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 than 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 $p_{i_1} \ldots p_{i_2}$ matches the text $t_j \ldots t_{j+i_2-i_1}$, then the verification has to consider the text area $t_{j-(i_1-1)-k} \ldots t_{j+(m-i_1)+k}$, 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( $p = p_i p_{i+1} \dots p_j$, $k$, $myParent$, $idx$, $plen$ )
(2)  // Note: the initial call is: CreateTree ( $p$, $k$, $nil$, 0, $\lfloor m/(k+1) \rfloor$ )
(3)  Create new node $node$
(4)  $from(node) = i$
(5)  $to(node) = j$
(6)  $left = \lceil (k+1)/2 \rceil$
(7)  $parent(node) = myParent$
(8)  $err(node) = k$
(9)  if $k = 0$
(10)   then $leaf_{idx} = node$
(11)   else
(12)       $lk = \lfloor (left \cdot k)/(k+1) \rfloor$
(13)       CreateTree( $p_i \dots p_{i+left \cdot plen - 1}$, $lk$, $node$, $idx$, $plen$ )
(14)       $rk = \lfloor ((k+1-left) \cdot k)/(k+1) \rfloor$
(15)       CreateTree( $p_{i+left \cdot plen} \dots p_j$, $rk$, $node$, $idx + left$, $plen$ )
(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 $(from, to, error)$.

```
                     "annual" n4=(1,6,2)
                     /                 \
           "annu" n3=(1,4,1)            \
            /         \                  \
  "an" n0=(1,2,0)   "nu" n1=(3,4,0)   "al" n2=(5,6,0)
      |                 |                 |
  leaf 0            leaf 1            leaf 2
```

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

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

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 $(pos, i) \in$ output of multi-pattern search do
(3)     $n = leaf_i$; $in = from(n)$; $n = parent(n)$;
(4)     $cand = true$;
(5)     while $cand = true$ and $n \neq nil$ do
(6)         $p_1 = pos - (in - from(n)) - err(n)$;
(7)         $p_2 = pos + (to(n) - in) + err(n)$;
(8)         verify text $t_{p_1} \dots t_{p_2}$ for pattern piece $p_{from(n)} \dots p_{to(n)}$
(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 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.