8.1 Approximate Search in Indices

This exposition has been developed by David Weese. It is based on the following source, which is recommended reading:

1. Kärkkäinen, J., and Na, J. C. (2007). Faster filters for approximate string matching, 7, 84–90. Springer

8.2 Definitions

We consider a string T of length n. For $i, j \in \mathbb{N}$ we define:

- $[i..j] := \{i, i+1, \ldots, j\}$
- $[i..j) := [i..j-1]$
- $T[i]$ is the i-th character of T (counting from 0)
- $|T|$ denotes the string length, i. e. $|T| = n$
- In this lecture $T[i..j]$ is a substring *but not* from position i to j (explained later)
- The concatenation of strings X, Y is denoted as $X \cdot Y$

8.3 Introduction

We have seen the effective filters for the approximate string matching problem based on q-gram counting, e.g. QUASAR and SWIFT. These are based on the q-gram lemma that states that a certain number of overlapping q-grams are shared between query and each approximate match.

Another simple but effective family of filters are the so-called *factor filters*, which are based on a factorization of the pattern. A *factorization* of a string S is a sequence of strings (factors) whose concatenation is S.

Example 1 (factorization)**.** A possible factorization of the string $S =$ `GATTACA` would be the sequence of factors `G`, `AT`, `TAC`, `A`.

8.4 Factor Filters

In the following, we will develop filters for the *approximate string matching problem* which is to find all substrings of a text T that are within a distance k of a pattern P. The distance we consider is *edit distance* (a.k.a. Levenshtein distance) under which the approximate string matching problem is also called the *k-difference problem* (Gusfield, 1997).

The simplest factor filter for the k-difference problem is based on the pigeonhole principle:

Theorem 2 (pigeonhole lemma)**.** *Let $A = A_0 A_1 \cdots A_k$ be a string that is the concatenation of $k+1$ non-empty factors A_i. If a string B is within edit distance k from A, then at least one of the factors A_i is a factor of B.*

As a consequence we can divide our pattern P into $k+1$ non-empty factors and an exact text occurrence of one factor signals a potential approximate match.

The specificity of a factor filter is influenced by the expected number of random hits of each factor. The more random text occurrences the factors have, the more false positive matches the filter will output resulting in more unsuccessful verifications in a subsequent step. A stronger filter criterion can be reached with longer but approximate factors, i.e. factors that are searched with errors.

However, before introducing stronger factor filters, we define an optimal factorization.

Theorem 3 (optimal factorization)**.** *For two strings A and B, let $A_0 A_1 \cdots A_{s-1}$ be a factorization of A. Then there exists a factorization $B_0 B_1 \cdots B_{s-1}$ with $dist(A, B) = \sum_{i \in [0..s)} dist(A_i, B_i)$. We call such a factorization of B* optimal.

Proof: Exercise.

Example 4. For the strings

$$\begin{aligned} A &= \texttt{GATTACA} \\ B &= \texttt{ATCACTA} \end{aligned}$$

an optimal factorization would be:

$$\begin{aligned} A &= \texttt{GAT} \cdot \texttt{TA} \cdot \texttt{CA} \\ B &= \texttt{AT} \cdot \texttt{CA} \cdot \texttt{CTA} \quad , \end{aligned}$$

as it holds:

$$\begin{aligned} dist(A,B) &= dist(\texttt{GAT},\texttt{AT}) + dist(\texttt{TA},\texttt{CA}) + dist(\texttt{CA},\texttt{CTA}) \\ 3 &= 1+1+1 \quad . \end{aligned}$$

If we now assign a weight $t_i \in \mathbb{N} \cup \{0\}$ to each factor A_i, we can search approximate matches with less than $t = \sum_{i \in [0..s)} t_i$ errors using the following corollary of theorem 3:

Corollary 5 (weighted pigeonhole lemma)**.** *If $dist(A,B) < t$, then there exists an $i \in [0..s)$ such that $dist(A_i, B_i) < t_i$.*

