5 BLAST and Aho-Corasick

This exposition was prepared by Clemens Gröpl, based on versions by Daniel Huson and Knut Reinert. It is based on the following sources, which are all recommended reading:

- Dan Gusfield: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997, pages 379ff. ISBN 0-521-58519-8
- Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, chapter 3.

5.1 BLAST, FASTA, ...

Pairwise alignment is used to detect homologies between different protein or DNA sequences, either as global or local alignments.

This can be solved using dynamic programming in time proportional to the product of the lengths of the two sequences being compared.

However, this is too slow for searching current databases and in practice algorithms are used that run much faster, at the expense of possibly missing some significant hits due to the heuristics employed.

Such algorithms are usually on *seed and extend* approaches in which first small exact matches are found, which are then extended to obtain long inexact ones.

5.2 Some BLAST terminology

BLAST, the Basic Local Alignment Search Tool, is perhaps the most widely used bioinformatics tool ever written. It is an alignment heuristic that determines "local alignments" between a *query* q and a *database* d.

A *segment* is simply a substring s of q or d.

A *segment-pair* (s, t) consists of two segments, one in q and one d, of the same length.

```
V A L L A R
P A M M A R
```

We think of s and t as being *aligned without gaps* and *score* this alignment using a substitution score matrix, e.g. BLOSUM or PAM.

The alignment score for (s, t) is denoted by $\sigma(s, t)$.

A *locally maximal segment pair (LMSP)* is any segment pair (s, t) whose score cannot be improved by shortening or extending the segment pair.

Given a *cutoff score* C, a segment pair (s, t) is called a *high-scoring segment pair (HSP)*, if it is locally maximal and $\sigma(s, t) \geq C$.

Given a cutoff score, the goal of BLAST is to compute all high-scoring segment pairs.

5.3 BLAST algorithm for protein sequences

A *word* is simply a short substring.

BLAST computes high-scoring segments pairs using short words as *seeds*. Exact matches of words are searched using a special algorithm. Then the short matches are *extended* to both sides, leading to locally maximal segment pairs.

All this is fine-tuned by a couple of parameters: the *word size* w, the *similarity threshold* T used when generating the list of short words to be searched for, and a minimum match score C which applies to HSPs.

The BLAST algorithm for *protein* sequences operates as follows:

1. The list of all words of length w that have similarity $\geq T$ to some word in the query sequence q is generated.
2. The database sequence d is scanned for all *exact matches* t of words s in the list.
3. Each such *seed* (s, t) is *extended* until its score $\sigma(s, t)$ falls a certain distance below the best score found for shorter extensions. All extensions are reported that have score $\geq C$.

In practice, w is around 3 for proteins.

Example.

Using words of length 2:

q	l	n	f	s	a	g	w
q	l						
	l	n					
		n	f				
			f	s			
				s	a		
					a	g	
						g	w

