6 Fast String Matching

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

1. An earlier version of this chapter by Knut Reinert.
2. Flexible Pattern Matching in Strings, Navarro, Raffinot, 2002, pages 15ff.
3. A nice overview of string matching algorithms with implementations can be found under `http://www-igm.univ-mlv.fr/~lecroq/string/`

We will first present three practical algorithms for matching a single pattern in a given text. We will then look at the problem of multiple string matching, that is, matching a whole set of patterns in a given text.

6.1 Some string-matching basics

Exact string matching: The task is to find all occurrences of a given pattern $p = p_1, \ldots, p_m$ in a text $T = t_1, \ldots, t_n$, usually with $m << n$.

The algorithmic ideas of exact string matching are useful to know, although in computational biology algorithms for *approximate* string matching, or *indexed* methods are of more use.

String matching can be addressed by approaches that range from the extremely theoretical to the extremely practical.

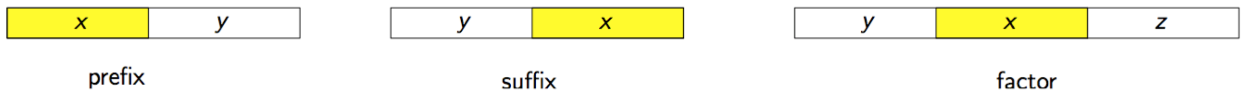
One example is the famous *Knuth-Morris-Pratt* (1977) algorithm that has $O(n)$ worst-case time complexity, but is in practice twice as slow as the brute force algorithm, and the well-known *Boyer-Moore* (1977) algorithm which has a worst-case running time of $O(mn)$, but is quite fast in practice.

The search is done in a window that slides along the text. The search window has the same width as the pattern.

Example:
Text: `CPM_annual_conference_announcement`
Pattern: `announce`

```
                 search
             ←   window   →
C P M _ a n  n u a l _ c o n  f e r e n c e _ a n n o u n c e m e n t
             a n n o u n c e
```

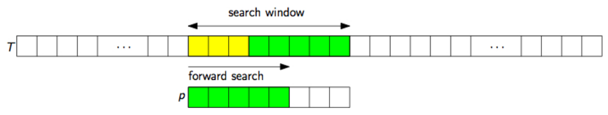
Some easy terminology: Given strings x, y, and z, we say that x is a *prefix* of xy, a *suffix* of yx, and a *factor* (substring) of yxz.



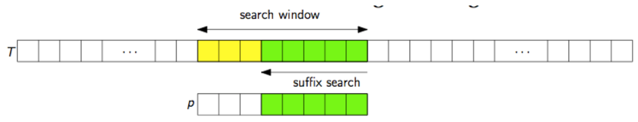
There are three basic approaches to string matching. In each a *search window* of the size of the pattern is moved from left to right along the text and the pattern is searched within the window.

They are:

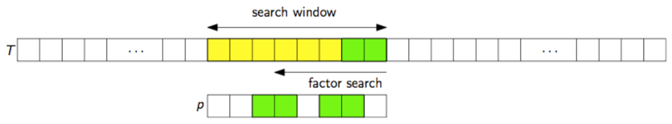
1. *Prefix searching.* For each position in the window we seek the longest suffix of the search window that is also a prefix of the pattern.



2. *Suffix searching.* The search is conducted backwards along the search window, matching a suffix in the search window to a suffix of the pattern. By avoiding to read some characters this can lead to sub-linear average case algorithms.



3. *Factor searching.* The search is done backwards in the search window, looking for the longest suffix in the window that is also a factor of the pattern.



6.2 Prefix-based approaches

In prefix-based approaches, we move a window of length m along the text. We consider the prefixes of the pattern p that are suffixes of the portion of the text contained in the window.

The Knuth-Morris-Pratt algorithm computes the longest suffix of the text in the window that is a prefix of the pattern p. We will not consider this algorithm further because it is inferior in practice.

The algorithms discussed here maintain the set of all prefixes of the pattern p that are suffixes of the portion of the text contained in the window and updates this set when moving the window to the next position in the text.

We will discuss the *Shift-And* and the *Shift-Or* algorithm. Both maintain the set of all prefixes of p that match a suffix of the text upto the current position i.

The algorithms use bit-parallelism to update this set for each new text character. The set is represented by a bit mask $D = d_m, \ldots, d_1$.

