becomes:

$$D' = (D \ll 1) \mid B[t_{pos}] \ .$$

Now let's look at an example of the Shift-Or algorithm.

Note that the bit order is reversed in the Java applet on `http://www-igm.univ-mlv.fr/~lecroq/string`. This is actually quite intuitive, to we will also write D from LSB-to-MSB. Just keep in mind that $\ll$ will become a *right* shift using this notation!

3.6 Shift-And Example

```
Pattern: tcaa
B[a] =   0011
B[c] =   0100
B[g] =   0000
B[t] =   1000
B[*] =   0000

Shift-Or algorithm:

D = 0000
(written LSB-to-MSB)

Text: atcatcaatc


atcatcaatc
^
0000.a -> 0000
```

```
atcatcaatc
 ^
0000.t -> 1000

aTcatcaatc
  ^
1000.c -> 0100

aTCatcaatc
   ^
0100.a -> 0010

aTCAtcaatc
    ^
0010.t -> 1000

atcaTcaatc
     ^
1000.c -> 0100
```

```
atcaTCaatc
      ^
0100.a -> 0010

atcaTCAatc
       ^
0010.a -> 0001
          ^ hit
atcaTCAAtc
        ^
0001.t -> 1000

atcaTCAATc
         ^
1000.c -> 0100

atcaTCAAtc

String  length: 10
Pattern length: 4
Character inspections: 10
```

3.7 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.

3.8 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

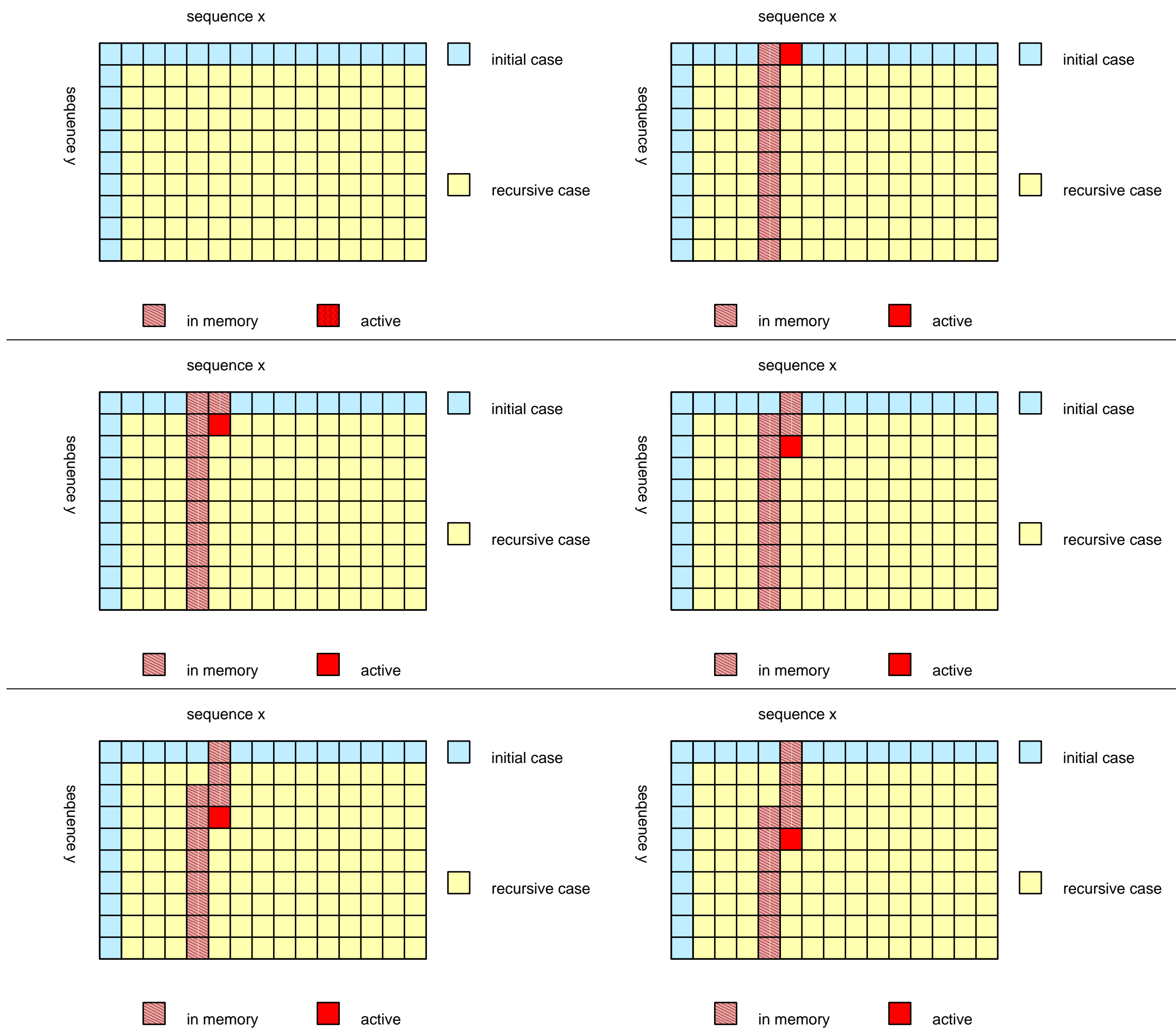
- at the left boundary by $C[i, 0] = i$, according to the edit distance.