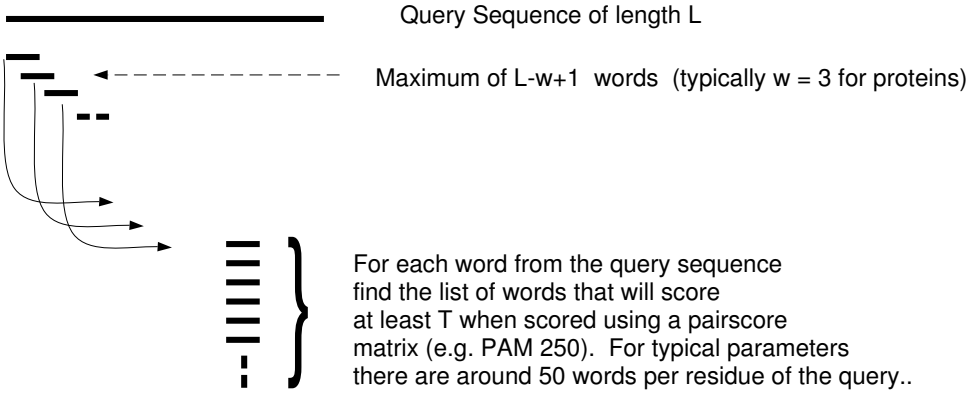
$\Longrightarrow$

Initial Words
ql
ln
nf
fs
sa
ag
gw

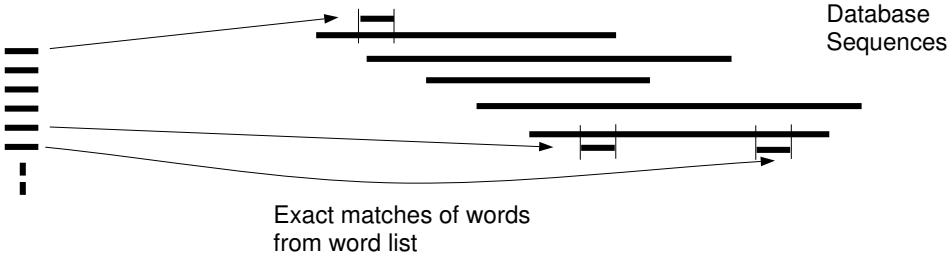
Expanding the initial word list:

Initial Words	Expanded List
ql	ql, qm, hl, zl
ln	ln, lb
nf	nf, af, ny, df, qf, ef, gf, hf, kf, sf, tf, bf, zf
fs	fs, fa, fn, fd, fg, fp, ft, fb, ys
sa	(nothing scores 8 or higher)
ag	ag
gw	gw, aw, rw, nw, dw, qw, ew, hw, iw, kw, mw, pw, sw, tw, vw, bw, zw, xw

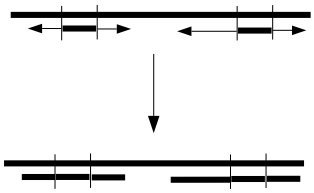
(1) For the query find the list of high scoring words of length w.



(2) Compare the word list to the database and identify exact matches.



(3) For each word match, extend alignment in both directions to find alignments that score greater than score threshold S.



With a careful implementation, the list of all words of length w that have similarity $\geq T$ to some word in the query sequence q can be produced in time proportional to the number of words in the list.

The similar words are immediately placed in a *keyword tree* and then, for each word in the tree, all exact locations of these words in the database d are detected in time linear to the length of d, using a variation of the *Aho-Corasick algorithm* for multiple exact string matching.

As BLAST does not allow indels, also the hit extension is very fast.

Note that the use of seeds of length w and the termination of extensions with fading scores are both steps that speed up the algorithm, but also imply that BLAST is not guaranteed to find all HSPs.

5.4 BLAST algorithm for DNA sequences

For *DNA* sequences, BLAST operates as follows:

- The list of all words of length w in the query sequence a is generated.
- The database d is scanned for all hits of words in this list. Blast uses a two-bit encoding for DNA. This saves space and also search time, as four bases are encoded per byte.

In practice, K is around 12 for DNA.

5.5 The BLAST family

There are a number of different variants of the BLAST program:

- BLASTN: compares a DNA query sequence to a DNA sequence database,
- BLASTP: compares a protein query sequence to a protein sequence database,
- TBLASTN: compares a protein query sequence to a DNA sequence database (6 frames translation),
- BLASTX: compares a DNA query sequence (6 frames translation) to a protein sequence database, and
- TBLASTX: compares a DNA query sequence (6 frames translation) to a DNA sequence database (6 frames translation).

BLAST is constantly being developed further and many internet services is available, e.g.,
`http://www.ncbi.nlm.nih.gov/BLAST/`
`http://blast.wustl.edu/`

5.6 Multiple string matching

The task at hand is to find all occurrences of a given set of r patterns $P = \{p^1, \ldots, p^r\}$ in a text $T = t_1, \ldots, t_n$. Each p^i is a string $p^i = p^i_1, \ldots, p^i_{m_i}$. Usually n is much bigger than m_i.

Idea of the Aho-Corasick algorithm:

The Aho-Corasick algorithm belongs to the class of *prefix-based* approaches. We assume that we have read the text up to position i and that we know the length of the longest suffix of $t_1, \ldots, t_i$ that is also a prefix of some pattern $p^k \in P$.

