Genome-scale algorithmics

Veli Mäkinen
Faculty of Technology – Genome Informatics
Bielefeld University
vmakinen@cebitec.uni-bielefeld.de

November 8, 2012

Contents

Chapter 1

Notation

A *string* (or *word* or *sequence*) $T = t_1 t_2 \cdots t_n$ is a sequence of *characters* (or *symbols* or *letters*) from an *alphabet* Σ. A *substring* of T is any string $T_{i \ldots j} = t_i t_{i+1} \cdots t_j$, where $1 \leq i \leq j \leq n$. A *suffix* of T is any substring $T_{i \ldots n}$, where $1 \leq i \leq n$. A *prefix* of T is any substring $T_{1 \ldots j}$, where $1 \leq j \leq n$. We sometimes also use notation $T[i, j] = T[i \ldots j] = T_{i,j}$ for $T_{i \ldots j}$.

The length of a string T is denoted $|T|$. The cardinality of a set U is also denoted $|U|$.

The set of all strings is denoted Σ^* and the set of all strings of length n is denoted Σ^n.

We use the common Big-O notation for complexities. Informally, $O(f(x))$ denotes something that grows at most as fast as $f(x)$, $o(f(x))$ something that grows strictly slower than $f(x)$, $\Theta(f(x))$ something that grows as fast as $f(x)$, $\Omega(f(x))$ something that grows at least as fast as $f(x)$, and $\omega(f(x))$ something that grows strictly faster than $f(x)$.

Chapter 2

Alignments Revisited

We will first explore general alignment techniques and then proceed into ones tailored for biosequences. For self-containedness we repeat some familiar concepts.

2.1 Edit Distance

Let $A = a_1 \cdots a_m \in \Sigma^*$ and $B = b_1 \cdots b_n \in \Sigma^*$ be two sequences. Edit distance is defined through edit operations, of which most common are

E1. *Deletion*: a_i does not correspond to any character in B, $a_i \to \epsilon$.

E2. *Insertion*: b_j does not correspond to any character in A, $\epsilon \to b_j$.

E3. *Substitution*: a_i corresponds to b_j, $a_i \neq b_j$, $a_i \to b_j$.

Operations E3 shall not cross:

- If $a_i \to b_j$ an $a_{i'} \to b_{j'}$, then $i < i'$, if and only if $j < j'$.

Definition 2.1.1 *Given sequences A and B, their* edit distance *$D(A, B)$ is the smallest amount of operations E1, E2 and E3 to convert A into B.*

Edit distance is also called *Levenshtein* distance. If only operation E3 is allowed, then edit distance is called *Hamming* distance. It can be shown that edit distance is a *metric*:

1) $D(A, B) \geq 0$,

2) $D(A, B) = 0$, if and only if $A = B$,

3) $D(A, B) = D(B, A)$,

4) $D(A,C) \leq D(A,B) + D(B,C)$.

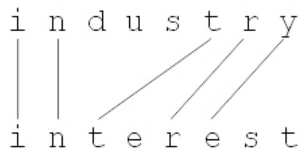
Example 2.1.2 $A = \texttt{abba}$, $B = \texttt{bbb}$

```
abba                          abba
|:||   3 operations           |::|   2 operations
bb b                          bbb
```

At least two operations are required, because $|A| = |B| + 1$ and B contains one less b than A. Hence, $D(A,B) = 2$.

Visualization of edits

1. Trace



Also identity operations are shown. Lines should not cross.

2. Alignment

```
indust-ry--
in---terest
```

3. Listing of operations

industry	$d \to \epsilon$	intery	$\epsilon \to e$
inustry	$u \to \epsilon$	interey	$y \to s$
instry	$s \to \epsilon$	interes	$\epsilon \to t$
intry	$\epsilon \to e$	interest	

2.1.1 Edit distance computation

Let us denote $d_{ij} = D(a_1 \cdots a_i, b_1 \cdots b_j)$, $0 \le i \le m$, $0 \le j \le n$.

