5 BLAST

- 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
- An earlier version of this lecture by Daniel Huson.
- A java applet visualizing the Aho-Corasick can be found at: `http://www-sr.informatik.uni-tuebingen.de/ buehler/AC/AC1.html`

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

5.2 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* and a *database*.

Let q be the query and d the database. 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 S, a segment pair (s, t) is called a *high-scoring segment pair (HSP)*, if it is locally maximal and $\sigma(s, t) \geq S$.

Finally, a *word* is simply a short substring.

Given S, the goal of BLAST is to compute all HSPs.

5.3 BLAST algorithm for protein sequences

Given three parameters, i.e. a word size K, a word similarity threshold T and a minimum match score S.

For *protein* sequences, BLAST operates as follows:

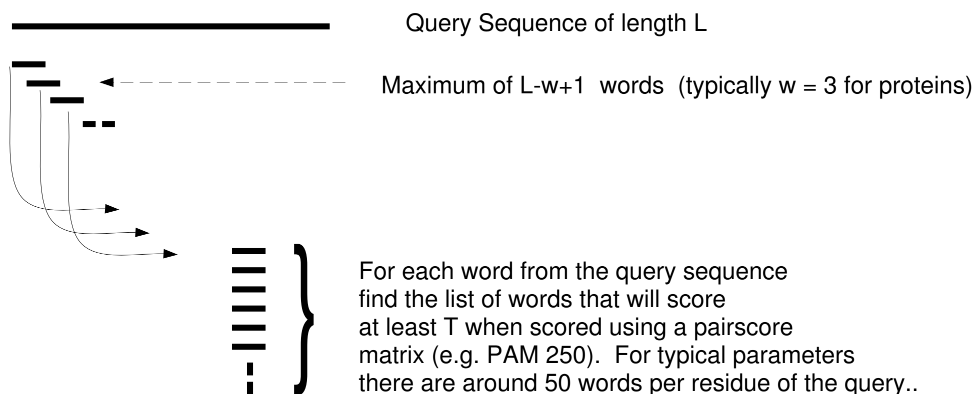
1. The list of all words of length K that have similarity $\geq T$ to some word in the query sequence q is generated.
2. The database sequence d is scanned for all hits t of words s in the list.
3. Each such *seed* (s,t) is extended until its score $\sigma(s,t)$ falls a certain distance below the best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

In practice, K is around 3 for proteins.

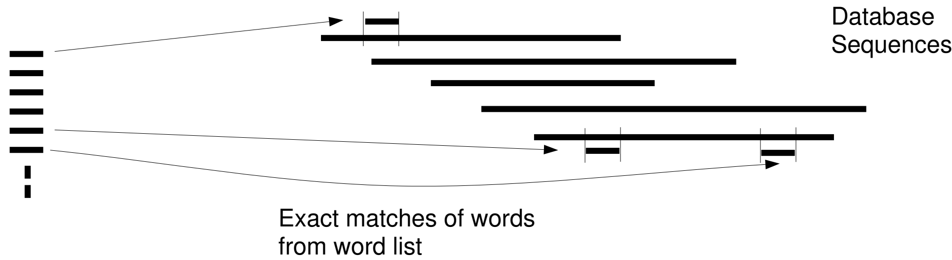
Words of length 2:

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

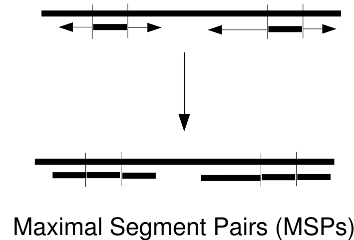
Initial Word	Expanded List
ql	ql, qm, hl, zl
ln	ln, lb
nf	nf, af, ny, df, qf, ef, gf, hf, kf, sf, tf, bf, zf
fs	fs, fa, fn, fd, fg, fp, ft, fb, ys
sa	(nothing scores 8 or higher)
ag	ag
gw	gw, aw, rw, nw, dw, qw, ew, hw, iw, kw, mw, pw, sw, tw, vw, bw, zw, xw



(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 care, the list of all words of length K 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 K 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 K 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 a number of internet services is available, e.g.,

`http://www.ncbi.nlm.nih.gov/BLAST/`

`http://blast.wustl.edu/`

6 Multiple String Matching

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

1. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, chapter 3.