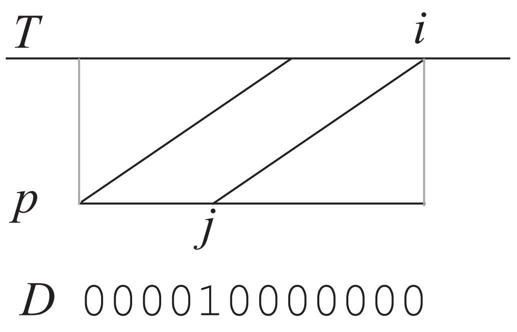
We first describe the Shift-And algorithm and then show how to obtain the Shift-Or algorithm.

6.2.1 Shift-And and Shift-Or

We maintain the following *invariant*:

There is a 1 at the j-th position of D (position j is "active")
if and only if
$p_1, \ldots, p_j$ is a suffix of $t_1, \ldots, t_i$.

Illustration:



When reading the next character t_{i+1}, we have to compute the new set D'. We use the following:

Observation: A position $j+1$ in the set D' will be active if and only if the position j was active in D and t_{i+1} matches p_{j+1}.

The algorithm uses a table B that contains a bit mask for each character $c \in \Sigma$, namely $B[c] = b_m, \ldots, b_1$, where the j-bit is set if $p_j = c$.

Algorithm 6.2.1 (Shift-And)

```
Input: text $T$ and pattern $p$
Output: all occurrences of $p$ in $T$
Init.:
for $c \in \Sigma$ do
    Set $B[c] = 0^m$
for $j = 1 \ldots m$ do
    Set $B[p_j] = B[p_j] \mid 0^{m-j}10^{j-1}$
Main loop:
Set $D = 0^m$
for $pos = 1 \ldots n$ do
    Set $D = ((D << 1) \mid 0^{m-1}1)\ \&\ B[t_{pos}]$        $(*)$
    if $D\ \&\ 1\,0^{m-1} \neq 0^m$ then
      report an occurrence at position $pos - m + 1$
end
```

Initially we set $D = 0^m$ and for each new character t_{i+1} we update D using the formula $D' = ((D << 1) \mid 0^{m-1}1)\ \&\ B[t_{pos}]$. This update maintains our invariant using the above observation.

The shift operation marks all positions as potential prefix-suffix matches that were such matches in step i (notice that this includes the empty string ϵ). In addition, to stay a match, the character $B[t_{i+1}]$ has to match p at those positions. This is achieved by applying an & with the appropriate bitmask from the table B.

What is the Shift-Or algorithm? It is just an implementation trick to avoid one bit operation, namely the $\mid 0^{m-1}1$ in the line $(*)$.

In the Shift-Or algorithm we complement all bit masks of B and use a complemented bit mask D. Now the $<<$ operator will introduce a 0 to the right of D' and the new suffix stemming from the empty string is already in D'. Obviously, we have to use a bit $\mid$ instead of an & and report a match whenever $d_m = 0$.

Lets look at an example of Shift-And and Shift-Or.

6.2.2 Example

Find all occurrences of `p=ATAT` in the text `T=ATACGATATATA`.

```
          Shift-And          Shift-Or
            A|0101             A|1010
       B =  T|1010      B   = T|0101
            *|0000             *|1111

          D = 0000          D = 1111

1 Reading A
           0001               1110
           0101               1010
           ----               ----
           0001               1110

2 Reading T          3 Reading A
0011     1100         0101     1010
1010     0101         0101     1010
-------------         -------------
0010     1101         0101     1010

4 Reading C           5 Reading G
1011     0100         0001     1110
0000     1111         0000     1111
-------------         -------------
0000     1111         0000     1111

6 Reading A           7 Reading T
0001     1110         0010     1100
0101     1010         1010     0101
-------------         -------------
0001     1110         0010     1101

8 Reading A           9 Reading T
0101     1010         1011     0100
0101     1010         1010     0101
-------------         -------------
0101     1010         1010     0101
```

Hence, in step 9, we found the first occurrence of p at position $9 - 4 + 1 = 6$.

6.3 Suffix-based approaches

As noted above, in the suffix-based approaches we match the characters from the back of the search window. Whenever we find a mismatch we can perform a "safe shift" of the window that does not miss any occurrence of the pattern in the text.

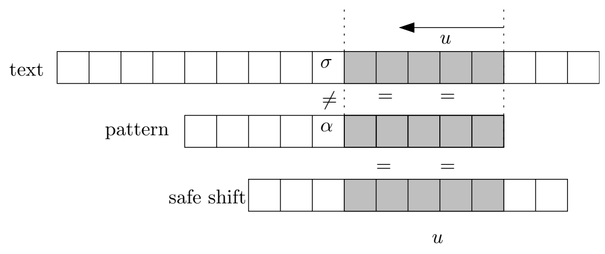
We present the idea of the Boyer-Moore algorithm and then provide code for the *Horspool* simplification which is often faster.

