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:

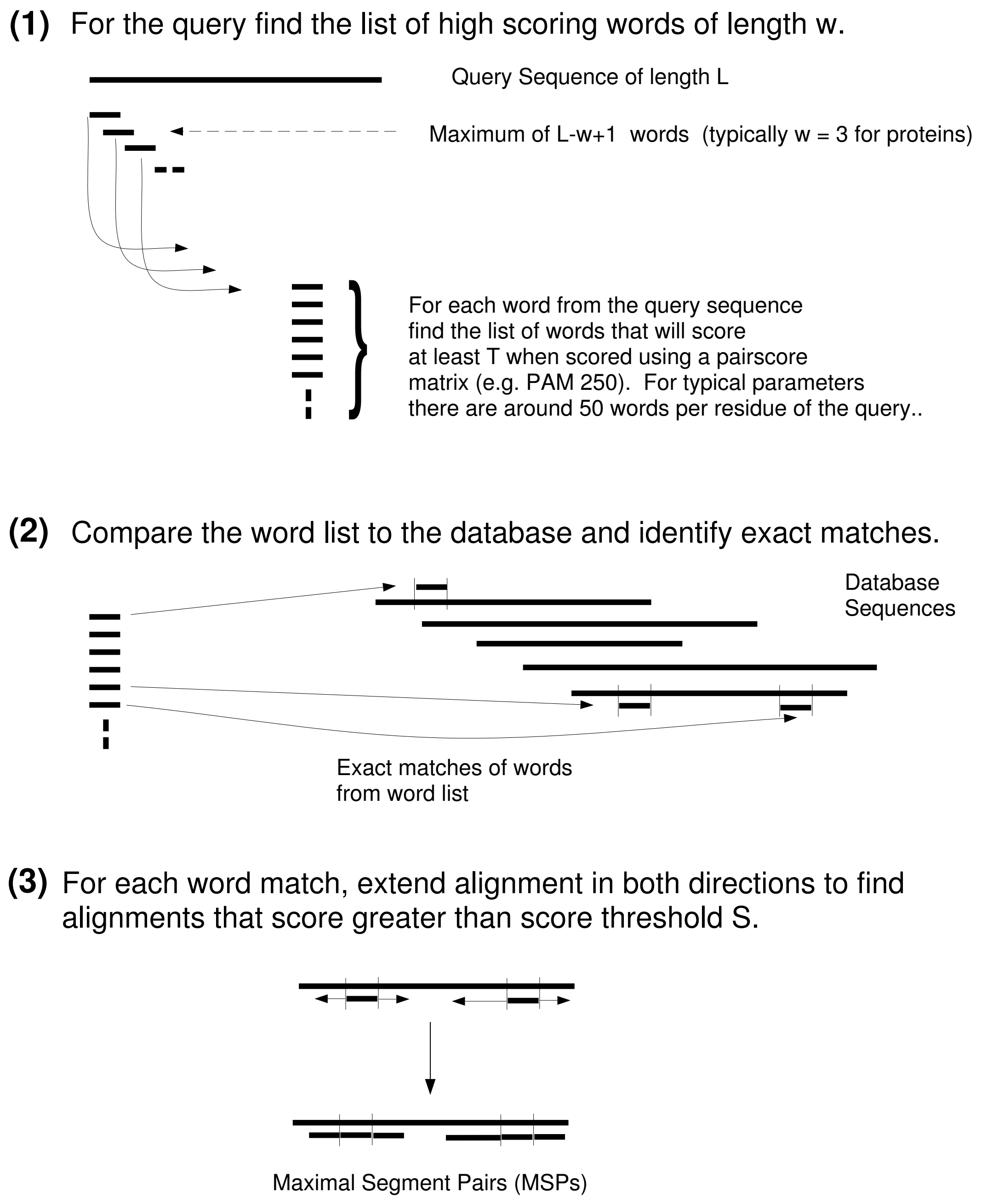
q	l	n	f	s	a	g	w
q	l						
	l	n					
		n	f				
			f	s			
				s	a		
					a	g	
						g	w

$\Longrightarrow$

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.
Query Sequence of length L
Maximum of L-w+1 words (typically w = 3 for proteins)
For each word from the query sequence find the list of words that will score at least T when scored using a pairscore matrix (e.g. PAM 250). For typical parameters there are around 50 words per residue of the query..
(2) Compare the word list to the database and identify exact matches.
Database Sequences
Exact matches of words from word list
(3) For each word match, extend alignment in both directions to find alignments that score greater than score threshold S.
Maximal Segment Pairs (MSPs)

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 $p^k \in P$.

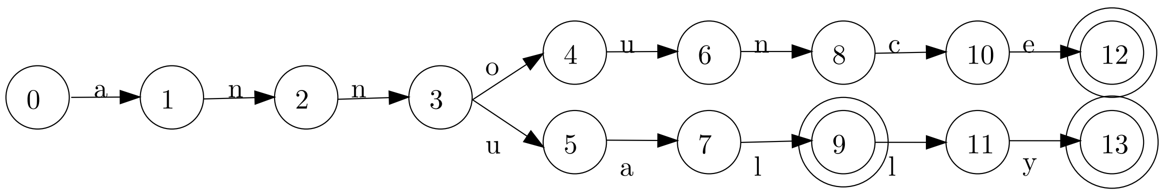
The Aho-Corasick algorithm maintains a data structure, called the *Aho-Corasick automaton*, to keep track of the longest prefix of some pattern that is also a suffix of the text window.

