Stoye's diploma thesis on affix trees (the English translation appeared in [300]), and Maaß [208] showed that affix trees can be constructed on-line in linear time. Basically, the affix tree of a string S comprises both the suffix tree of S (supporting forward search) and the suffix tree of the reverse string S^{rev} (supporting backward search). Strothmann [302] showed that affix arrays have the same functionality as affix trees, but they require less than half the space. An affix array combines the suffix arrays of S and S^{rev}, but it is a complex data structure because the interplay between the two suffix arrays is rather difficult to implement. A reimplementation of affix arrays is described in [221].

7.9 Approximate string matching

Approximate string matching is the technique of finding substrings of a long string S (or a collection of strings) that match a pattern P approximately (rather than exactly). Approximate search algorithms are abundant and there is a vast literature on the topic. We shall not discuss this field in detail, but instead refer to the overview article [236]. Here, we consider solely the case in which S is fixed and many on-line queries of the form "Where are all approximate matches of P in S?" must be answered efficiently. A prime example in bioinformatics is short read mapping. High-throughput sequencing (or next-generation sequencing) technologies produce billions of bases in a single run. In their short read mapping primer [312], Trapnell and Salzberg write:

> One of the challenges presented by the new sequencing technology is the so-called 'read mapping' problem. Sequencing machines made by Illumina of San Diego, Applied Biosystems (ABI) of Carlsbad, California, and Helicos of Cambridge, Massachusetts, produce short sequences of 25–100 base pairs (bp), called 'reads', which are sequence fragments read from a longer DNA molecule present in the sample that is fed into the machine. In contrast to whole-genome assembly, in which these reads are assembled together to reconstruct a previously unknown genome, many of the next-generation sequencing projects begin with a known, or so-called 'reference', genome. In this case, to make sense of the reads, their positions within the reference sequence must be determined. This process is known as aligning or 'mapping' the read to the reference.

Short-read mappers are, among others, Bowtie [198], BWA [202], SOAP2 [203], and 2BWT [197]; see e.g. [113, 312] for overview articles. In the following, we discuss the basic algorithms used in BWA, Bowtie, and 2BWT.

The short read mapping problem is exacerbated by sequencing errors and variations between the sequenced chromosomes and the reference genome.[9] First, we consider the scenario in which only mismatches are allowed (Hamming distance) and subsequently address the problem of also allowing insertions and deletions (edit distance).

7.9.1 Using backward search

Definition 7.9.1 The *Hamming distance* between two strings S^1 and S^2 of equal length is the number of positions at which the corresponding characters are different:

$$hdist(S^1, S^2) = |\{i \mid S^1[i] \neq S^2[i]\}|$$

To put it another way, the Hamming distance measures the minimum number of substitutions required to change S^1 into S^2 (or vice versa).

Definition 7.9.2 Let P and S be strings with $m = |P| < |S| = n$, and let k be a natural number with $k < m$. An m-length substring $S[i..i+m-1]$ is called a *k-mismatch* of P in S if $hdist(P, S[i..i+m-1]) \leq k$. The *$k$-mismatch problem* is to find all positions in S at which a k-mismatch of P in S starts.

When k and Σ are small, one can solve the k-mismatch problem by the following approach. First, generate the so-called Hamming sphere $\mathcal{P}$ of radius k at center P (defined below). Second, use the Aho-Corasick algorithm from Section 2.5 to find all positions in S at which a pattern from $\mathcal{P}$ starts. The Hamming sphere of radius k at center P is the set

$$\mathcal{P} = \{P' \mid hdist(P, P') \leq k\}$$

The number of strings in the Hamming sphere $\mathcal{P}$ is

$$\sum_{i=0}^{k} \binom{m}{i} (|\Sigma| - 1)^i \in O(m^k |\Sigma|^k)$$

As an example, consider the pattern $P = tact$ on the alphabet $\Sigma = \{a, c, g, t\}$. The Hamming sphere of radius $k = 1$ at center P is the set

$$\{tact, aact, cact, gact, tcct, tgct, ttct, taat, tagt, tatt, taca, tacc, tacg\}$$

Li and Durbin [202] suggested a different solution to the problem. Their algorithm uses an FM-index to simultaneously find different occurrences of subpatterns, and it prunes the search space using a lower bound on the distance.