6.3.1 The Boyer-Moore algorithm

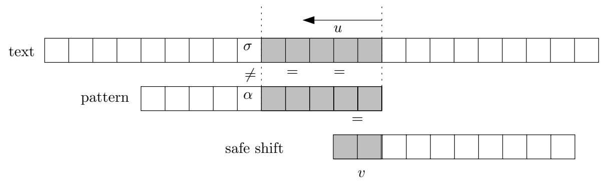
The Boyer-Moore algorithm precomputes three shift functions d_1, d_2 and d_3 that correspond to three different cases.

In each case we assume that we have just read a suffix u of the search window that is also a suffix of p, and we have failed on a text character σ that does not match the next character α.

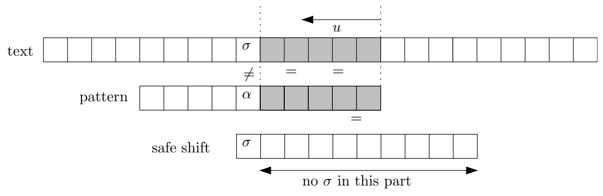
First case: The suffix u occurs in another position as a factor of p. Then a safe shift is to move the window such that it matches the nearest occurrence. We precompute for each suffix of the pattern the distance to its next occurrence backwards in the pattern. We call this function d_1.



Second case: The suffix u does not occur anywhere as a factor of p. This does not mean we can safely skip the whole search window (see picture below), because a suffix v of u can also be a prefix of the pattern. To deal with this we compute a function d_2 for all suffixes of the pattern which returns the length of the longest prefix v of p that is also a suffix of u.



Third case: The backward search has failed on character σ. If we shift the window with the first function d_1 and this letter is not aligned with a σ in the pattern, then we will perform an unnecessary verification of the new search window. A third function d_3 is computed to ensure that the text character σ corresponds to a σ in the pattern. It contains for each character in the alphabet its rightmost occurrence to the end of the pattern. If σ is not in p it returns m.



Based on the analysis of the three cases, the algorithm determines the shift to apply to the search window as follows:

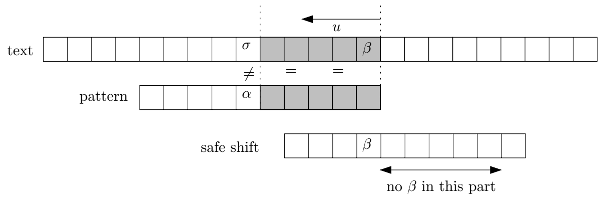
- the maximum of $d_1(u)$ and $d_3(\sigma)$ since we want to align u with its next occurrence in the pattern, knowing that the σ of the text has to match another σ in the pattern.
- the minimum of the above and $m - d_2(u)$, since the latter expression is the maximum safe shift that can be performed.

If the beginning of the window has been reached, then we have found an occurrence and only d_2 is used to shift the search window.

6.3.2 The Horspool algorithm

Horspool simplified the BM algorithm avoiding the complicated computation of d_1, d_2 and d_3. The idea is that d_3 will usually provide the longest shift, for a reasonably sized alphabet. A small modification also makes d_3 easy to compute and produces even longer shifts.

For each position of the search window we compare the last character β with the last character of the pattern. If they match we verify until we find the pattern or fail on the text character σ. Then we simply shift the window according to the next occurrence of β in the pattern.



Algorithm 6.3.1 (Horspool)

```
*Input: text $T$ and pattern $p$*
*Output: all occurrences of $p$ in $T$*
*Init.:*
for $c \in \Sigma$ do
        Set $d[c] = m$
for $j = 1 \ldots m-1$ do
        Set $d[p_j] = m - j$
*Main loop:*
Set $pos = 0$
while $pos \leq n - m$ do
        Set $j = m$
        while $j > 0$ and $t_{pos+j} = p_j$ do
                $j--$
        if $j=0$ then
                report an occurrence at $pos + 1$
        Set $pos = pos + d[t_{pos+m}]$
end
```

6.3.3 Horspool example

We search for the string `announce` in the text `CPM_annual_conference_announcement`.

```
m=8,  d = a c n o u *
          7 1 2 4 3 8

1)  < CPM_annu > al_conference_announcement
   u != e, d[u]=3

2) CPM < _annual_ > conference_announcement
   _ != e, d[_] = 8

3) CPM_annual_ < conferen > ce_announcement
   n != e, d[n] = 2



4)  CPM_annual_co < nference > _announcement
   e == e => verify  until it fails with e != u => d[e] = 8

5) CPM_annual_conference < _announc > ement
   c != e, d[c]=1
```

```
6) CPM_annual_conference_ < announce > ment
   e == e => verify  until occurrence is found

7)
```

6.4 Summary

- There are three basic approaches for string matching: prefix searching, suffix searching, and factor searching.
- Although theoretical optimal the KMP algorithm is outperformed by almost all simple algorithms even if they do not possess an optimal time complexity.
- Bit parallelism can speed up simple algorithms even more.
- Both suffix-based and factor based search compare a suffix of the pattern with a suffix of the search window. In factor search we compare the text with all factors of the patterns whereas in suffix search we compare only with a suffix of the pattern.