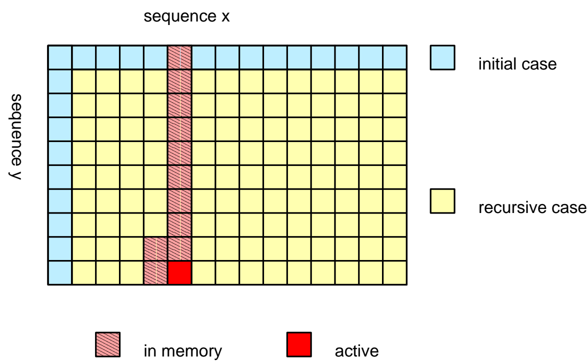
Example. We want to find all occurences with less than 2 differences of the query `annual` in the text `annealing`. The DP matrix looks as follows:

		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

3.9 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.

3.10 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.

3.11 Pseudocode of Ukkonen's algorithm

```
 1 // Preprocessing
 2 for i ∈ 0...m do C_i = i; od
 3 lact = k + 1;
 4 // Searching
 5 for pos ∈ 1...n do
 6     Cp = 0; Cn = 0;
 7     for i ∈ 1...lact do
 8         if p_i = t_pos
 9           then Cn = Cp;
10           else
11               if Cp < Cn then Cn = Cp; fi
12               if C_i < Cn then Cn = C_i; fi
13               Cn++;
14         fi
15         Cp = C_i; C_i = Cn;
16     od
17     // Updating lact
18     while C_lact > k do lact−−; od
19     if lact = m then report occurrence
20                 else lact++;
21     fi
22 od
```

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

3.12 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 $O(kn)$.

3.13 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. It will be an exercise to extend the algorithm to handle $m \geq w$.

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).

3.14 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"):

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

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 $score_j = score_{j-1} + \Delta h_{m,j}$.

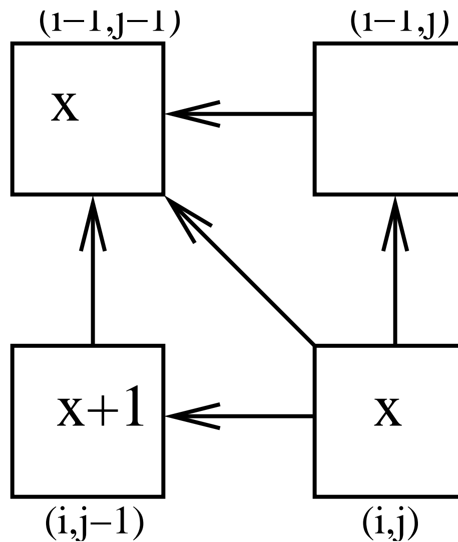
In the following slides we will make some observations about the dependencies of the bit-vectors.

3.15 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.



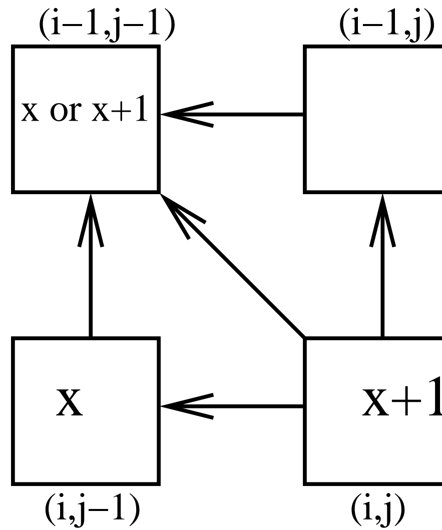
By symmetry we have:

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

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:

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

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

3.16 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.