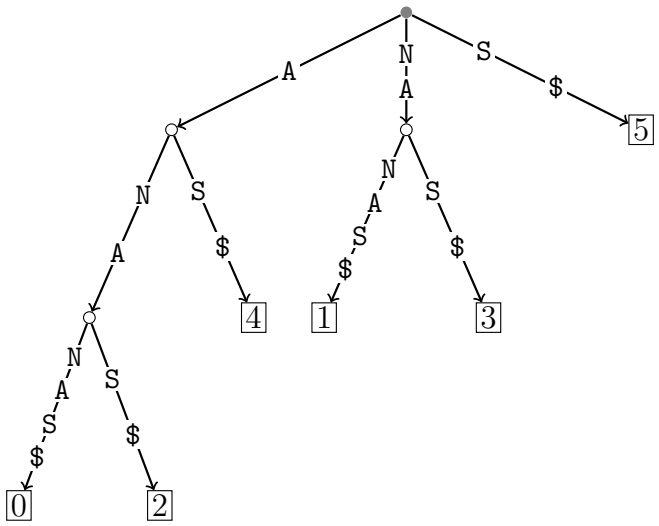
The corollary can be trivially be proven with the contraposition

$$\forall_{i \in [0..s)} dist(A_i, B_i) \geq t_i \;\Rightarrow\; dist(A,B) \geq t \quad .$$

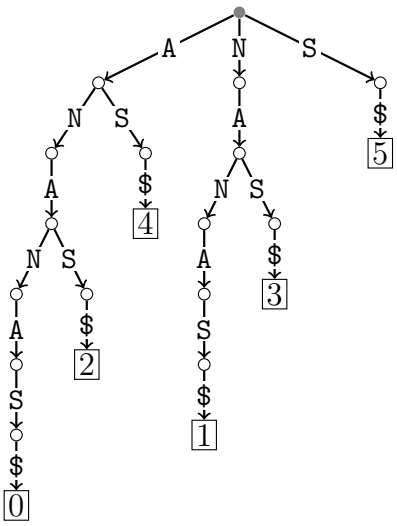
For the k-difference problem the optimal choice would be $t = k + 1$. As a consequence, if $dist(A,B) \leq k < t$, B must have a factor whose edit distance to some A_i is less than t_i.

8.5 How to Implement a Factor Filter

Given a pattern P, a factorization $P = P_0 P_1 \cdots P_{s-1}$ and weights $t_0 + t_1 + \ldots + t_{s-1} = k + 1$, we want to implement a factor filter using the *suffix trie*[1] of the text T.



suffix tree of T=ANANAS



suffix trie of T

8.6 Exact Factor Search

For the special case $t_0 = \ldots = t_{s-1} = 1$, each factor P_i can be searched exactly with a top-down traversal of the suffix trie along the path of characters $P_i[0], P_i[1], P_i[2] \cdots$.

If the factor occurs in the text, the search ends in a leaf or an inner suffix tree node. The leaf or the leaves beneath the node represent all occurrences of the factor in the text.

To verify whether an occurrence is part of a true match, we search P with up to k errors in the surroundings using a (more expensive) approximate search algorithm.

[1] The suffix trie results from the suffix tree after breaking all edges into paths of single character edges.

8.7 Approximate Factor Search

In the more general case, factors have to be searched with $t_i - 1$ errors in the suffix tree. This can be done with a backtracking approach.

Instead of traversing only matching edges, we traverse all outgoing edges of a suffix tree node and tolerate errors. Whenever we descend over a mismatching edge, we decrease a counter e that records the number of remaining tolerable errors.

Indels are simulated by skipping a character either in the pattern or in the suffix tree and decreasing e.

```
(1)  ApproxRecur(F, i, ᾱ, e);
(2)  // F..factor, i..compared prefix length
(3)  // ᾱ..suffix trie node with path label α
(4)  // e..remaining errors to tolerate
(5)  if e ≥ 0
(6)    then
(7)        if i = |F|
(8)          then
(9)                report occurrences at getOccurrences(ᾱ);
(10)       fi
(11)       ApproxRecur(F, i + 1, ᾱ, e − 1); // insertion in factor
(12)       for ᾱc ∈ children(ᾱ) do
(13)           ApproxRecur(F, i, ᾱc, e − 1); // deletion in factor
(14)           if F[i] = c
(15)             then
(16)                 ApproxRecur(F, i + 1, ᾱc, e); // match
(17)             else
(18)                 ApproxRecur(F, i + 1, ᾱc, e − 1); // mismatch
(19)           fi
(20)       od
(21) fi
```

8.8 Suffix Filters

Again we consider strings A and B and optimal factorizations $A_0A_1 \cdots A_{s-1}$ and $B_0B_1 \cdots B_{s-1}$ and weights t_i such that $dist(A, B) < \sum_{i \in [0..s)} t_i$.