Theorem 2.1.3 *Edit distances d_{ij} and especially $d_{mn} = D(A,B)$ can be computed using the recurrence*

$$d_{ij} = \min\{d_{i-1,j-1} + (\textbf{\textit{if}}\ a_i = b_j\ \textbf{\textit{then}}\ 0\ \textbf{\textit{else}}\ 1), d_{i-1,j} + 1, d_{i,j-1} + 1\}, \quad (2.1)$$

where $1 \le i \le m$, $1 \le j \le n$, and using initialization

$$\begin{aligned} d_{00} &= 0, \\ d_{i0} &= i, 1 \le i \le m, \ \textit{and} \\ d_{0j} &= j, 1 \le j \le n. \end{aligned}$$

Proof. Induction, exercise for the reader.

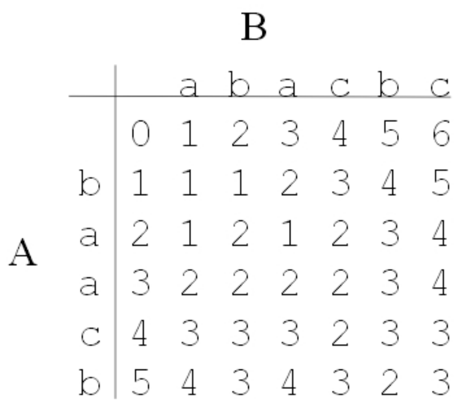
Values (d_{ij}), $0 \le i \le m$, $0 \le j \le n$, can be computed using *tabulation*, i.e. *dynamic programming*:

Algorithm 2.1.4 *$D(A,B)$ computation using dynamic programming, basic version.*

Input: $A = a_1 a_2 \cdots a_m$, $B = b_1 b_2 \cdots b_n$
Output: Matrix (d_{ij}), $0 \le i \le m$, $0 \le j \le n$, $d_{mn} = D(A,B)$

```
(1)  for $i := 0$ to $m$ do $d_{i0} := i$; (* initialization *)
(2)  for $j := 1$ to $n$ do $d_{0j} := j$;
(3)  for $j := 1$ to $n$ do
(4)      for $i := 1$ to $m$ do
(5)          $d_{ij} := \min\{d_{i-1,j-1} + (\textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } 1), d_{i-1,j} + 1, d_{i,j-1} + 1\}$
```

Example 2.1.5 $A = \texttt{baacb}, B = \texttt{abacbc}$



		B						
			a	b	a	c	b	c
		0	1	2	3	4	5	6
	b	1	1	1	2	3	4	5
	a	2	1	2	1	2	3	4
A	a	3	2	2	2	2	3	4
	c	4	3	3	3	2	3	3
	b	5	4	3	4	3	2	3

$D(A, B) = d_{mn} = d_{5,6} = 3.$

The time requirement of Algorithm 2.1.4 is $\Theta(mn)$. The space requirement is the same, but easily improved to $O(m)$; Values d_{ij} depend from each others as follows:

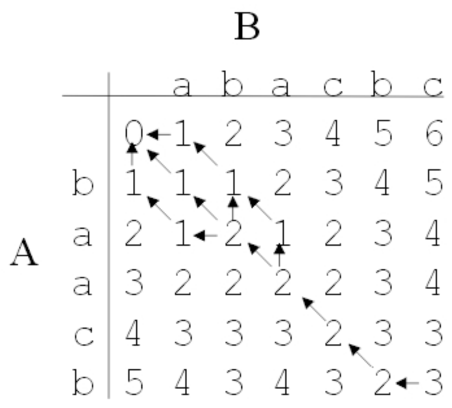
$$\begin{tikzcd}
d_{i-1,j-1} \arrow[dr] & d_{i-1,j} \arrow[d] \\
d_{i,j-1} \arrow[r] & d_{i,j}
\end{tikzcd}$$

To compute values d_{ij}, $1 \leq i \leq m$, at column j is is sufficient to know values $d_{i,j-1}$, $1 \leq i \leq m$, at column $j - 1$. We leave for the reader to modify Algorithm 2.1.4 to use only one vector of length $m + 1$ in the computation.