We will present two very practical *multiple* string matching algorithms with sublinear average running times.

6.1 Thoughts about 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 $T = t_1, \ldots, t_n$ usually with $n \gg m_i$. Each p^i is a string $p^i = p^i_1, \ldots, p^i_{m_i}$. We denote with $|P|$ the sum of the lengths of all patterns, i. e. $|P| = \sum_{i=1}^{r} \left|p^i\right|$.

As with single string matching we have three basic approaches:

1. *Prefix searching*. We read each character of the string with an automaton built on the set P. For each position in the text we compute the longest prefix of the text that is also a prefix of one of the patterns.
2. *Suffix searching*. A positions *pos* is slid along the text, from which we search backward for a suffix of any of the strings.
3. *Factor searching*. A position *pos* is slid along the text from which we read backwards a factor of some prefix of size *lmin* of the strings in P.

6.2 Prefix based approaches

The extension of the prefix based approaches leads to the *Multiple Shift-And* and *Aho-Corasick* algorithms. As with a single pattern 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 a prefix of a pattern $p^k \in P$.

The *Shift-And* algorithm can been extended to multiple patterns. It is practical if all the patterns fit into a couple of computer words.

Among prefix based approaches, the *Aho-Corasick* respectively the *advanced Aho-Corasick* algorithm are to be preferred. They are generalizations of the Knuth-Morris-Pratt algorithm.

Idea of the Shift-And algorithm for multiple patterns:
Recall that the Shift-And algorithm maintains bitmasks B for each character of the alphabet and a bit-vector D which has a 1 at the j-th position iff $P[1..j]$ is a suffix of $T[1..pos]$, where *pos* denotes the end of the search window. Initially $D = 0^m$. The heart of the algorithm is the update formula

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

and a match is recognized if

$$D \ \& \ \underline{10^{m-1}} \neq 0^m$$

is true.

For multiple patterns $p^1, \ldots, p^r$, we preprocess the bit-masks B for their concatenation. Let $m_i := |p_i|$. Then, essentially, the pseudo-code for the Shift-And algorithm is obtained by replacing

- $0^{m-1}1$ with $DI := 0^{m_r-1}1 \ldots 0^{m_1-1}1$, and
- 10^{m-1} with $DF := 10^{m_r-1} \ldots 10^{m_1-1}$.

Of course, these bit masks DI, DF need to be computed only once.

The Shift-Or trick cannot help us here, since the $\ll$ operation only introduces *one* zero to the right, but we need a 0 in *each* position that begins a new pattern in the computer word.

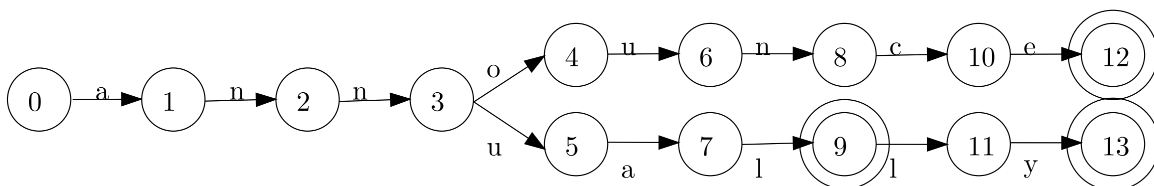
Some extra care has to be taken when reporting the actual matches and their positions.

Idea of the Aho-Corasick algorithm:
The Knuth-Morris-Pratt algorithm keeps track of the longest prefix of the pattern that is also a suffix of the text window. The Aho-Corasick algorithm extends this idea to a set of patterns. It 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.