For $0 \le i \le j < s$, let $A[i..j]$ denote the concatenation of factors $A_iA_{i+1} \cdots A_j$. In particular, $A[i..s-1]$ is a suffix of A. (We use the same notation for B).

Definition 6 (strong match)**.** We say that A and B *match* on interval $[i..j]$ if

$$dist(A[i..j], B[i..j]) \; < \sum_{h \in [i..j]} t_h \; ,$$

and *strongly match* on $[i..j]$ if they match on every interval $[i..j']$, $j' = i, \ldots j$, that means if they match on every nonempty prefix of $[i..j]$.

While the weighted pigeonhole lemma only states that there is a match $[i..i]$, the following more general theorem guarantees that there is a strong match $[i..s)$ for a certain $i \in [0..s)$.

Theorem 7. *If $dist(A, B) < t$, there exists $i \in [0..s)$ such that A and B strongly match on $[i..s)$.*

Proof: Let $[0..i)$ be the longest prefix interval, on which A and B do not match, i.e. $dist(A[0..i), B[0..i)) \ge \sum_{h \in [0..i)} t_h$. It is well defined as $i = 0$ always satisfies the condition. It holds $i < s$ as $[0..s)$ is always a match.

Then, A and B strongly match on $[i..s)$. To show this, assume the opposite, i.e. that there exists $j \in [i..s)$ such that A and B do not match on $[i..j]$. But then A and B do not match on $[0..j] = [0..i) \cup [i..j]$, which contradicts $[0..i)$ being maximal.

Example 8. In the situation of the following table, A and B strongly match on $[1..5)$ but not on any other suffix $[i..5)$:

i	0	1	2	3	4
t_i	1	1	2	1	1
$dist(A_i, B_i)$	1	0	1	2	1

,

as A and B match on $[1..j]$ for each $j \in [1..5)$:

I	[1..1]	[1..2]	[1..3]	[1..4]
$\sum_{i \in I} t_i$	1	3	4	5
$>$				
$\sum_{i \in I} dist(A_i, B_i)$	0	1	3	4

.

The filter algorithm is identical to factor filters except instead of searching for separate factors, the filtration phase will search for suffixes of the pattern satisfying the strong match condition. That is, it searches for each suffix $A[i..s)$ with less than $\sum_{j \in [i..s)} t_j$ errors.

To get an intuition on suffix filters, we want to compare the following (sub)problems of using the index to find the occurrences of:

1. pattern A with less than t errors
2. factor A_0 with less than t_0 errors
3. suffix $A[0..s)$ under the strong match condition

1. In the first case t errors need to be found to eliminate a candidate.
2. In the 2nd and 3rd case only t_0 in the first factor are sufficient.
3. While the factor filter produces a verification candidate after matching the first factor, a suffix filter continues the search and has more chances for elimination

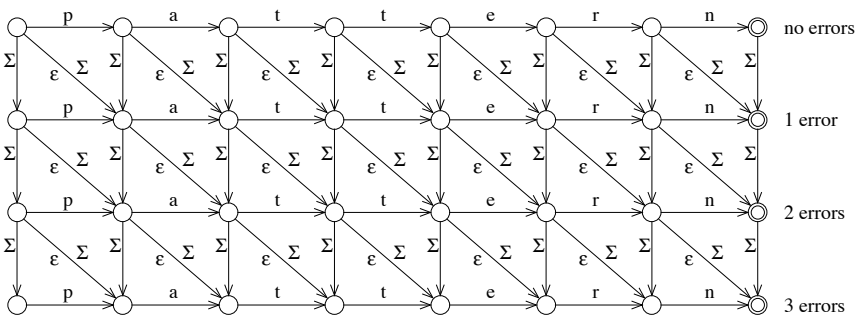
Hence a suffix filter is more specific then a factor filter and saves time due to less unsuccessful verifications.