Tracing the edit operations

As typical in dynamic programming, the steps (here edit operations) chosen for the optimum solution (here d_{mn}) can be traced back afterwards. From d_{mn} we can trace back towards d_{00} and re-evaluate the decisions at each (d_{ij}). There is no need to store explicit pointers.

Example 2.1.6



		a	b	a	c	b	c
	0	1	2	3	4	5	6
b	1	1	1	2	3	4	5
a	2	1	2	1	2	3	4
a	3	2	2	2	2	3	4
c	4	3	3	3	2	3	3
b	5	4	3	4	3	2	3

Four different solutions are found, e.g.:

$$(a_1 \rightarrow b_1), (a_2 \rightarrow b_2), (A_{3...5} \rightarrow B_{3...5}), (\epsilon \rightarrow b_6)$$

Some improvements

It is possible to speed-up edit distance computation in many ways. Several algorithms exist with running time $O(mn/\log n)$ [MP80, Mye99, CLZU02] (differing completely in the techniques and in the assumptions on the machine model). Also, it is possible to develop faster algorithms for the case where the edit distance is small or large. For example, using the diagonal method [Ukk85] one can compute edit distance in $O(D(A,B)m)$ time. Many of the techniques for speeding-up edit distance computation are quite specific, and cannot easily be modified to work on any variation of edit distance, like those tailored for biological sequences. We will next cover one speed-up trick applicable to many variants of sequence alignment: *sparse dynamic programming*.

2.2 Sparse Dynamic Programming

The goal in sparse dynamic is to compute only some cells of the dynamic programming matrix. Let $M(A,B)$ denote the set of matching character pairs, that is, $M(A,B) = M = \{(i,j) \mid a_i = b_j\}$. We will next show how to compute edit distances $d_{i,j} = D'(A_{1\dots i}, B_{1\dots j})$ only for pairs $(i,j) \in M$, for $D'(A,B)$ being the edit distance where substitution operation is forbidden.

Longest common subsequence and (deletion/insertion) edit distance

Definition 2.2.1 *Given sequences A and B, the (deletion/insertion) edit distance $D'(A,B)$ is the smallest amount of operations E1 and E2 (see page 5) to convert A into B.*

We observe the following connection to *longest common subsequence* -problem.

Definition 2.2.2 *Sequence $C = c_1c_2\cdots c_r$ is the* subsequence *of $A = a_1a_2\cdots a_m$, if C can be obtained by deleting zero of more characters from A. Sequence C is the* longest common subsequence *of A and B, $LCS(A,B) = C$, if C is the longest sequence that is a subsequence of both A and B.*

Theorem 2.2.3 *a) $|LCS(A,B)| = (|A| + |B| - D'(A,B))/2$.*

b) Let

(1) $a_{i_1} \to \epsilon, a_{i_2} \to \epsilon, \dots, a_{i_p} \to \epsilon$ and

(2) $\epsilon \to b_{j_1}, \epsilon \to b_{j_2}, \dots, \epsilon \to b_{j_r}$

be the deletions and insertions in the optimal listing of edit operations corresponding to $D'(A,B)$. Then $LCS(A,B) = C$, where C equals A after deletions (1) and C equals B after deletions corresponding to the insertions (2) inverted.

Proof.

b) Due to the construction C is clearly a subsequence of A and B. If C is not longest possible, we have $|C| < |C'|$, where $C' = LCS(A, B)$. Then we can convert A to B with $(|A| - |C'|)$ deletions and $(|B| - |C'|)$ insertions, that is

$$D'(A,B) \quad \le |A| - |C'| + |B| - |C'| < |A| - |C| + |B| - |C| = D'(A,B),$$

which is a contradiction. Hence $C = LCS(A, B)$.

a) From b) we have $|LCS(A, B)| = |A| - p$ and $|LCS(A, B)| = |B| - r$, therefore

$$2 \cdot |LCS(A,B)| \quad = |A| + |B| - (p + r) = |A| + |B| - D'(A,B).$$

$\square$

Example 2.2.4 *$LCS($`stockholm`, `tukholma`$)$ = `tkholm`; $D'($`stockholm`, `tukholma`$) = 5$; $|tkholm| = 6 = (9 + 8 - 5)/2$.*

Modifying previous algorithms for $D'(A, B)$.