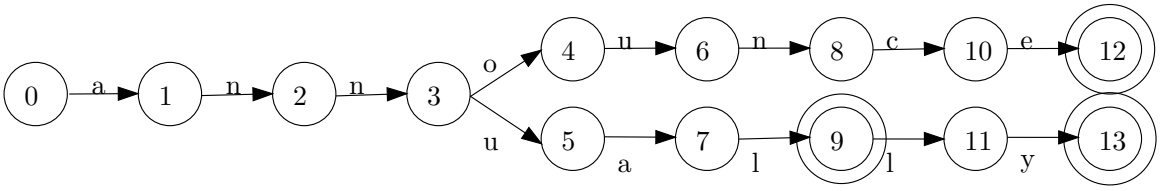
The Aho-Corasick algorithm maintains a data structure, called the *Aho-Corasick automaton*, to keep track of the longest prefix of some pattern that is also a suffix of the text window.

The AC automaton is built on top of another data structure called trie, so we explain this one first.

5.7 The trie data structure

A *trie* is a compact representation of a set of strings; in our case, the set P. It is a rooted, directed tree. The *path label* of a node v is the string $L(v)$ read when traversing the trie from the root to the node v. A node is a *terminal* node if its path label is within the set of strings to be represented by the trie. Each $p^i \in P$ corresponds to a terminal node.

Example: $P = \{\texttt{annual}, \texttt{annually}, \texttt{announce}\}$



In the following pseudocode, $\delta(current, p^i_j)$ denotes the successor state reached from state *current* by the outgoing edge that is labeled with p^i_j. The symbol θ means "undefined".

The set of terminal states is denoted F (for "final"). (In the code each state has a set F of indices indicating that it is terminal for the respective strings.)

```
(1)  Trie(P = p^1, ..., p^r);
(2)  Create an initial non-terminal state root
(3)  for i ∈ 1...r do
(4)      current = root; j = 1;
(5)      while j ≤ m_i ∧ δ(current, p^i_j) ≠ θ do
(6)          current = δ(current, p^i_j); j++;
(7)      od
(8)      while j ≤ m_i do
(9)          create state state;
(10)         δ(current, p^i_j) = state; current = state; j++;
(11)     od
(12)     if current is terminal then
(13)                                  F(current) = F(current) ∪ {i};
(14)                              else
(15)                                  mark current as terminal ;
(16)                                  F(current) = {i};
(17)     fi
(18) od
```

The size of the trie and the running time of its basic operations depend on the implementation of the transition function δ.

- We can provide each node with a table of size $|\Sigma|$. This yields access in $O(1)$, but may still be worse in practice because the high space consumption will lead to many processor (L1) cache misses.
- We can use a sorted array of size $\sum_{p\in P} |p|$ and store only the existing edges. The edge list is accessed using binary search which takes $O(log|\Sigma|)$ time per transition.
- In practice, a hash table will be the best compromise for most applications. It uses $O(\sum_{p\in P} |p|)$ space and usually $O(1)$ time per access (but $O(\sum_{p\in P} |p|)$ time in the worst case).

5.8 The Aho-Corasick automaton

The Aho-Corasick automaton augments the trie for P with a supply function. Except for the root it holds that for each node x, the *supply link* $S(x)$ points to a node y such that $L(y)$ is the longest proper suffix of $L(x)$ that is represented by a trie node (that means of *some* string in the input set). $S(x)$ can be the root. The root represents the empty string, i.e., $L(root) = \epsilon$.

The supply links can be computed in $O(\sum_k |p^k|)$ time while constructing the trie. They are used to perform safe shifts in the Aho-Corasick algorithm.

