Approximate string matching

Main application areas:

- Computational biology
  DNA and protein sequences represented as strings
- Signal processing
  Detection/correction of transmission errors
- Text retrieval
- Intrusion detection
- Knowledge discovery
- Pattern recognition
  Signature recognition/verification
  Handwriting recognition
  Processing and recognition of speech

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Problem

Σ is a finite alphabet of size $|\Sigma| = \sigma$.

$T \in \Sigma^*$ is a text of length $n = |T|$.

$P \in \Sigma^*$ is a pattern of length $m = |P|$.

$k \in \Re$ is a maximum allowable number of errors.

$d : \Sigma^* \times \Sigma^* \to \Re$ is a distance function.

The problem is following: for given *T*, *P*, *k* and *d()* find all text positions *j* for each exists *i*, for which $d(P, T_{i..j}) \leq k$.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Distance functions

Basic operations $\delta(t_{src}, t_{dst})$ transforming string *x* into string *y* are in most cases constricted to:

- Insertion: $\delta(\varepsilon, a)$, i.e. adding a symbol *a*.
- Deletion: $\delta(a, \varepsilon)$, i.e. removing of symbol *a*.
- Substitution: $\delta(a, b)$ for $a \neq b$, i.e. changing of *a* to *b*.
- Transposition: $\delta(ab, ba)$ for $a \neq b$, i.e. changing positions of neighboring symbols *a* and *b*.

Edit distance (*Levenshtein d.*) allows for insertions, deletions and substitutions.
Hamming distance allows only for substitutions.
Episode distance allows only for insertions.
Longest common subsequence distance allows for insertions and deletions.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Computing edit distance – dynamic programming

Initialization: $C_{i,0} = i \quad C_{0,j} = j$

Other values: $C_{i,j} = (P_i == T_j)\,?\,C_{i-1,j-1} : 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

$C_{m,n} = ed(P,T)$

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Finding pattern in text – a dynamic programming approach (~1980)

The beginning of the pattern match must be allowed on any position, so $C_{0,j} = 0$ for all $j \in 0..n$.

The algorithm then initializes its column $C_{0..m}$ with the values $C_i = i$ and processes the text character by character. At each new text character T_j, its column vector is updated to $C'_{0..m}$. The update formula is the same as in edit distance computation:

$$C'_i = (P_i == T_j)? \quad C_{i-1} \quad : \quad 1 + \min(C'_{i-1}, C'_i, C_{i-1})$$

The algorithm returns all positions for which $C_m \leq k$.

It is not fast solution, but it is very flexible and can be adapter to different distance functions.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Example

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

The search time of this algorithm is $O(mn)$ and its space requirement is $O(m)$.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Cut-off heuristics - Ukkonen 1985

Observation: pattern is most cases not present in a text.

We can expect that values in columns will quickly reach $k+1$, i.e. mismatch.

Let's call a cell an active cell, when its value is at most *k*. The algorithm simply tracks the last active cell and skips computing values of the rest of cells.

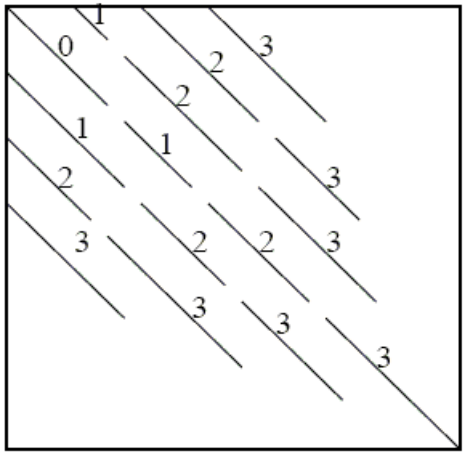
		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