6.3 The trie data structure

A *trie* is a compact representation of a set of strings. It is a rooted, directed tree that represents the set P. 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, $\delta(cur, p^i_j)$ denotes the successor state reached from state *cur* 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.)

The size of the trie and the running time of its basic operations depend on the implementation of the transition function δ. The fastest implementation uses for each node a table of size $|\Sigma|$. If the alphabet is too large or space is at a premium, one can bound the total space used to code for the transitions by $O(|P|)$. However one has to pay for the reduced space requirement by increasing the $O(1)$ time for table access to $O(\log |\Sigma|)$ or by using a hash function.

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

6.4 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 state v, the *supply link* $S(v)$ points to a node w such that $L(w)$ is the longest proper suffix of $L(v)$ that is represented by a trie node (that means of *some* string in the input set). $S(v)$ can be the root, which represents the empty string.

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.

Lets have a look at an example. Assume we are given the set $\{ATATATA, TATAT, ACGATAT\}$.

(Blackboard)

Lets look at our example and try to conduct a multiple string matching on $T = CGACGATATATAGC$. You can see that it really pays off not to search individually for the three strings.

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

Let have a look at how the trie is augmented via the supply links. We built them via a BFS traversal of the trie after we constructed it.

Inductively assume that we have computed the supply links of all states before state *cur* in BFS order. Now consider the parent *par* of *cur* in the trie which leads to *cur* via the symbol σ (that means $\delta(par, \sigma) = cur$). Note that the supply link for *par* has already been computed.

Now we search for the state where u ends, u being the longest suffix of $v = L(cur)$ that labels a path in the trie. The string v has the form $v'\sigma$. If there exists a nonempty string u, it must be of the form $u = u'\sigma$ (since it is a suffix of v).

In that case u' is a suffix of v' that is the label of a path in the trie.

Now, if $S(par)$ has an outgoing transition labeled by σ to a state h, then $w = L(S(par))$ is the longest suffix

of v' that is the label of a path and $w\sigma$ is also a label of a path in the trie.

Hence it is the longest suffix u of $v = v'\sigma$ that we were searching for, and $S(cur)$ has to be set to h.

If $S(cur)$ does not have an outgoing transition labeled by σ, we consider $S(S(cur))$ and so on. We repeat that until we find a state 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 cur in BFS order do
5      par = par(cur);
6      σ = label of transition from par to cur;
7      down = S(par);
8      while down ≠ θ ∧ δ(down, σ) = θ do
9              down = S(down);
10     od
11     if down ≠ θ then
12                     S(cur) = δ(down, σ);
13                     if S(cur) = terminal then
14                                         mark cur as terminal;
15                                         F(cur) = F(cur) ∪ F(S(cur));
16                     fi
17                 else
18                     S(cur) = 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.

There is a nice java applet on the web, illustrating the Aho-Corasick algorithm:
`http://www-sr.informatik.uni-tuebingen.de/~buehler/AC/AC1.html`

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

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

for all σ where δ would otherwise be undefined.

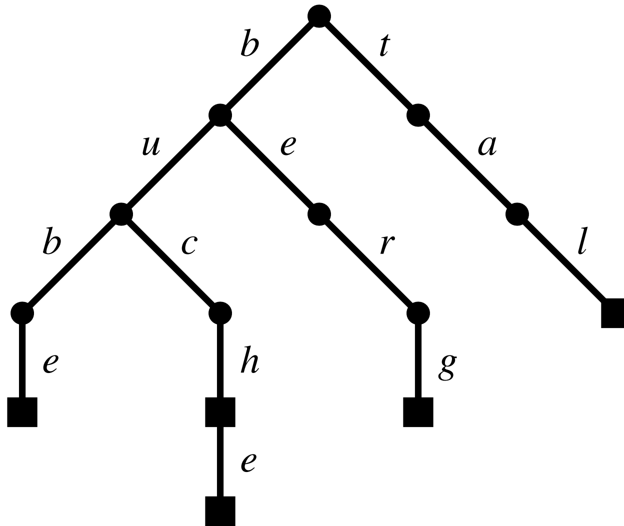
The extended preprocessing pays of for relatively small sets of patterns and for small alphabets. However, we need more space to represent δ.