To compute $D'(A, B)$ the recurrence for (d_{ij}) becomes

$$d_{ij} = \min\{d_{i-1,j-1} + (\textbf{if } a_i = b_j \textbf{ then } 0 \textbf{ else } \infty), d_{i-1,j} + 1, d_{i,j-1} + 1\}$$

There exists bit-parallel algorithms for $D'(A, B)$ with running time $O(\lceil \frac{n}{w} \rceil m)$ that together with the *shortest detour* technique give running time $O(\lceil \frac{d}{w} \rceil m)$. There exist also algorithms that work faster when $LCS(A, B)$ is large.

$O(|M| \log m)$ time algorithm to compute $D'(A, B)$

There are several different algorithms for computing LCS using sparse dynamic programming. First one is the Hunt-Szymanski algorithm [HS77]. We will explore a variant of this algorithm [MNU03], that extends to many directions.

Theorem 2.2.5 *For values $d_{ij} = D'(A_{1\ldots i}, B_{1\ldots j})$, $(i, j) \in M$, where $M = M(A, B) = \{(i, j) \mid a_i = b_j\} \cup \{(0, 0)\}$, holds*

$$d_{ij} = \min\{d_{i',j'} + i - i' + j - j' - 2 \mid i' < i, j' < j, (i', j') \in M\}, \qquad (2.2)$$

with initialization $d_{0,0} = 0$. We have $D'(A, B) = d_{m+1,n+1}$.

Proof. Consider consecutive characters of A, $a_{i'}$ and a_i, in $LCS(A,B)$. They have counterparts $b_{j'}$ and b_j in B. The corresponding optimal listing of edit operations for $D'(A,B)$ contains only deletions and insertions between the two identity operations $a_{i'} \to b_{j'}$ and $a_i \to b_j$. The smallest number of deletions and insertions to convert $A_{i'+1\ldots i-1}$ into $B_{j'+1\ldots j-1}$ is $i-1-(i'+1)+1+j-1-(j'+1)+1 = i-i'+j-j'-2$. The recurrence considers for all pairs (i,j), $a_i = b_j$, all possible preceding pairs (i',j'), $a_{i'} = b_{j'}$; from these the one is chosen that minimizes the overall cost of converting first $A_{1\ldots i'}$ into $B_{1\ldots j'}$ (cost $d_{i',j'}$), and then $A_{i'\ldots i}$ into $B_{j'\ldots j}$ (cost $i-i'+j-j'-2$). The initialization is correct; if $a_i \to b_j$ is the first identity operation, the cost of converting $A_{1\ldots i-1}$ into $B_{1\ldots j-1}$ is $i-1+j-1 = d_{0,0}+i-0+j-0-2 = d_{i,j}$. Make an induction assumption that all values $d_{i',j'}$, $i' < i$, $j' < j$, are computed correctly. Then by induction it follows that each d_{ij} receives the correct value. Finally, $d_{m+1,n+1} = D'(A,B)$, because if (i',j') is the last edit operation, converting $A_{i'+1\ldots m}$ into $B_{j'+1\ldots n}$ costs $m-i'+n-j' = d_{m+1,n+1} - d_{i'-j'}$. $\square$

Algorithm derivation from recurrence.