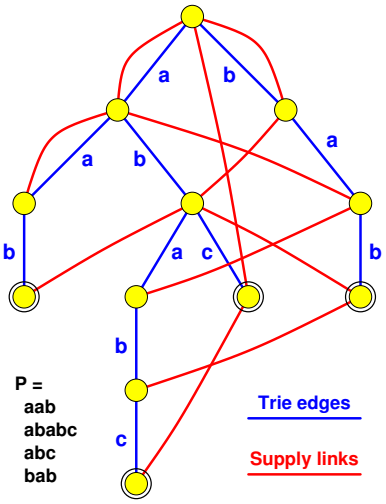
Here is an example. The patterns are

$$P = \{\mathsf{aab}, \mathsf{ababc}, \mathsf{abc}, \mathsf{bab}\}$$

Let's conduct a multiple string matching on

$$T = \mathsf{abaababcbabcbb}\,.$$

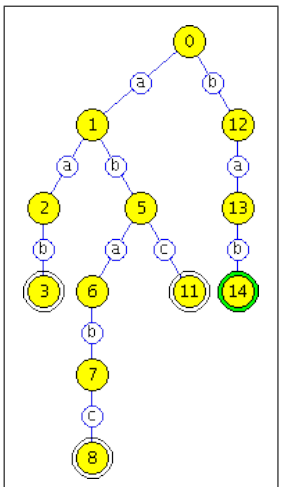
You can see that it really pays off not to search individually for the four strings.




```
/* m[p] = length of pat[p] */
/* #pat = number of patterns */

final = empty set
FOR p = 1 TO #pat
  q = root
  FOR j = 1 TO m[p]
    IF g(q,pat[p][j]) == NULL
      insert(q,pat[p][j])
    ENDIF
    q = g(q,pat[p][j])
    /* visualization point */
  ENDFOR
  final = union(final, q)
ENDFOR
```

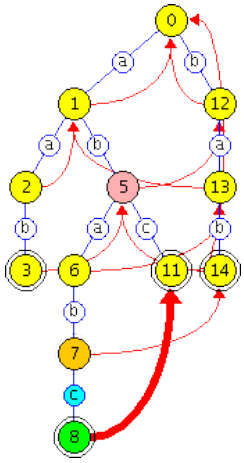
Code to create trie



Generated trie

```
h(root) = fail
FORALL {q' | parent(q') == root}
  h(q') = root
  /* visualization point */
ENDFOR
FORALL {q' | depth(q')>1} in BFS order
  q = parent(q')
  c = the symbol with g(q,c) == q'
  r = h(q)
  /* visualization point */
  WHILE r != fail AND g(r,c) == fail
    r = h(r)
    /* visualization point */
  ENDWHILE
  IF r == fail
    h(q') = g(root,c)
    /* visualization point */
  ELSE
    h(q') = g(r,c)
    IF isElement(h(q'), final)
      final = union(final, q')
    ENDIF
    /* visualization point */
  ENDIF
ENDFOR
```

Code to add supply links



Aho-Corasick automaton

```
q = root
FOR i = 1 TO n
  WHILE q != fail AND g(q, text[i]) == fail
    q = h(q)
    /* visualization point */
  ENDWHILE
  IF q = fail
    q = root
    /* visualization point */
  ELSE
    q = g(q, text[i])
    /* visualization point */
  ENDIF
  IF isElement (q, final)
    RETURN TRUE
  ENDIF
ENDFOR
RETURN FALSE
```

Code to perform search

A "translation table" for the animation

Animation code	Our code
q' (green)	current
q (orange)	parent
h	S
r (pink)	down
fail	θ
c (cyan)	σ

Matches:

```
abaababcbabcbb
  aab
   ababc
     abc abc
    bab bab
```

Note that when the match for ababc is found, we have to report an occurrence of abc as well. One pattern can be a suffix of another.

The discussion how we just built the example together with the pseudocode follows.

First the trie is constructed, then it is augmented with the supply links by a breadth-first search (= BFS traversal.) See the pseudocode below.

Inductively can assume that we have already computed the supply links of all states before state *current* in BFS order. Now consider the parent *parent* of *current* in the trie and assume that it leads to *current* via the symbol σ. That means, $\delta(parent, \sigma) = current$. The supply link for *parent* has already been computed. It points to $S(parent)$.

Let $v = L(current)$ and let u be the longest proper suffix of v that labels a path in the trie, or let $u = \epsilon$ (the empty string). The string v has the form $v'\sigma$, for some string v'. In fact, we have $v' = L(parent)$.

Thus, if u is non-empty, it must have the form $u = u'\sigma$, for some string u', and in that case u' is a proper suffix of v' that is the label of a path in the trie. In fact, we have $u' = L(S(parent))$.

Now, if $S(parent)$ *does have* an outgoing transition labeled by σ to a state h, i.e., $\delta(S(parent), \sigma) = h$, then $u'\sigma = L(S(parent))\sigma$ is the longest suffix of v that is the label of a path, namely $u'\sigma = L(h)$. Hence $S(current)$ has to be set to h.