[9] Li and Durbin [202] suggest the following number k of differences (mismatches or gaps) that should be tolerated: for 15-37 bp reads, k equals 2; for 38-63 bp, $k = 3$; for 64-92 bp, $k = 4$; for 93-123 bp, $k = 5$; and for 124-156 bp reads, $k = 6$.

Algorithm 7.40 The procedure *k-mismatch*.

```
procedure k-mismatch(P, j, d, [lb..rb])
  if d < 0 then
    return ∅
  if j = 0 then        /* k-mismatches detected */
    return {[lb..rb]}
  I ← ∅
  list ← getIntervals([lb..rb])
  for each (c, [lb..rb]) in list do
    if P[j] = c then
      I ← I ∪ k-mismatch(P, j − 1, d, [lb..rb])
    else        /* substitution of P[j] with c */
      I ← I ∪ k-mismatch(P, j − 1, d − 1, [lb..rb])
  return I
```

Algorithm 7.40 implements the approach, but it does not prune the search space. The procedure call $k\text{-}mismatch(P, m, k, [1..n])$ returns all k-mismatches of P in S. Let us illustrate the algorithm for $k = 1$. For each position j in P, it uses backward search to find the $P[j+1..m]$-interval, generates all $bP[j+1..m]$-intervals (where b can be any character except \$), and for each such interval it continues the backward search to find the $P[1..j-1]bP[j+1..m]$-interval. As an example, consider the pattern $P = tact$ and the full-text index of the string $S = ctaataatg\$$ shown on the left-hand side of Figure 7.38. For the position $j = 4$, the algorithm generates the b-interval of every character $b \in \{a, c, g, t\}$, and with each interval it continues the backward search to find the $tacb$-interval. In other words, it searches for $taca$, $tacc$, $tacg$, and $tact$. Only the recursive search for $tact$ still allows for one mismatch; in the other cases the algorithm searches for the exact phrases $taca$, $tacc$, and $tacg$ (in all three cases, the backward search stops after one step because neither ca nor cc nor cg are substrings of S). In case $j = 3$, the algorithm generates the at-interval $[4..5]$ and the ct-interval $[6..6]$. The latter results in the recursive call $k\text{-}mismatch(tact, 2, 1, [6..6])$; since $getIntervals([6..6])$ returns an empty list, the search stops here. The former results in the recursive call $k\text{-}mismatch(tact, 2, 0, [4..5])$, which leads to the output $\mathcal{I} = \{[8..9]\}$. This means that there are two 1-mismatches of P in S, namely starting at the positions $\mathsf{SA}[8] = 2$ and $\mathsf{SA}[9] = 5$.

Algorithm 7.40 can be extended in such a way that it can deal with insertions and deletions. To be precise, the modified algorithm solves the k-differences problem, which we formally define below.

i	SA	BWT	$S_{\mathsf{SA}[i]}$
1	10	g	\$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	\$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	BWT^{rev}	$S^{rev}_{\mathsf{SA}^{rev}[i]}$
1	c	\$
2	t	$aataatc\$$
3	t	$aatc\$$
4	a	$ataatc\$$
5	a	$atc\$$
6	t	$c\$$
7	\$	$gtaataatc\$$
8	g	$taataatc\$$
9	a	$taatc\$$
10	a	$tc\$$

Figure 7.38: Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$ (the backward index). Right-hand side: Burrows-Wheeler transform BWT^{rev} of $S^{rev} = gtaataatc\$$ (the forward index).

Definition 7.9.3 Given $S^1, S^2 \in \Sigma^*$, we write $S^1 \to S^2$ if

- S^2 can be obtained from S^1 by replacing one occurrence of $x \in \Sigma$ by $y \in \Sigma$, i.e., $S^1 = uxv$ and $S^2 = uyv$ (substitution),
- S^2 can be obtained from S^1 by inserting one occurrence of $y \in \Sigma$, i.e., $S^1 = uv$ and $S^2 = uyv$ (insertion),
- S^2 can be obtained from S^1 by deleting one occurrence of $x \in \Sigma$, i.e., $S^1 = uxv$ and $S^2 = uv$ (deletion).