Let us write (2.2) as

$$d_{ij} \quad = i+j-2+\min\{d_{i',j'} - i' - j' \mid i' < i, j' < j, (i',j') \in M\},$$

by bringing all values out from the minimization that are not affected. Compute values d_{ij} in the *reverse column-order* $<^{rc}$:

$$(i',j') \quad <^{rc} (i,j) \text{ if and only if } j' < j \text{ or } (j' = j \text{ and } i' > i).$$

The observation is that if all values $d_{i',j'}$, $(i',j') <^{rc} (i,j)$ are computed before computing d_{ij}, the condition $j' < j$ does not need to be taken into account separately; for all values $d_{i',j'}$ already computed, the condition $i' < i$ is enough to guarantee that also $j' < j$. We can write the recurrence as

$$d_{ij} = i+j-2+\min\{d_{i',j'} - i' - j' \mid i' < i, (i',j') <^{rc} (i,j), (i',j') \in M\}. \quad (2.3)$$

The idea of the algorithm is to consider cells $(i',j') \in M$ in the reverse column-order: Store each value $d_{i',j'} - i' - j'$ in a data structure $\mathcal{T}$ (to be defined) with key i' so that the minimum among values with key $i' < i$ can be retrieved from $\mathcal{T}$. Let this minimum be d. Then $d_{ij} = i+j-2+d$. Figure illustrates the situation.

Lemma 2.2.6 *The following operations can be supported with a binary search tree $\mathcal{T}$ in time $O(\log n)$, where n is the number of nodes in the tree.*

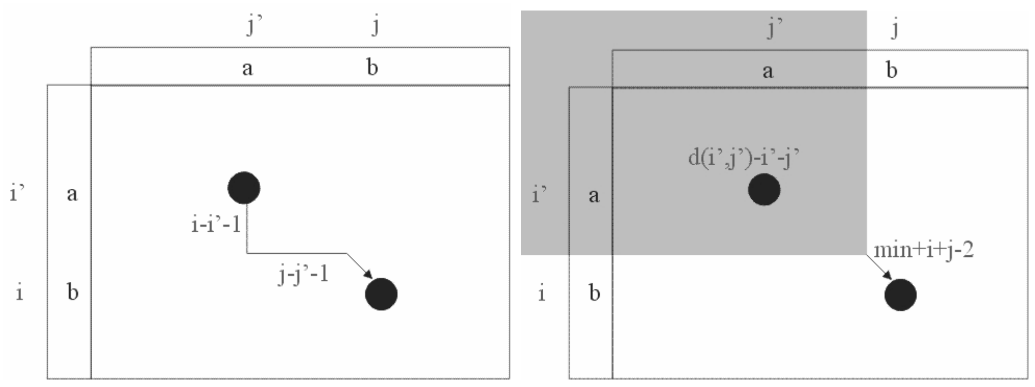


Figure 2.1: Geometric interpretation of recurrence (2.3).

$Insert(v, i)$: *Add value v to the tree with key i. If key i is already in the tree, replace its value v' with* $\min(v, v')$.

$v = Minimum(l, r)$: *Returns the minimum value v from nodes $\{i\}$ that belong to the interval $l \leq i \leq r$.*

Proof. Let us consider a binary tree with key and value pairs stored only in its leaves. The keys define a total order for the leaves. Each internal node contains a search key such that the key of node s is greater or equal to the largest key in its left subtree, and smaller that any key in its right subtree. As known, this kind of tree can be maintained balanced so that each root to leaf path has length $O(\log n)$.

Store for each internal node s the minimum value among the values $v(i)$ associated to the leaves i under it. Let us denote this minimum value $v(s)$. These values can be easily maintained when inserting leaves and when balancing the tree; only values on the paths from the inserted leaf / constant number of nodes towards the root are affected.

It is hence sufficient to show that query $Minimum(l, r)$ can be answered in $O(\log n)$ time: Find node s, where the search paths to keys l and r separate (can be the root, or empty when no keys in the query interval). Let $path(s, l)$ denote the set of nodes through which the path from s goes when searching for key l, excluding node s and leaf L where the search ends. Similarly, let $path(s, r)$ denote the set of nodes through which the path from s goes when searching for key r, excluding node s and leaf R where the search ends.

Now for each node in $path(s, l)$, where the path continues to the left, holds that the keys i in the right subtree are at least l and at most r. Choose $vl = \min_{s' \in S'}(v(s'))$, where S' is the set of roots of there right subtrees.