If $S(parent)$ *does not have* an outgoing transition labeled by σ, then we consider the second-longest proper suffix of v' that is the label of a path. That is, we consider $S(S(parent))$, and if that fails, then $S(S(S(parent)))$, and so on.

We repeat that until *either* we find a non-empty suffix of u that that has an outgoing transition labeled with σ or we reach the initial state.

```
(1)  Build_AC(P = p^1, ..., p^r);
(2)  AC_trie = Trie(P); δ is transition function, root initial state
(3)  S(init) = θ;
(4)  for current in BFS order do
(5)      parent = par(current);
(6)      σ = label of transition from parent to current;
(7)      down = S(parent);
(8)      while down ≠ θ ∧ δ(down, σ) = θ do
(9)          down = S(down);
(10)     od
(11)     if down ≠ θ then
(12)                 S(current) = δ(down, σ);
(13)                 if S(current) = terminal then
(14)                                          mark current as terminal;
(15)                                          F(current) = F(current) ∪ F(S(current));
(16)                 fi
(17)             else
(18)                 S(current) = root;
(19)     fi
(20) od
```

Searching in the Aho-Corasick automaton is straightforward and very similar to the building phase. While scanning the text, we walk through the automaton. Whenever we enter a terminal state we report the set of strings as matching. If we cannot walk on, we follow supply links to find a safe shift.

5.9 Running time

We start with the search phase, since the time bound is a bit easier to obtain for it than for the preprocessing.

Lemma.
The search phase for the Aho-Corasick algorithm (just reporting whether there exists a match or not) takes $O\big(|T|\big)$ time.

Proof. We need to bound the number of transitions within the AC automaton. Let σ be the next letter in T. If the transition to $\delta(current, \sigma)$ is defined, then $L(current)$ increases by 1. Otherwise, $L(current)$ will become strictly smaller. Since $0 \leq |L(current)|$, it follows that, during the whole search over T, at most $|T|$ times a supply link is being used – we cannot walk more often upward than downward in the automaton. ∎

Lemma.
The preprocessing for the Aho-Corasick algorithm takes $O\big(\sum_{p\in P} |p|\big)$ time.

Proof. The argument is the similar to the search argument. After the construction of the trie, we conduct a BFS search. Both clearly need time $O\big(\sum_{p\in P} |p|\big)$.

During the construction of the supply links, we process a total of $\sum_{p\in P} |p|\big)$ characters. For each processed character we spend constant work except for following already inserted supply links. So in total, we cannot follow more supply links than there are characters in the patterns.

As a final comment, one can implement the set union in line (15) by a simple concatenation of lists in constant time, since the sets are disjoint.

5.10 'Advanced' Aho-Corasick algorithm

We can further speed up the *search* phase the AC algorithm if we are willing to spend more time for an extended preprocessing.

Note that it is possible to precompute all transitions implied by the supply links in advance. Essentially, the formula is

$$\delta(current, \sigma) = \delta(\mathcal{S}(current), \sigma)$$

for all σ where δ would otherwise be undefined.

The extended preprocessing pays of for relatively small sets of patterns and for small alphabets. However, we need more space to represent δ.

A compromise is to compute the additional edges "on the fly", when they are "discovered" for the first time. This technique is also known as path compression. We can simply add the compressed transitions to the hash table representing δ. Then the next time we will not need to follow supply links, but will be directly forwarded to the destination. This strategy was implemented in the first version of the well-known unix application `grep`.

4 Fast filtering algorithms

This exposition is based on the following sources, which are all recommended reading:

1. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, chapter 6.5, pages 162ff.
2. Burkhardt et al.: q-gram Based Database Searching Using a Suffix Array (QUASAR), RECOMB 99

We will present the hierarchical filtering approach of Navarro and Baeza-Yates and and the simple QUASAR idea.

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 that to verify that it does.

Filtering algorithms *filter out* portions of the text that cannot possibly contain a match, and, at the same time, find positions that can possibly match.

These 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.

4.2 The pidgeonhole principle

The idea behind the presented filtering algorithm is very easy. 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 has to 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 $\lfloor a_i k/A \rfloor$ 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 $p_{i_1} \dots p_{i_2}$ matches the text $t_j \dots t_{j+i_2-i_1}$, then the verification has to consider the text area $t_{j-(i_1-1)-k} \dots t_{j+(m-i_1)+k}$, which is of length $m + 2k$.