8.9 Approximate Suffix Search

To search each suffix under the strong match condition we could extend the ApproxRecur algorithm to search a sequence of factors and allow t_{i+1} more errors (increase e) after successfully matching the factor A_i.

However, the backtracking of algorithm ApproxRecur visits the same suffix tree node multiple times. Kärkkäinen and Na use an approach that visits each nodes at most once and hence requires less backtracking steps. They construct an NFA of the suffix that accepts a suffix within a certain edit distance.

The NFA nodes are arranged in a grid, where the i-th row represents i errors and the j-th column a consumed prefix of length j (counting from 0). Matches and mismatches are transitions from column j to $j + 1$, where matches connect nodes of the same row and mismatches from row i to $i + 1$. Indels are empty diagonal transitions and any-character transitions.



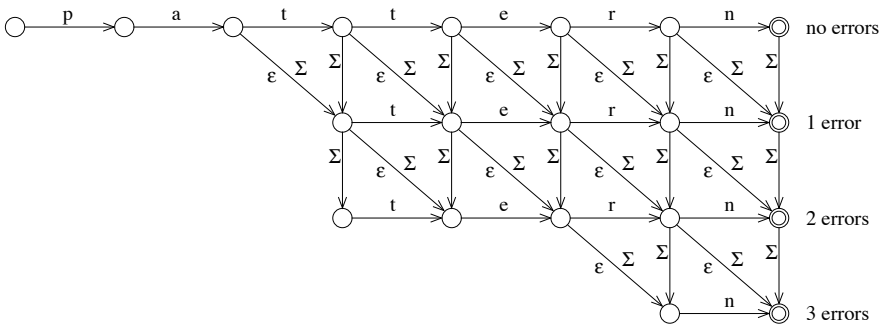
(a) An NFA for the pattern “pattern” with 3 errors.

Suffix filters require recognition of strong matches, which do not allow all errors to occur in the beginning. The corresponding NFA (also called *staircase NFA*) is obtained from the standard NFA by eliminating states that violate the strong match conditions. A suffix has been found if any of the (accepting) states in the last column become active.

Example 9. For example, consider the following factorization of the string `pattern`

factor	`pa`	`tte`	`rn`
t_i	1	2	1

Because the prefix `pa` allows only exact matches we eliminate all states with $i > 0$ and $j \leq 2$. Because the prefix `patee` allows only 2 errors we eliminate all states with $i > 2$ and $j \leq 5$.

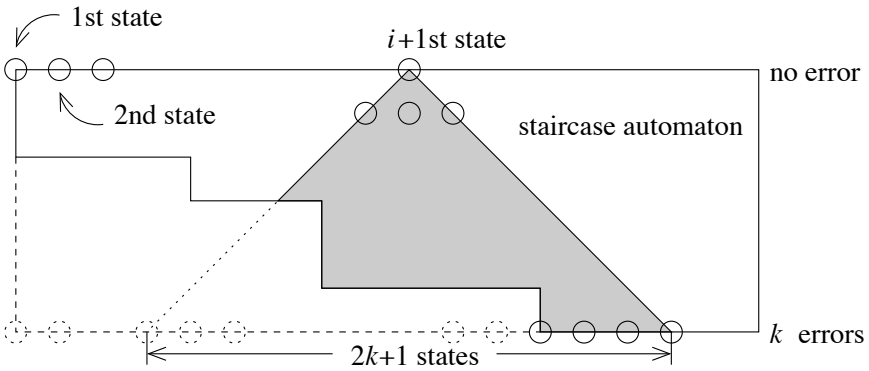


(b) A staircase NFA for suffix filters.

The suffix tree is searched with the staircase NFA via backtracking. A subtree can be skipped if the NFA has no more active states.

The simulation of the NFA can be sped up with the following heuristics:

- After reading i characters, only states within a triangle whose peak is the $(i + 1)$-st state of the first row can be active.
- If the top h rows have no active states, they will stay inactive also in the future, and can be omitted.