Similarly for each node in $path(s, r)$, where the path continues to the right, holds that the keys i in the left subtree are at most r and at least l. Choose $vl = \min_{s'' \in S''}(v(s''))$, where S'' is the set of roots of there left subtrees. The final result is the minimum of vl, vr, (**if** $L = l$ **then** $v(L)$ **else** ∞), and (**if** $R = r$ **then** $v(R)$ **else** ∞).

The correctness follows from the fact that the subtrees of nodes in $S' \cup S''$ contain all keys that belong to the interval $[l, r]$, and only them (excluding leaves L and R, which are taken into account separately). Running time is clearly $O(\log n)$. □

Algorithm 2.2.7 *Computation of $D'(A, B)$ with sparse dynamic programming*

Input: Set $M = \{(i, j) \mid a_i = b_j\}$ in reverse column-order, $|A| = m \leq n = |B|$

Output: $D'(A, B)$

```
(1)  $\mathcal{T}.Insert(0, 0)$; (* add $d_{0,0} = 0$ with key 0 *)
(2)  **for** $p := 1$ **to** $|M|$ **do begin**
(3)        $(i, j) := M[p]$;
(4)        $d := i + j - 2 + \mathcal{T}.Minimum(-\infty, i - 1)$; (* $d = d_{i,j}$ *)
(5)        $\mathcal{T}.Insert(d - i - j, i)$; **end**;
(6)  **return** $\mathcal{T}.Minimum(-\infty, m) + m + n$; (* $d_{m+1,n+1}$ *)
```

Theorem 2.2.8 *Distance $D'(A, B)$ or $|LCS(A, B)|$ can be computed using Algorithms 2.2.7 in time $O((n + |M|) \log m)$ and space $O(|M|)$, where $M = \{(i, j) \mid a_i = b_j\}$.*

Proof. Reverse column-order guarantees that the call $\mathcal{T}.Minimum(-\infty, i - 1)$ corresponds to taking minimum in (2.3). The only difference is that $\mathcal{T}$ contains only the smallest value at each row i', which does not however affect the correctness. Algorithm calls $O(|M|)$ times the operations in $\mathcal{T}$. Each of these take $O(\log m)$ time, as there can only be $m+1$ leaves at a time. Finally, set M can be easily constructed in $O(|\Sigma| + m + n + |M|)$ time on constant alphabet, or in general $O((m + n) \log m + |M|)$ time (exercise). As $m \leq n$, the claim holds. □

Observation 2.2.9 *The previous theorem can be improved; one can achieve $O(n \log m + |M| \log \log m)$ time and $O(m)$ space. The improvement is based on replacing the binary search tree with a more efficient data structure. Space requirement can be improved by constructing M column by column simultaneously with Algorithm 2.2.7.*

Sparse dynamic programming with $D(A, B)$.

The previous method for computing $D'(A, B)$ can be extended to compute unit cost edit distance $D(A, B)$ as well. The recurrence becomes more complex and one has to resort to two-dimensional range minimum queries. The running time then is $O(n \log m + |M| \log n \log \log m)$. With somewhat different techniques (and somewhat more complex algorithm) one can achieve $O((n + |M|) \log \log n)$ time also for distance $D(A, B)$ [EGGI92, GP92].

2.3 Approximate String Matching

Let $S = s_1 s_2 \cdots s_n \in \Sigma^*$ be a *text* string and $P = p_1 p_2 \cdots p_m \in \Sigma^*$ a *pattern* string. Let k be a constant, $0 \leq k \leq m$. The *k mismatches* problem is to search for all substrings X of S, $|X| = |P|$, that differ from P in at most k positions. The *k errors* problem is to search for all ending positions of substrings X in S for which hold $D(P, X) \leq k$. Instead of fixed k, one can consider error level $\alpha = k/m$.