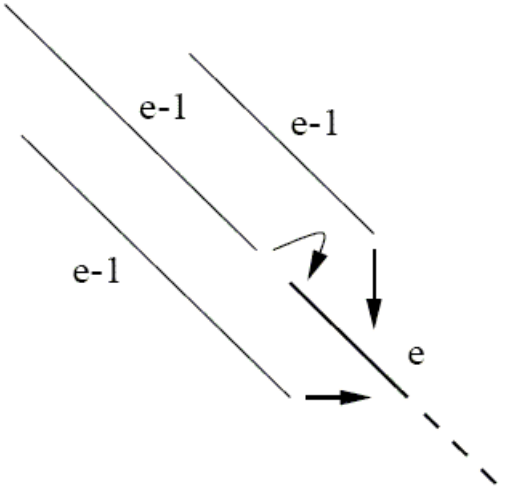
Average search time is $O(kn)$.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Diagonal transition algorithms

Observation: that the diagonals of the dynamic programming matrix (running from the upper-left to the lower-right cells) are monotonically increasing. More than that: $C_{i+1,j+1} \in \{C_{i,j}, C_{i,j}+1\}$.
The algorithm is based on computing **in constant time** the positions where the values along the diagonals are incremented.





Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Algorithms based on automata

Non-deterministic finite-state automaton or NFA consists of:

- Alphabet Σ of *input symbols*
- Set of *states*, in which we discern:
  - One *initial state*
  - Set of *final states*
- Transition relation, which for given state and input symbol defines set (possibly empty) of succeeding states.

Each NFA corresponds to a regular expression and a regular expression can be relatively easy converted into NFA.
What's more NFA can be transformed into Deterministic Automaton.

Regular expression ↔ NFA ↔ DFA

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

NFA Operation

We start in initial state.

In a given state we select transition to a succeeding state marked with a current input symbol. If from a current state a transition marked with empty symbol ε is outgoing we can select this transition without consuming input symbol.

If more than one transition is available we select a transition at random.

We repeat until there are no more input symbols or we reach one of the final states.

If we reached one of final states and there are no input symbols we have pattern recognized.

In other case we have to return to the last state in which we've made random transition choice and try different path.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Deterministic Finite-State Automata

NFA are useful because we can easily build then from regular expressions. Random transition selection which makes re-entries (i.e. return to some state with un-getting input symbols) possible are questionable in implementation. In implementation much better choice are *Deterministic Finite-State Automata - DFA*, which are a special case of NFA, in which:

1. There are no transitions marked with empty symbol ε.
2. In every state there is at most one transition marked with given input symbol.

Each NFA can be transformed into equivalent DFA.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

NFA to DFA Conversion

We join NFA states into unions looking at them from the point of view of input symbols processing:

- Any two states connected with ε-transition will be represented as one DFA state.
- Union of all NFA states reachable from a current state with a given input symbol will define one DFA state.

Let's define two functions:

ε **-closure** function has one argument which is a source state and its value is a set of states reachable from the source state with ε-transitions (note that this means that we have to consider many such transitions: important thing is that no input symbol is processed).

move function has two arguments: a source state and an input symbol; its value is a set of states reachable in one transition with a given input symbol.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