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

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

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

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

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

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 $Y_s \ldots Y_{i-1} = 1$ 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 must be covered by a 1 in X. Here is an example:

$$Y = \texttt{00011111000011}$$
$$X = \texttt{00001010000101}$$
$$D0 = \texttt{00111110000111}$$

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 is also an X_r set to 1.

3.17 Computing $D0$

Now we solve the problem to compute $D0$ from X and Y using bit-vector 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 by the addition one position to the left of a block of 1s.

Again our example:

$$Y = \texttt{00011111000011}$$
$$X = \texttt{00001010000101}$$
$$X \,\&\, Y = \texttt{00001010000001}$$
$$(X \,\&\, Y) + Y = \texttt{00}\underline{\texttt{1}}\texttt{01001000}\underline{\texttt{1}}\texttt{00}$$
$$\ldots$$
$$D0 = \texttt{00111110000111}$$

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 $(X \,\&\, Y)$.

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

$$Y = \texttt{00011111000011}$$
$$X = \texttt{00001010000101}$$
$$X \,\&\, Y = \texttt{00001010000001}$$
$$(X \,\&\, Y) + Y = \texttt{00101001000100}$$
$$((X \,\&\, Y) + Y) \wedge Y = \texttt{00110110000111}$$
$$\ldots$$
$$D0 = \texttt{00111110000111}$$

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:

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

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

3.18 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.

3.19 Myers' bit-vector algorithm

```
1  // Preprocessing
2  for c ∈ Σ do B[c] = 0^m od
3  for j ∈ 1...m do B[p_j] = B[p_j] | 0^(m−j)10^(j−1) od
4  VP = 1^m; VN = 0^m;
5  score = m;
```

```
1  // Searching
2  for pos ∈ 1...n do
3      X = B[t_pos] | VN;
4      D0 = ((VP + (X & VP)) ∧ VP) | X;
5      HN = VP & D0;
6      HP = VN | ~ (VP | D0);
7      X = HP ≪ 1;
8      VN = X & D0;
9      VP = (HN ≪ 1) | ~ (X | D0);
10     // Scoring and output
11     if HP & 10^(m−1) ≠ 0^m
12        then score += 1;
13         else if HN & 10^(m−1) ≠ 0^m
14                then score −= 1;
15             fi
16     fi
17     if score ≤ k report occurrence at pos fi;
18 od
```

Note that this algorithm easily computes the *edit distance* if we add in line **??** 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.

3.20 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

$$\begin{aligned} VN &= 000000 \\ VP &= 111111 \\ score & \quad 6 \end{aligned}$$

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

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

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

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

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

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

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

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

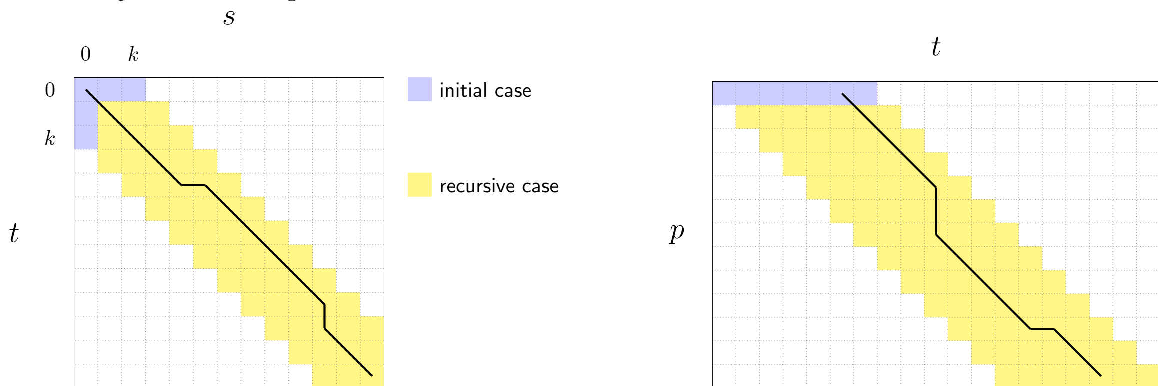
We read the fifth letter of the text an a.

$$
\begin{array}{rcl}
\textit{Reading} & e & 010001 \\
D0 & = & 110111 \\
HN & = & 110011 \\
HP & = & 001100 \\
VN & = & 010000 \\
VP & = & 100110 \\
\textit{score} & & 2
\end{array}
$$

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

3.21 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.