6.5 Multiple String Matching

In multiple string matching we are given a *text* $T = t_1 t_2 \dots t_n$ and want to search simultaneously for a set of strings

$$P = \{p^1, p^2, \dots, p^r\},$$

where $p^i = p^i_1 p^i_2 \dots p^i_{m_i}$ is a string of length m_i, for $i = 1, \dots, r$.

Define $|P| = \sum_{i=1}^r |p^i| = \sum_{i=1}^r m_i$ and let $lmin$ and $lmax$ denote the minimum and maximum length of any pattern in P, respectively.

Strings in P may be prefixes, suffixes, infixes or even the same as others.

E.g., if we search for $P = \{\texttt{ATATA}, \texttt{ATAT}\}$ in a DNA sequence, then each occurrence of `ATATA` will also imply an occurrence of `ATAT`.

The simplest solution to this problem is to repeat a single-pattern matching algorithm, such as Shift-Or, r times.

Under the assumption that a pattern fits into a few computer words, the worst case time complexity of this approach is $O(|P|)$ for preprocessing and $O(n \times r)$ for searching.

We will discuss how to extend the single-pattern matching algorithms to obtain an improved run-time complexity.

Here are two examples of multiple-pattern matching problems:

English:

Text: `CPM_annual_conference_announce`

Set of patterns: $\left\{ \begin{array}{l} \texttt{announce} \\ \texttt{annual} \\ \texttt{annually} \end{array} \right.$

DNA:

Text: `AGATACGATATATAC`

$$\text{Set of patterns: } \begin{cases} \texttt{ATATATA} \\ \texttt{TATAT} \\ \texttt{ACGATAT} \end{cases}$$

As for single-pattern matching, there are three types of approaches:

- **Prefix searching** Search forward, reading characters of the text with an automaton built on the set P.
- **Suffix searching** A position *pos* is slid along the text, from which we search backward for a suffix of any of the strings. We shift *pos* according to the next occurrence of the suffix read in P. We may be able to avoid reading all the characters of the text.
- **Factor searching** A position *pos* is slid along the text, from which we search backward for a prefix of size *lmin* of the strings in P. Again, we may be able to avoid reading all the characters of the text.

6.6 Multiple Shift-And algorithm

Extension of prefix-based approaches to single-pattern matching lead to the *Aho-Corasick* and the *Multiple Shift-And* algorithms. The former is an extension of the KMP algorithm. We will discuss the latter in detail.

The Multiple Shift-And algorithm is only useful when the set $P = \{p^1, \ldots, p^r\}$ fits into a few machine words. For simplicity, we will assume that $|P| \leq w$ holds, where w is the word size of the computer.

The idea is to use bit-parallelism to perform all computations required for the r strings in the same computer word.

	p_4	p_3	p_2	p_1
	------------	-------------	----------	-----------
computer word	$\leftarrow m_4 \rightarrow$	$\leftarrow m_3 \rightarrow$	$\leftarrow m_2 \rightarrow$	$\leftarrow m_1 \rightarrow$

We pack the patterns together so that the positions of the different characters correspond to positions in a computer word, as indicated above.

We use the Shift-And algorithm as defined for single-pattern matching, but with two changes:

The initialization word DI is the concatenation of the initialization words for each string:

$$DI \leftarrow 0^{m_r-1}1 \ldots 0^{m_2-1}10^{m_1-1}1,$$

whereas the word used in the final test (whether to print out a match) is

$$DF \leftarrow 10^{m_r-1} \ldots 10^{m_2-1}10^{m_1-1}.$$

For example, consider the following set of patterns:

$$P = \left\{ \begin{array}{ll} p^1 & = \texttt{ATG} \\ p^2 & = \texttt{CCAT} \\ p^3 & = \texttt{AGAT} \end{array} \right\}.$$