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: $P = \{$`annual`, `annually`, `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($P = p^1, \dots, p^r$);
(2)  Create an initial non-terminal state $root$
(3)  for $i \in 1 \dots r$ do
(4)      $current = root$;  $j = 1$;
(5)      while $j \le m_i \wedge \delta(current, p^i_j) \ne \theta$ do
(6)          $current = \delta(current, p^i_j)$;  $j{+}{+}$;
(7)      od
(8)      while $j \le m_i$ do
(9)          create state $state$;
(10)         $\delta(current, p^i_j) = state$;  $current = state$;  $j{+}{+}$;
(11)     od
(12)     if $current$ is terminal  then
(13)                  $F(current) = F(current) \cup \{i\}$;
(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.

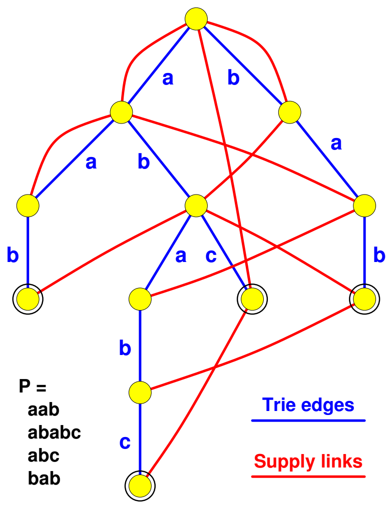
Here is an example. The patterns are

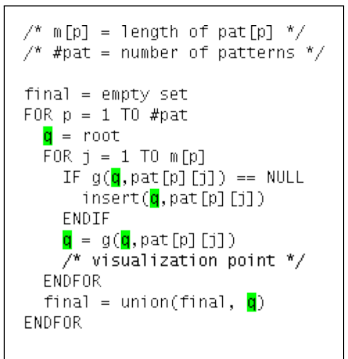
$$P = \{\mathtt{aab}, \mathtt{ababc}, \mathtt{abc}, \mathtt{bab}\}$$

Let's conduct a multiple string matching on

$$T = \mathtt{abaabababcbabcbb}\,.$$

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

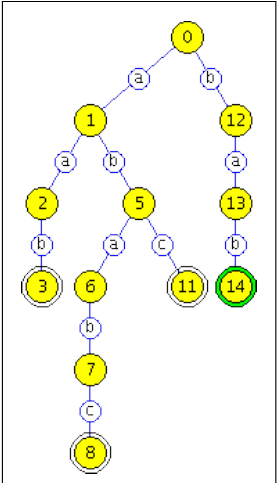




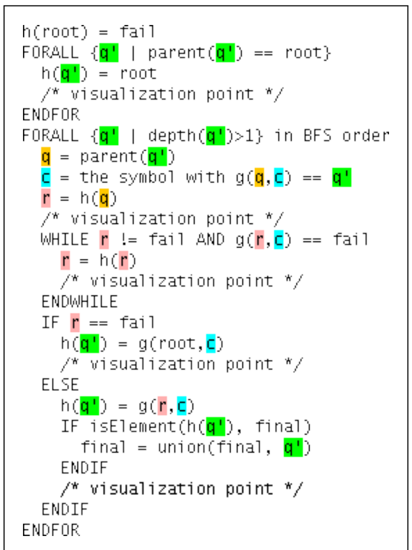
```
/* m[p] = length of pat[p] */
/* #pat = number of patterns */

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

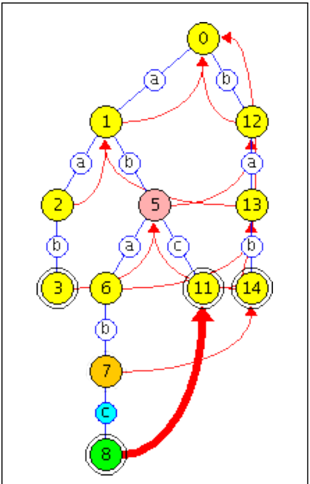


Generated trie

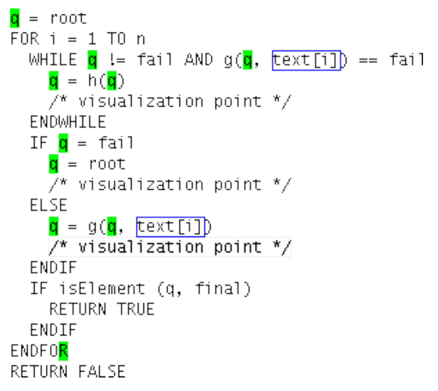


```
h(root) = fail
FORALL {q' | parent(q') == root}
  h(q') = root
  /* visualization point */
ENDFOR
FORALL {q' | depth(q')>1} in BFS order
  q = parent(q')
  c = the symbol with g(q,c) == q'
  r = h(q)
  /* visualization point */
  WHILE r != fail AND g(r,c) == fail
    r = h(r)
    /* visualization 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	$\emptyset$
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.

```
Build_AC($P = p^1, \dots, p^r$);
$AC\_trie = Trie(P)$; $\delta$ is transition function, $root$ initial state
$S(init) = \theta$;
for $current$ in BFS order do
    $parent = par(current)$;
    $\sigma$ = label of transition from $parent$ to $current$;
    $down = S(parent)$;
    while $down \neq \theta \wedge \delta(down, \sigma) = \theta$ do
        $down = \mathcal{S}(down)$;
    od
    if $down \neq \theta$ then
        $S(current) = \delta(down, \sigma)$;
        if $S(current) = terminal$ then
            mark $current$ as terminal;
            $F(current) = F(current) \cup F(S(current))$;
        fi
    else
        $S(current) = root$;
    fi
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 $O\Big(\sum_{p \in P} \big|p\big|\Big)$ 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 $O\Big(\sum_{p \in P} \big|p\big|\Big)$.

During the construction of the supply links, we process a total of $\sum_{p \in P} \big|p\big|\Big)$ 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`.