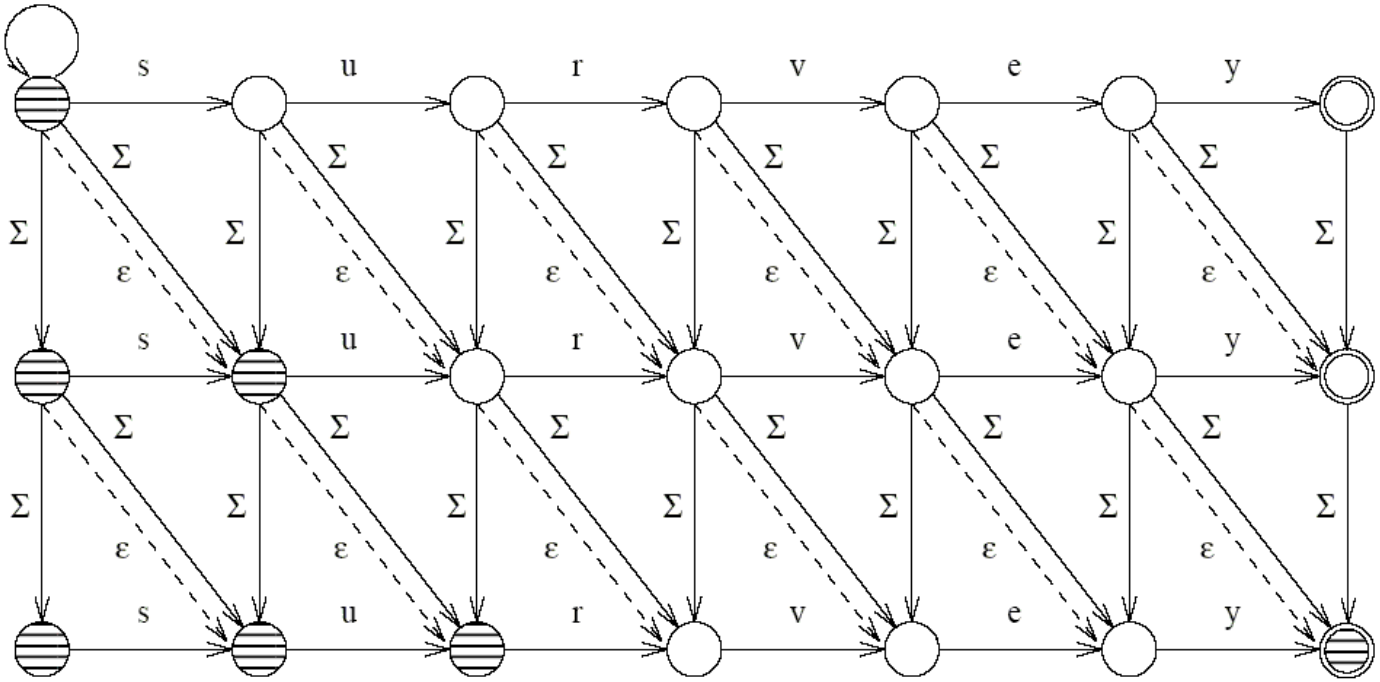
NFA to DFA Conversion Algorithm

1. Create initial DFA state as ε *-closure* of initial NFA state.
2. For each DFA state:
   For each input symbol:
   a. Apply *move* to DFA state (which in general is a set of NFA states) and input symbol – this will give set of NFA states.
   b. Apply ε *-closure* to all elements in this set creating adding new states to the set of NFA states.
   Such a set of NFA states is a (potentially) new DFA state.
3. If new state was created we return to step 2. This process ends when in step 2 no new states are created.
4. As *final states* of DFA, we mark those states that contain any of the NFA final states.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

NFA for approximate search

Every row denotes the number of errors seen. Every column represents matching a pattern prefix. Horizontal arrows represent matching a character. All the others increment the number of the errors: vertical arrows insert a character in the pattern, solid diagonal arrows replace a character, and dashed diagonal arrows delete a character of the pattern.



Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Text preprocessing

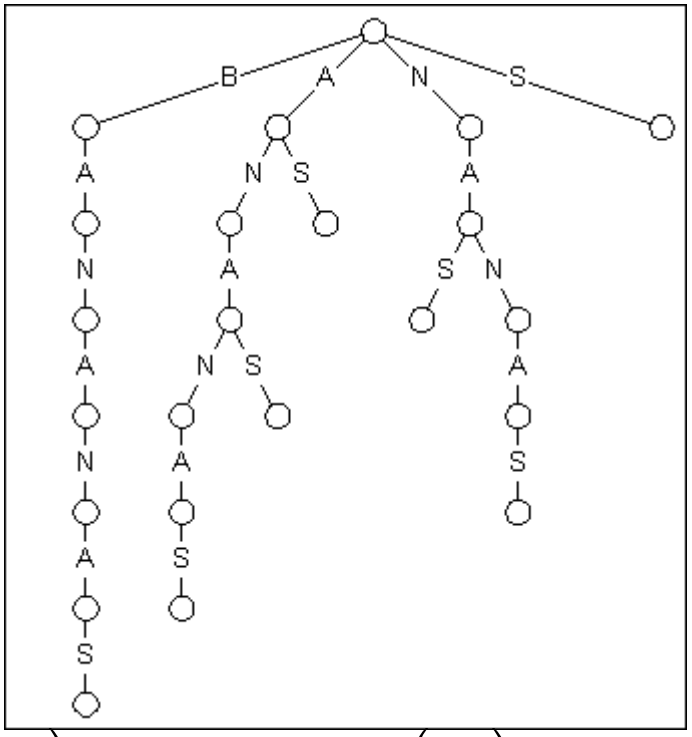
Text is searched often multiple times. It pays off to invest in structures speeding up search in a text.