In what follows, the term *indel* is used to mean an insertion or a deletion; substitutions and indels are collectively referred to as *edit operations*.

Furthermore, we write $S^1 \to^k S^2$ if S^1 can be transformed into S^2 by a sequence of $k \in \mathbb{N}$ edit operations.

Definition 7.9.4 The *edit distance* (or *Levenshtein distance*) between two strings S^1 and S^2 is the minimum number of edit operations needed to transform S^1 into S^2. Formally,

$$edist(S^1, S^2) = \min\{k \mid S^1 \to^k S^2\}$$

Definition 7.9.5 Let P and S be strings with $m = |P| < |S| = n$, and let k be a natural number with $k < m$. A substring $S[i..j]$ is called an

Algorithm 7.41 The procedure call $k\text{-}differences(P, m, k, [1..n])$ finds all approximate occurrencees of P in S, using the array M_{lr}.

```
procedure k-differences(P, j, d, [lb..rb])
  if d < M_lr[j] then /* M_lr[j] is a lower bound on the remaining differences */
    return ∅
  if j = 0 then /* approximate occurrencees of P in S detected */
    return {[lb..rb]}
  I ← ∅
  I ← I ∪ k-differences(P, j − 1, d − 1, [lb..rb]) /* deletion of P[j] */
  list ← getIntervals([lb..rb])
  for each (c, [lb..rb]) in list do
    I ← I ∪ k-differences(P, j, d − 1, [lb..rb]) /* insertion of c */
    if P[j] = c then
      I ← I ∪ k-differences(P, j − 1, d, [lb..rb])
    else /* substitution of P[j] with c */
      I ← I ∪ k-differences(P, j − 1, d − 1, [lb..rb])
  return I
```

approximate occurrence of P in S if $edist(P, S[i..j]) \leq k$. The *k-differences problem* is to find all positions in S at which an approximate occurrence of P in S starts.

Algorithm 7.41 solves the k-differences problem. In the first if-then statement, it uses a lower bound on the edit distance that can be used to prune the search space. To derive the lower bound, we use the following definition, which is motivated by Ehrenfeucht and Haussler's [84] notion of *compatible markings*. As a side remark: Ukkonen [314] as well as Chang and Lawler [54] used this technique in fast approximate string matching algorithms.

Definition 7.9.6 For a string S of length n and a pattern P of length m, there is a unique *left-to-right partition* $P = w_1c_1w_2c_2 \ldots w_kc_kw_{k+1}$ of P w.r.t. S so that each w_i is a substring of S but w_ic_i is not. The characters $c_1, \ldots, c_k$ are the *marked characters* and the *left-to-right marking*

$$M_{lr}(P, S) = \{p_i \mid p_i = \sum_{j=1}^{i} |\omega_j c_j|\}$$

is the set of positions at which the marked characters appear in P.

As an example, consider the pattern $P = ttaatt$ and the string $S = ctaataatg\$$; see Figure 7.38 (page 377). The left-to-right partition consists of $w_1 = t$, $c_1 = t$, $w_2 = aat$, $c_2 = t$, and $w_3 = \varepsilon$. Therefore, $M_{lr}(P, S) = \{2, 6\}$.

Lemma 7.9.7 *If $|M_{lr}(P,S)| = k$, then no substring of S matches P with less than k differences.*

Proof Let $P = w_1c_1w_2c_2 \ldots w_kc_kw_{k+1}$ be the left-to-right partition of P w.r.t. S. We show by finite induction on d, $1 \leq d \leq k$, that no substring of S matches the prefix $w_1c_1 \ldots w_dc_d$ of P with less than d differences. In the base case $d = 1$, we know that the longest substring of S that matches a prefix of P is w_1. Thus, no substring of S exactly matches the prefix w_1c_1 of P. In the inductive step, consider d with $2 \leq d \leq k$. According to the inductive hypothesis, no substring of S matches the prefix $w_1c_1 \ldots w_{d-1}c_{d-1}$ of P with less than $d-1$ differences. The longest substring of S that matches a prefix of $w_dc_d \ldots w_kc_kw_{k+1}$ has length $|w_d|$. Therefore, a substring of S may match $w_1c_1 \ldots w_{d-1}c_{d-1}w_d$ with $d-1$ differences, but no substring of S can match $w_1c_1 \ldots w_{d-1}c_{d-1}w_dc_d$ with $d-1$ differences (an exact match of a substring of S with a suffix of P that starts at or before position p_{d-1} must end before position p_d in P). This proves the lemma. □