A compromise is to compute the additional edges "on the fly", when they are used for the first time. (This technique is known as path compression.) This was done in the first version of the well-known unix application `grep`.

Note: the following slides may help as well, but they use a bit different notation ... I will fix that later.

Definition 1. A *trie* for a set of patterns $\{P_1, \ldots, P_N\} \subseteq \Sigma^*$ is a tree $K = (V, E)$ with a root root(K), in which the edges are directed away from the root and labeled with letters. All outgoing edges of a node have different labels. The string formed by the letters along the edges of the path from the root to a node v is called $L(v)$. ($L(\text{root}) = \varepsilon$.) If $L(v) = P_p$, then v carries a mark $\text{pat}(v) = p$, otherwise $\text{pat}(v) = 0$. All L(v) are prefixes of patterns and each pattern P_p appears as $P_p = L(v)$ for some $v \in V$. (Then it is easy to see that $v \mapsto L(v)$ is an injective map.)

Example. Trie for {berg, bube, buch, buche, tal}:



Definition 2. Let $K = (V, E)$ be a trie for $\{P_1, \ldots, P_N\}$ and $v \in V(K)$.

- The *successor function* $\text{succ} : V \times \Sigma \to V \cup \{\text{fail}\}$ is defined by

$$\text{succ}(v, a) := \begin{cases} w \in V\,, & \text{where } L(v)a = L(w)\text{, or} \\ \text{fail}\,, & \text{if no such } w \text{ exists.} \end{cases}$$

- The *failure links* flink : $V \to V$ are defined by $\text{flink}(v) := w$, where $L(w)$ is the longest prefx of a pattern P_p that is also a proper suffix of $L(v)$ ist. (Special case: flink(root) = root; note that ε has no proper suffix.)
- Moreover each node v can have an *output link* $\text{olink}(v) \in V \cup \{\text{fail}\}$. If available, the ouptut link points to the next node w with $\text{pat}(w) \neq 0$ that can be reached from v using failure links.

The AC algorithm maintains the following invariant at the end of each loop: $L(v)$ *is the longest prefix of a pattern that is also a suffix of* $T[1 .. i-1]$.

AC-Algorithm, Searching

```
    i = 1                                              // i runs over T
v = root                                  // L(v) runs over prefixes of patterns
while (i ≤ n)
    while (v ≠ root) and (succ(v, T[i]) = fail)
        v = flink(v)                                   // jump backward in K
    if (succ(v, T[i]) ≠ fail)
        w = v = succ(v, T[i])                          // step in K
        if (pat(v) ≠ fail)                             // output matches
            print (i − |P_pat(v)|, pat(v))
        while (olink(w) ≠ fail)
            print (i − |P_pat(w)|, pat(w))
            w = olink(w)
    i = i + 1                                          // step in T
```

The preprocessing is necessary to augment the trie for the patterns with failure and output links. They are computed by a breath-first traversal.

AC-Algorithm, Preprocessing

```
    flink(root) = root
olink(root) = fail
for all v ∈ V in BFS order
    let L(v) =: L(v′)a                              // v′ earlier visited, a ∈ Σ
    w = flink(v′)
    while (w ≠ root) and (succ(w, a) = fail)
        w = flink(w)
    if (succ(w, a) = fail) or (succ(w, a) = v)      // assign flink
    if (w ≠ root)                                   // assign flink
        flink(v) = root
    else
        flink(v) = succ(w, a)
    if (pat(flink(v)) ≠ fail)                       // assign olink
        olink(v) = flink(v)
    else
        olink(v) = olink(flink(v))
```