We set:

$$\begin{array}{lclll} & & p^3 & p^2 & p^1 \\ p & = & \texttt{TAGA} & \texttt{CGAT} & \texttt{GTA} \\ & & & & \\ DI & = & \texttt{0001} & \texttt{0001} & \texttt{001} \\ & & & & \\ DF & = & \texttt{1000} & \texttt{1000} & \texttt{100} \end{array}$$

Algorithm 6.6.1 (Multiple Shift-And)

```
Input: Set of patterns $P = \{p^1, \dots, p^r\}$, text $T$
Output: All occurrences of any pattern in $T$
Init.:
for $c \in \Sigma$ do
    Set $B[c] = 0^{|P|}$
Set $l = 0$
for $k = 1 \dots r$ do
    for $j = 1 \dots m_k$ do
        Set $B[p^k_j] = B[p^k_j] \mid 0^{|P|-l-j}10^{l+j-1}$
    Set $l = l + m_k$
Main loop:
Set $D = 0^{|P|}$
for $pos = 1 \dots n$ do
    Set $D = ((D << 1) \mid DI)\ \&\ B[t_{pos}]$
    if $D\ \&\ DF \neq 0^{|P|}$ then
        Check which patterns match and report occurrences
end
```

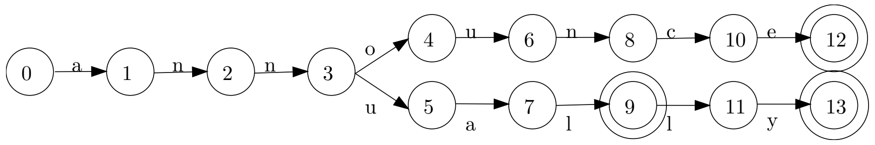
6.7 The trie data-structure

Given a set of patterns $P = \{p^1, p^2, \dots, p^r\}$, a useful representation of P is a *trie*, which is is used by most classical multi-string matching algorithms.

Definition 6.7.1 (Trie) *The* trie *associated with a set of patterns P is a rooted directed tree such that:*

- *Every path starting from the root ρ is labeled by one of the strings p^i.*
- *Vice-versa, every string $p^i \in P$ labels a path from the root.*
- *Every node q (or* state*) corresponding to an entire string is called* terminal *and a function $F(q)$ points to a list of all strings in P that correspond to q.*

For example, the trie for {`announce`, `annual`, `annually`} is:



Algorithm 6.7.2 (Trie)

```
Input: Set of patterns $P = \{p^1, \ldots, p^r\}$
Output: trie for $P$
Init.: Create an initial non-terminal state root
Main loop:
for $i = 1 \ldots r$ do
    Set $cur = root$ and $j = 1$
    while $j \leq m_i$ and $\delta(cur, p^i_j) \neq null$ do
        Set $cur = \delta(cur, p^i_j)$ and $j{+}{+}$
    while $j \leq m_i$ do
        Create a new state called "state"
        Set $\delta(cur, p^i_j) = state$, $cur = state$ and $j{+}{+}$
    if $cur$ is terminal then
        Set $F(cur) = F(cur) \cup \{i\}$
    else
        Mark $cur$ as terminal
        Set $F(cur) = \{i\}$
end
```

The size of the trie and the running time of its basic operations depend on the implementation of the *transition function* $\delta(v, c)$, which maps each node v onto a next node depending on the character c. The fastest implementation uses for each node a table of size $|\Sigma|$, where $|\Sigma|$ is the size of the alphabet.

If $|\Sigma|$ is too large or space is limited, one can bound the total space used to code for the transitions by $O(|P|)$. This leads however from the $O(1)$ time table access to a running time of $O(|\Sigma|)$ or $O(\log |\Sigma|)$.

6.8 The Set Horspool algorithm

The Horspool algorithm for single-pattern matching can be extended to provide a suffix-based approach to multiple-pattern matching. The resulting algorithm is called the *Set Horspool* algorithm.

The algorithms starts reading the text backwards from position *pos* that is initialized to *lmin*. The characters are compared with the trie built on the set $\bar{P}$ of reverse patterns.

If a terminal state is reached, then an occurrence is reported.

If the character read does not match the trie, then we shift the position *pos* using the first character read β. We shift until β is aligned with another β in the trie. If such a β does not exist, simply shift by *lmin* characters.

6.9 Summary

- We saw how to generalize the Shift-And and Horspool algorithms to address the problem of multiple-string matching.
- Bit-vector-based approaches are not as successful as in single string matching since the overall length of the pattern is too long.
- Suffix-based approaches (Wu-Manber) and factor-based approaches are the most successful in practice (but where not discussed here).
- In a later chapter we will look at the use of *suffix trees* and *suffix arrays* for matching of multiple patterns.