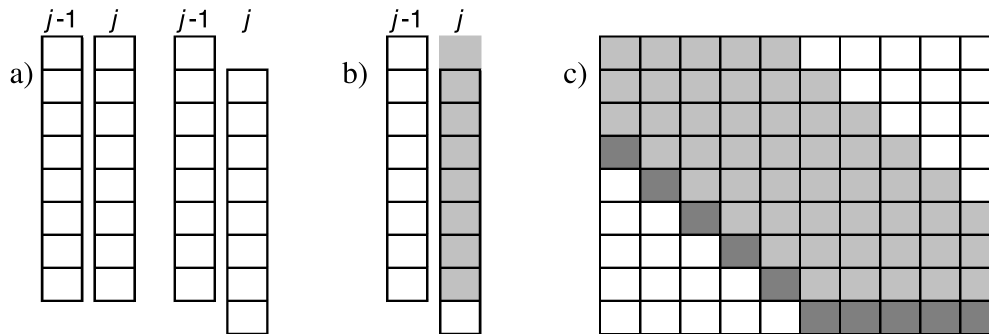
Banded Myers' bit vector algorithm:

- 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:

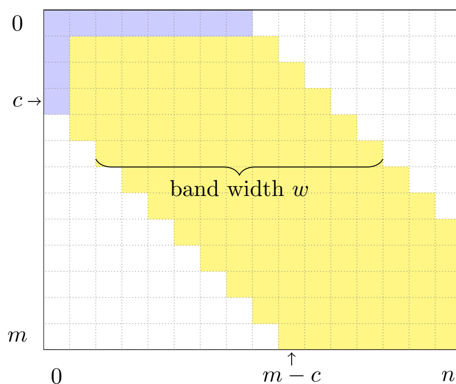
- Calculate VP and VN for column j from column $j-1$

- Right-shift VP and VN (see b)
- Use either $D0$ or HP, HN to track *score* (dark cells in c)



3.22 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 ∈ Σ do B[i] = 0^m od
3 for j ∈ 1...m do B[p_j] = B[p_j] | 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 ∈ 1...n do


    X = B[t_pos] | VN;
    D0 = ((VP + (X & VP)) ∧ VP) | X;
    HN = VP & D0;
    HP = VN | ~ (VP | D0);
    X = HP ≪ 1;
    VN = X & D0;
    VP = (HN ≪ 1) | ~ (X | D0);
    // Scoring and output
    ...
od
```

```
// Banded
for pos ∈ 1...n do
    // Use shifted pattern mask
    B = (B[t_pos]0^w ≫ (pos + c)) & 1^w;
    X = B | VN;
    D0 = ((VP + (X & VP)) ∧ VP) | X;
    HN = VP & D0;
    HP = VN | ~ (VP | D0);
    X = D0 ≫ 1;
    VN = X & HP;
    VP = HN | ~ (X | 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 ≤ m − c
3     then
4           score += 1 − ((D0 ≫ (w − 1)) & 1);
5     else
6           s = (w − 2) − (pos − (m − c + 1));
7           score += (HP ≫ s) & 1;
8           score −= (HN ≫ s) & 1;
9  fi
10 if pos ≥ m − c ∧ score ≤ k report occurrence at pos fi;
```

3.23 Additional Improvements

- *score* can decrease along the horizontal, but *not* along the diagonal
  - Break if $score > k + n - (m - c)$
- Instead of precomputing large pattern bit masks, use bit masks of size w and update them online
  - For small alphabets, e. g. DNA:

    ```
    // shift all masks and set bit in B[p_{pos+c}]
    for i ∈ Σ do B[i] = B[i] ≫ 1; od
    if pos + c ≤ m then B[p_{pos+c}] = B[p_{pos+c}] | 10^{w−1}; fi
    B = B[t_pos];
    ```

  - For large alphabets:

    ```
    // store conducted shift length for every mask in S
    // only shift used and updated masks and set bit in B[p_{pos+c}]
    if pos + c ≤ m
       then
            B[p_{pos+c}] = (B[p_{pos+c}] ≫ (pos − S[p_{pos+c}])) | 10^{w−1};
            S[p_{pos+c}] = pos;
    fi
    B = B[t_pos] ≫ (pos − S[t_pos]);
    ```

3.24 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					

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