Data Structure	Search Approach				
	Neighborhood Generation	Partitioning into Exact Searching		Intermediate Partitioning	
		Errors in Text	Errors in Pattern	Errors in Text	Errors in Pattern
Suffix Tree	Jokinen & Ukkonen 91 Ukkonen 93 Cobbs 95		Shi 96		
Suffix Array	Gonnet 88				Navarro & Baeza-Yates 99
Q-grams	n/a	Jokinen & Ukkonen 91 Holsti & Sutinen 94	Navarro & Baeza-Yates 97		Myers 90
Q-samples	n/a	Sutinen & Tarhio 96	n/a	Navarro et al. 2000	n/a

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Suffix trie

Suffix trie is a tree build over **all** suffixes of T.



Suffix trie requires $O(n^2)$ time and $O(n^2)$ memory.
These requirement make such a structure practically useless.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

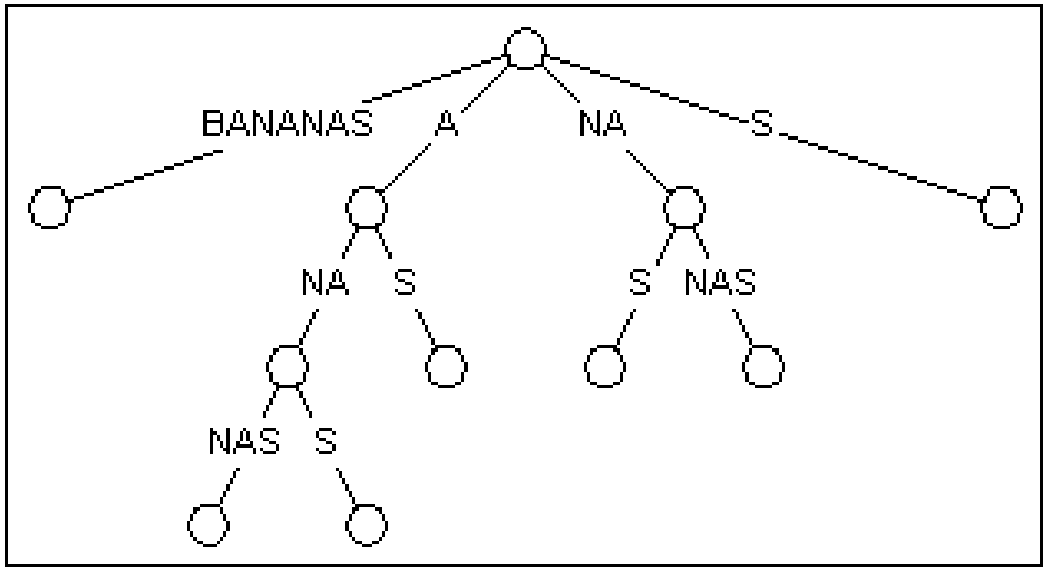
Suffix tree

Suffix trie can be „reduced”.
First, we can see that each tree leaf points to suffix of T. We can compact in one edge lists (i.e. degenerated subtrees) leading to leaves.
Now each internal node represents unique substring of T, which appears in text more than once
Second, we can compact all unary nodes, i.e. nodes which have just one successor.
After these operations tree edges can be labelled with more than one character.
Such a structure is called *Patricia trie* (Practical algorithm to retrieve information coded in alphanumeric).

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Suffix tree

Suffix trie after pruning (i.e. performing *compacting* and *collapsing*) becomes a suffix tree.