Definition 7.9.8 Given S and P, let the array M_{lr} of size m be defined by

$$M_{lr}[j] = |\{p_i \leq j \mid p_i \in M_{lr}(P,S)\}|$$

for all j with $1 \leq j \leq m$.

Continuing our example from above, we have $M_{lr} = [0,1,1,1,1,2]$.

Corollary 7.9.9 *If $M_{lr}[j] = d$, then no substring of S matches $P[1..j]$ with less than d differences.*

Proof Since $M_{lr}[j] = d$, the left-to-right partition $P = w_1c_1 \ldots w_kc_kw_{k+1}$ of P w.r.t. S restricted to the first j characters is $P[1..j] = w_1c_1 \ldots w_dc_dw$, where w is a prefix of w_{d+1}. It is readily verified that the left-to-right partition of $P[1..j]$ w.r.t. S coincides with $w_1c_1 \ldots w_dc_dw$. Thus, the corollary immediately follows from Lemma 7.9.7. □

According to the preceding corollary, the backward search in Algorithm 7.41 can be stopped when $M_{lr}[j]$ is larger than the number of tolerated mismatches. This criterion effectively prunes the search space without sacrificing the correctness of the algorithm. Of course, we still have to find a way to compute M_{lr} efficiently. Li and Durbin [202] used the forward index for this purpose; see Exercise 7.9.10. An alternative is to use matching statistics, which can be computed space efficiently with the balanced parentheses sequence of the LCP-array; see Exercise 7.9.11.

Exercise 7.9.10 Give pseudo-code of an algorithm that computes the M_{lr}-array based on BWT^{rev}, the Burrows-Wheeler transform of S^{rev}. Analyze the run-time of the algorithm.

Exercise 7.9.11 Prove that Algorithm 7.42 correctly calculates the array M_{lr} and analyze its worst-case time complexity.

Algorithm 7.42 The procedure $calcM_{lr}(P)$ computes the M_{lr}-array based on the BWT of S.

```
compute the matching statistics ms of P w.r.t. S by Alg. 7.25 (page 337)
m ← |P|
k ← 0
j ← 1
flag ← true
while j ≤ m do
  if flag = true then
    for i = j to j + ms[j] − 1 do
      M_lr[i] ← k
    j ← j + ms[j]
    flag ← false
  else
    k ← k + 1
    M_lr[j] ← k
    j ← j + 1
    flag ← true
```

7.9.2 Using bidirectional search

During the development of the tool Bowtie, Langmead et al. [198] observed that a similar approach as in Algorithm 7.40 suffered from excessive backtracking. They write:

> Backtracking scenarios play out within the context of a stack structure that grows when a new substitution is introduced and shrinks when the aligner rejects all candidate alignments for the substitutions currently on the stack.

At first glance, there is no stack in Algorithm 7.40, but it is implicitly there: when the procedure is called, the program's runtime environment keeps track of the various instances of the procedure using a call stack. Bowtie mitigates excessive backtracking using two indexes that support backward and forward search (but without synchronization). We use the 1-mismatch problem to convey the flavor of the method. The mismatch (if there is one at all) either occurs (a) in the first half or (b) in the second half of the pattern. Let $s = \lfloor \frac{m}{2} \rfloor$.

(a) In this case, the second half $P[s+1..m]$ of the pattern must match exactly, and the procedure call $backwardSearch(P[s+1..m])$ returns the $P[s+1..m]$-interval $[lb..rb]$ (if it exists). It then tries to extend this exact match to the left, allowing for one mismatch. This is exactly what the procedure $k\text{-}mismatch(P, s, 1, [lb..rb])$ does.