4.3 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 at positions (indexed by the last character) $9, 10, 11$ in t_1 and *none* in t_2.

4.4 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 (exercise).

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 1 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 1. 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
```

4.5 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.

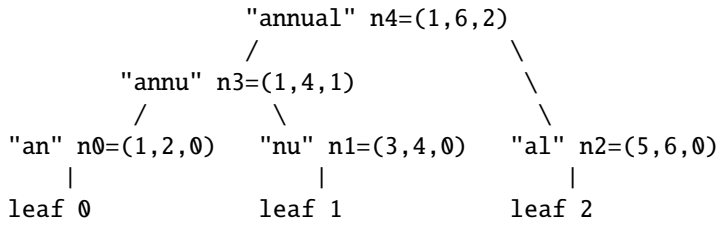
Build Tree: Build a tree of the pattern for the hierarchical verification. If $k + 1$ is not a power of 2, we try to keep the binary tree as balanced as possible.

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 12 and 14 are justified by Lemma 1.)

```
(1)  CreateTree( p = p_i p_{i+1} ... p_j, k, myParent, idx, plen )
(2)  // Note: the initial call is: CreateTree ( p, k, nil, 0, ⌊m/(k+1)⌋ )
(3)  Create new node node
(4)  from(node) = i
(5)  to(node) = j
(6)  left = ⌈(k+1)/2⌉
(7)  parent(node) = myParent
(8)  err(node) = k
(9)  if k = 0
(10)   then leaf_idx = node
(11)   else
(12)        lk = ⌊(left · k)/(k+1)⌋
(13)        CreateTree( p_i ... p_{i+left·plen−1}, lk, node, idx, plen )
(14)        rk = ⌊((k+1−left) · k)/(k+1)⌋
(15)        CreateTree( p_{i+left·plen} ... 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 $(from, to, error)$.