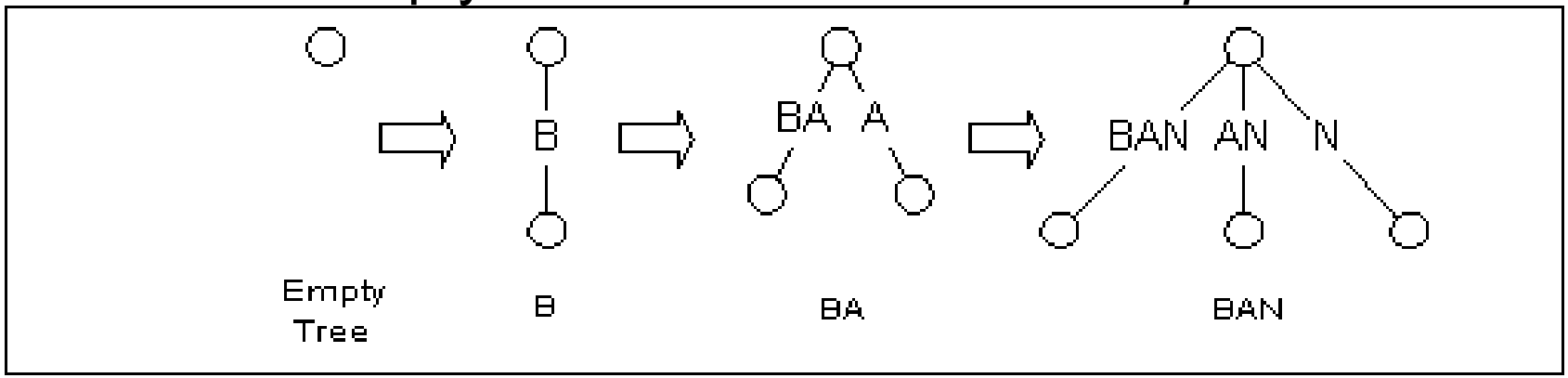
We replace strings on the tree edges with indices pairs (a, b), where a is an index of the beginning of the string, and b is an end index.

This structure has $O(n)$ memory and $O(n)$ construction time requirements.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Construction of the suffix tree (Ukkonen 1995)

We start with an empty tree and we add to it all n *prefixes* of the text.



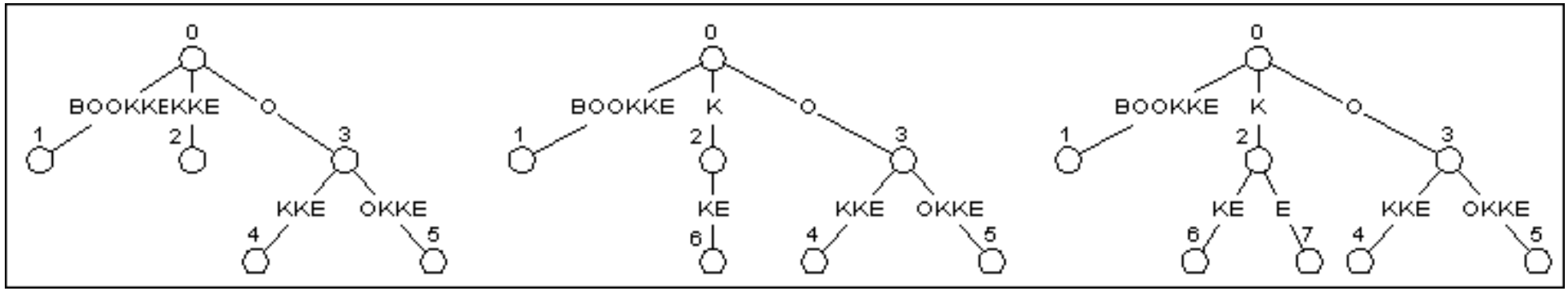
Adding a new prefix requires visiting all the suffixes in the tree. We start with the longest suffix and we move down to the shortest suffix (an empty string). Depending on the suffix node the procedure can end with appending a symbol or in splitting of the node. We can encounter hidden nodes (hidden, because they exists in the middle of an edge) which require splitting.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Construction of the suffix tree

Each suffix ends in the node of one of the following types:

1. Leave.
   Simple update of string labelling an edge leading to this node, but split & add procedure is possible.
2. Explicit node.
   Adding of a new edge to the leaf required.
3. Implicit (hidden) node.
   Split & add procedure required.



Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Suffix array

The suffix array is a weak version of the suffix tree, which requires less space and poses a small penalty over the search time.