First row to zero -trick

Let $D(P, X)$ denote the edit distance with operations E1-E3 such that each operation has cost 1. Define (h_{ij}), $0 \leq i \leq m, 0 \leq j \leq n$ with recurrence:

$$\begin{aligned} h_{0j} &= 0, 0 \leq j \leq n, \\ h_{i0} &= i, 1 \leq i \leq m, \text{ and} \\ h_{ij} &= \min\{h_{i-1,j-1} + (\textbf{if } p_i = s_j \textbf{ then } 0 \textbf{ else } 1), h_{i-1,j} + 1, h_{i,j-1} + 1\}. \end{aligned}$$

Matrix (h_{ij}) is like (d_{ij}), but first row is initialized to 0. The approximate occurrences of the pattern are found on the last row of (h_{ij}), as follows.

Theorem 2.3.1 *Let an optimal path to h_{mj} start at h_{0r}. Then $D(P, s_{r+1} s_{r+2} \cdots s_j) = h_{mj}$ and $h_{mj} = \min\{D(P, s_t s_{t+1} \cdots s_j) \mid t \leq j\}$.*

Proof. For contradiction, assume that there is $t \neq r + 1$ for which $d = D(P, s_t \cdots s_j) < D(P, s_{r+1} \cdots s_j)$. It follows that the optimal path from $h_{0,t-1}$ to h_{mj} has cost $d < h_{mj}$, which is a contradiction as h_{mj} is the cost of the optimal path.. □

Let $s_{r+1} s_{r+2} \cdots s_j$ as in Theorem 2.3.1. Mark the counterpart of h_{mj} in S as $V(m, j) = s_{r+1} s_{r+2} \cdots s_j$.

Algorithm 2.3.2 *Approximate string matching, k errors*

```
Input: $P$, $S$, $k$.
Output: Approximate occurrences of $P$ in $S$, whose edit distance from $P$ is at most $k$.
(1) for $i = 0$ to $m$ do $h_{i0} := i$;
(2) for $j = 1$ to $n$ do begin
(3)     $h_{0j} := 0$;
(4)     for $i := 1$ to $m$ do
(5)         $h_{ij} := \min\{h_{i-1,j-1} +$ (if $p_i = s_j$ then $0$ else $1$), $h_{i-1,j} + 1$, $h_{i,j-1} + 1\}$;
(6)     if $h_{mj} \leq k$ then begin
(7)         compute $V(m, j)$; (* trace back optimal path*)
(8)         $write(j, V(m, j))$; end end (* end position and matching sequence*)
```

Algorithm 2.3.2 has running time $O(mn)$ and space requirement $O(mn)$. To save space, one can e.g. use a circular buffer to maintain only columns $j - m - k, \ldots, j - 1$ in memory. The space is then $O(m^2)$. If only the end positions of occurrences are of interest, it is sufficient to maintain only previous column.

Theorem 2.3.3 *Approximate string matching under k errors can be solved using Algorithm 2.3.2 in $O(mn)$ time and $O(m)$ space, when the outputs are the occurrence end positions, and in space $O(m^2)$, when the outputs are the occurrence substrings.*

The same algorithm allowing only substitutions solves the k mismatches problem. Faster algorithms exist, for example both problems can be solved in $O(kn)$ time [LV88] using some suffix tree techniques together with the diagonal algorithm in [Ukk85].

Myers' bit-parallel algorithm [Mye99] extends easily to approximate string matching (with a bit-equivalent of the zero the first row -trick). The running time is $O(\lceil \frac{m}{w} \rceil n)$ in the worst case and $O(\lceil \frac{k}{w} \rceil n)$ in the average case.

2.4 Fundamentals of Biological Sequence Alignment

Biological sequence alignment is typically defined through maximum weight alignment rather than minimum amount operations in alignment as in edit distance. Algorithm-wise the differences are minimal, but the maximization framework allows a probabilistic interpretation for alignments. Let us define $s(a,b)$ the *score* for aligning symbol a with symbol b (or identically substituting a with b). Let d be the *penalty* of an *indel*, that is, penalty for inserting a symbol or deleting a symbol. Then the weight of an alignment is defined by the sum of substitution scores minus the sum of indel penalties in it.