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

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

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) ∈ 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 ≠ 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} ... t_{p_2} for pattern piece p_{from(n)} ... 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 !
```

4.6 Quasar

QUASAR, or "Q-gram Alignment based on Suffix ARrays", is a filtering approach. QUASAR finds all *local* approximate matches of a *query sequence* S in a *database* $D = \{d, \ldots\}$. The verification is performed by other means.

Definition. A sequence d is *locally similar* to S, if there exists at least one pair $(S_{i,i+w-1}, d')$ of substrings such that:

1. $S_{i,i+w-1}$ is a substring of length w and d' is a substring of D, and
2. the substrings d' and $S_{i,i+w-1}$ have edit distance at most k.

We call this the *approximate matching problem with k differences and window length w*.

For simplicity, we assume that the database consists of only one sequence, i. e. $D = \{d\}$.

4.7 The q-gram lemma

A short subsequence of length q is called a *q-gram*. In the following we start by considering the first w letters of S. The algorithm uses the following lemma:

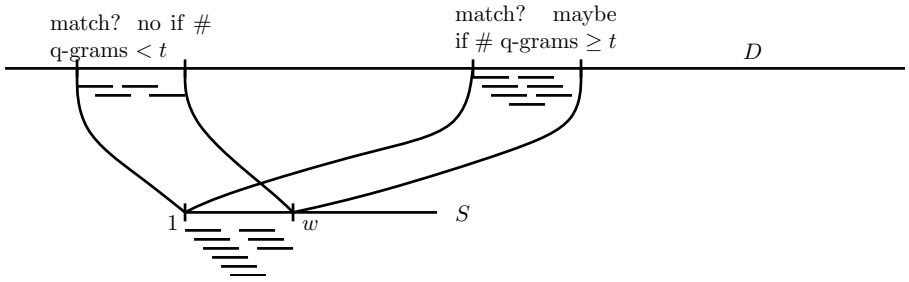
Lemma 2. *Let P and S be strings of length w with at most k differences. Then P and S share at least $w + 1 - (k + 1)q$ common q-grams.*

In our case, this means:

Lemma 3. *Let an occurrence of $S_{1,w}$ with at most k differences end at position j in D. Then at least $w + 1 - (k + 1)q$ of the q-grams in $S_{1,w}$ occur in the substring $D_{j-w+1,j}$.*

Proof: Exercise. . . .

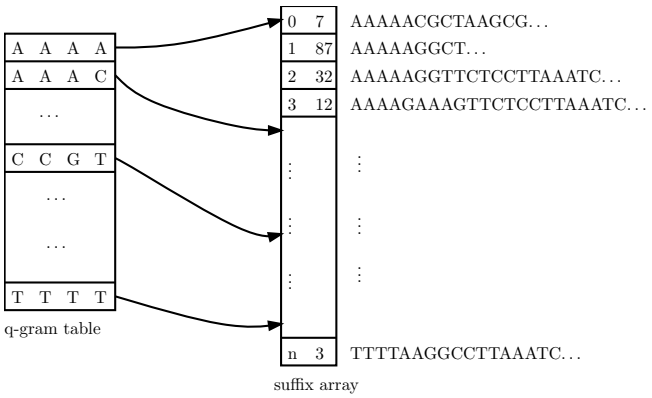
That means that as a necessary condition for an approximate match, at least $t = w + 1 - (k + 1)q$ of the q-grams contained in $S_{1,w}$ occur in a substring of D with length w. For example the strings `ACAGCTTA` and `ACACCTTA` have $8 + 1 - (1 + 1)3 = 3$ common 3-grams, namely `ACA`, `CTT` and `TTA`.



4.8 q-gram index

The algorithm builds in a first step an indexing structure as follows:

1. Build a suffix array A over D.
2. Given q, compute for all possible $|\Sigma|^q$ q-grams the start position of the hitlist. This allows to lookup a q-gram in constant time.
3. If another q is specified, A is used to recompute the above table.

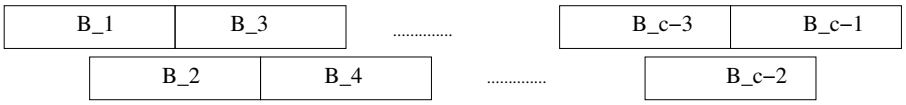


4.9 Counting q-grams

Now we have to find all approximate matches between $S_{1,w}$ and D, that means we have to find all substrings in D that share at least t q-grams with $S_{1,w}$. The algorithm proceeds in the following basic steps on which we will elaborate:

1. Define two arrays of *non-overlapping* blocks of size $b \geq 2w$. The first array is shifted by $b/2$ against the other.
2. Process all q-grams in $S_{1,w}$ and increment the counters of the corresponding blocks.
3. All blocks containing approximate matches will have a counter of at least t. (The reverse is not true).
4. Shift the search window by one. Now we consider $S_{2,w+1}$.

4.10 Blocking



Since we want to count the q-grams that are in common between the query and the database, we use counters. Ideally we would use a counter of size w for each substring of this size. Since this uses too much memory, we build larger, non-overlapping blocks. While this decreases the memory usage, it also decreases the specificity.

Since the blocks are not overlapping we might miss q-grams that cross the block boundary. As a remedy, we use a second, shifted array of blocks.

4.11 Window Shifting

We started the search for approximate matches of window length w with the first w-mer in S, namely $S_{1,w}$. In order to determine the approximate matches for the next window $S_{2,w+1}$, we only have to discard the old q-gram $S_{1,q}$ and consider the new q-gram $S_{w-q+2,w+1}$.

To do that we decrement the counters of all blocks that contain $S_{1,q}$ that have not reached the threshold t. However, if the counter has already reached t it stays at this value to indicate a match for the extension phase.

For the new block we use the precomputed index and the suffix array to find the occurrences of the new q-gram and increment the corresponding block counters (at most two).

4.12 Alignment

After having computed the list of blocks, QUASAR uses BLAST to actually search the blocks. Here are some results from the inital implementation. QUASAR was run with $w = 50$, $q = 11$, and t such that windows with at most 6% differences are found. Reasonable values for the block size are 512 to 4096.

DB size	query	id. res.	filtr. ratio	QUASAR	BLAST
73.5 Mb	368	91.4%	0.24%	0.123 s	3.27 s
280 Mb	393	97.1%	0.17%	0.38 s	13.27 s

"A database in BLAST format is built in main memory which is then passed to the BLAST search engine. The construction of this database requires a significant amount of time and introduces unnecessary overhead."

4.13 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 pidgeonhole 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.