If the leaves of the suffix tree are traversed in left-to-right order, all the text suffixes are retrieved in lexicographical order. A suffix array is simply such and ordered array containing all the pointers to the text suffixes.

Text

1	2	3	4	5	6	7	8	9	10	11
a	b	r	a	c	a	d	a	b	r	a

Suffix Array

11	8	1	4	6	2	9	5	7	10	3

Suffix arrays can simulate by binary searching almost every algorithm on suffix trees, at an $O(\log n)$ time penalty factor.

BTW: how to check if pattern p of length m is present in the text having suffix array at hand?

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Q-grams and Q-samples Indices

Yet a weaker (and less space demanding) scheme is to limit the length of the strings that can be directly found in the index. A q-gram index allows retrieval of text strings of length at most q.

In a q-gram index, every different text q-gram (substring of length q) is stored. For each q-gram, all its positions in the text (called occurrences) are stored in increasing text order.

An even less space demanding alternative is a q-sample index, where only some text q-grams (called text q-samples) are stored, and therefore not any text q-gram can be found. The text q-samples, unlike the q-grams, do not overlap, and there may even be some space between each pair of samples.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Neighborhood generation

The number of strings that match a pattern *P* with at most *k* errors is finite, as the length of any such string cannot exceed *m* + *k*. We call this set of strings the *k-neighborhood* of *P*, and denote it $U_k(P) = \{x \in \Sigma^*, d(x, P) \leq k\}$.

The idea of this approach is to generate all the strings in $U_k(P)$ and use an index to search for their text occurrences (**without errors!**). Each such string can be searched for separately or a more sophisticated technique based on string similarity can be used.

The main problem with this approach is that $U_k(P)$ is quite large; it grows exponentially in *k*, e.g. $|U_k(P)| = O(m^k \sigma^k)$. So this approach works well for small *m* and *k*.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Backtracking

The suffix tree or array can be used to find all the strings in $U_k(P)$ that are present in text. Since every substring of the text (i.e. every potential occurrence) can be found by traversing the suffix tree from the root, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch cannot start a string in $U_k(P)$.

We compute the edit distance between our pattern $x = P$ and every text string y that labels a path from the root to a tree node N.

We start at the root with the initial column $C_{j,root} = j$ and recursively descend by every branch of the tree.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Backtracking

Three cases can occur at node N:

1. $C_{m,N} \leq k$, which means that $y \in U_k(P)$ and hence we report all the leaves of the current subtree as answers..
2. $C_{j,N} > k$ for every *j*, which means that y is not a prefix of any string in $U_k(P)$ and hence we can abandon this branch of the tree.
3. If none of the above two cases occur, we continue descending by every branch.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Partitioning into exact search

Each approximate occurrence of a pattern contains some pattern substrings that match without errors. Hence, we can derive sufficient conditions for an approximate match based on exact matching of one or more carefully selected pattern pieces. These pieces are searched for without errors, and the text areas surrounding their occurrences are verified for an approximate occurrence of the complete pattern.

A general limitation of such filtration methods is that there is always a maximum error ratio α up to where they are useful.

Let *A* and *B* be two strings such that $d(A,B) \leq k$. Let $A = A_1 x_1 A_2 x_2 \ldots x_{k+s-1} A_{k+s}$ for strings A_i and x_i for any $s \geq 1$. Then, at least *s* strings $A_{i1} \ldots A_{is}$ appear in *B*.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds

Intermediate partitioning

We filter the search by looking for pattern pieces, but those pieces are large and still may appear with errors in the occurrences. However, they appear with less errors, and therefore we use neighborhood generation to search for them.

Let *A* and *B* be two strings such that $d(A,B) \leq k$. Let $A = A_1 x_1 A_2 x_2 \ldots x_{j-1} A_j$ for strings A_i and x_i for any $j \geq 1$. Let k_i be any set of nonnegative numbers such that $\sum_{i=1}^{j} k_i \geq k - j + 1$. Then, at least one string A_i appears with at most k_i errors in *B*.

Preparing of these materials was supported by a grant from Liechtenstein, Iceland and Norway through EEA Grants and Norway Grants and co-financed by domestic funds