8.10 Verification

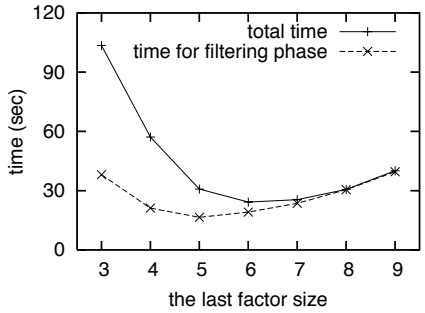
After a suffix was found with $e \leq k$ errors, we search the remaining pattern prefix left of the text occurrences with up to $k - e$ errors to verify whether the suffix is part of a true k-difference match of the pattern.

This can be done with any approximate pattern matching algorithm, e.g. Myers' bitvector algorithm (Myers, 1999) or the above mentioned NFA approach. Alternatively we could use backtracking in an index that allows a bidirectional search, e.g. the bidirectional BWT (Schnattinger et al., 2012).

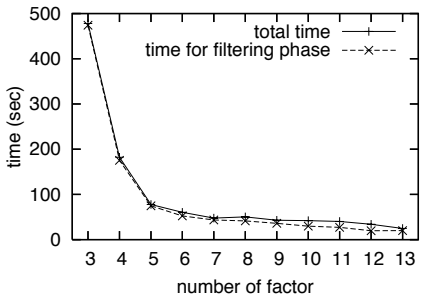
8.11 Suffix Filter Parameters

The practically best factorization divides a pattern of length m into $k+1$ factors of weight 1.

The length ℓ of the last factor should be greater then others to avoid random matches of the shortest suffix. All other factor sizes should be distributed evenly, i.e. either $\left\lfloor \frac{m-\ell}{k} \right\rfloor$ or $\left\lceil \frac{m-\ell}{k} \right\rceil$.



(a) varying the last factor size ℓ ($s = 13$)



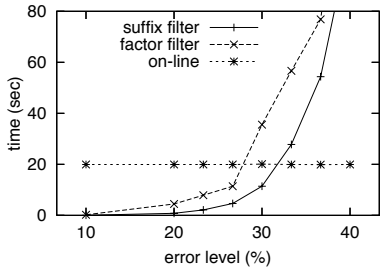
(b) varying the number of factors s ($\ell = 6$) [2]

8.12 Performance Results

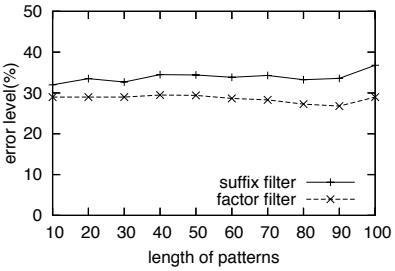
The real-world experiments were conducted on English texts and DNA data which were obtained from the Pizza&Chili Corpus[3] and truncated to length 16Mbytes.

In the first experiment, 100 queries of length $m = 30$ were searched varying the error level (left side).

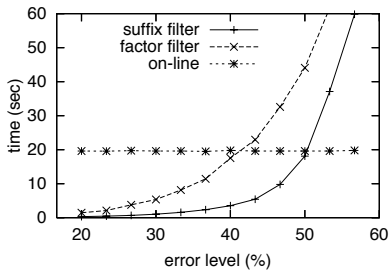
The second experiment determines for different pattern lengths m the error levels up to which each indexing method wins upon online search (right side).



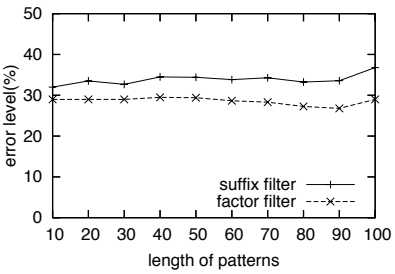
(a) DNA



(a) DNA



(b) English



(b) English

[2] Performance of suffix filter on DNA data for 200 queries of length $m = 40$ and $k = 12$

[3] `http://pizzachili.dcc.uchile.cl/`