For example, the following *substitution matrix* is often used for DNA sequence alignments:

$s(a,b)$	'A'	'C'	'G'	'T'
'A'	1	-1	-0.5	-1
'C'	-1	1	-1	-0.5
'G'	-0.5	-1	1	-1
'T'	-1	-0.5	-1	1

The judgement for the scores is that so-called transition mutations (here with score -0.5) are twice as frequent as so-called transversions (here with score -1). Let $d = 1$. The weight or total score of the alignment

```
A C C - G A T G
|   |   | | | |
A - C G G C T A
```

is $1-1+1-1+1-1+1-0.5 = 0.5$.

2.4.1 Global alignment

To define the problem of finding maximum scoring alignment, i.e., *global alignment* problem, we first need to formalize the concept of an alignment. An *alignment* of A and B is a pair of sequences $U = u_1u_2\cdots u_h$ and $L = l_1l_2\cdots l_h$ such that A is subsequence of U, B is subsequence of L, U contains $h-m$ symbols $'-'$, and L contains $h-n$ symbols $'-'$. The *weight* or *score* of alignment U, L is $W(U,L) = \sum_{i=1}^{h} s(u_i, l_i)$, where $s(a,'-') = s('-',b) = -d$. Let $S(A,B)$ be the total score of the optimal alignment of A and B, that is, $S(A,B) = \max_{(U,L)\in\mathcal{A}(A,B)} W(U,L)$, where $\mathcal{A}(A,B)$ denotes the set of all valid alignments of A and B. Then the global alignment problem is to find an alignment with score $S(A,B)$.

Let us denote $s_{ij} = S(a_1\cdots a_i, b_1\cdots b_j)$, $0 \le i \le m$, $0 \le j \le n$.

Theorem 2.4.1 *Global alignment scores s_{ij} and especially $s_{mn} = S(A,B)$ can be computed using the recurrence*

$$s_{ij} = \max\{s_{i-1,j-1} + s(a_i,b_j), s_{i-1,j} - d, s_{i,j-1} - d\}, \qquad (2.4)$$

where $1 \le i \le m$, $1 \le j \le n$, and using initialization

$$\begin{aligned} s_{00} &= 0, \\ s_{i0} &= -id, 1 \le i \le m, \text{ and} \\ s_{0j} &= -jd, 1 \le j \le n. \end{aligned}$$

The correctness proof is identical to the corresponding theorem on edit distance; in the sequel, we leave similar proofs for the reader. For completeness the algorithm is given in Algorithm 2.4.2. The algorithm is also called *Needleman-Wunsch* after the inventors.

Algorithm 2.4.2 *$S(A,B)$ computation using dynamic programming, basic version.*

```
Input: $A = a_1 a_2 \cdots a_m$, $B = b_1 b_2 \cdots b_n$
Output: Matrix $(s_{ij})$, $0 \le i \le m$, $0 \le j \le n$, $s_{mn} = S(A,B)$
(1) for $i := 0$ to $m$ do $s_{i0} := -id$; (* initialization *)
(2) for $j := 1$ to $n$ do $s_{0j} := -jd$;
(3) for $j := 1$ to $n$ do
(4)     for $i := 1$ to $m$ do
(5)         $s_{ij} := \max\{s_{i-1,j-1} + s(a_i, b_j), s_{i-1,j} - d, s_{i,j-1} - d\}$
```

Tracing back the optimal alignment(s) is identical to the corresponding procedure for edit distances.

An *approximate string matching version of global alignment* is easy to derive: zero the first row and check for maximum values in the last row.

2.4.2 Local alignment

Local alignment is the problem of finding the substrings of A and B with highest scoring alignment. For an arbitrary scoring scheme this problem can be solved e.g. by applying global alignment for all suffix pairs from A and B in $O((mn)^2)$ time. However, the scoring schemes are designed so that local alignments with score less than zero are not statistically significant; it is more likely to find a zero scoring alignment in random strings. Therefore, one can use a version of the zero the first row -trick: use global alignment recurrence, but add an option to start a new alignment at any suffix/suffix pair by assigning score 0 for an empty alignment. This observation is the *Smith-Waterman* algorithm for local alignment.

Let us denote $l_{ij} = \max\{S(a_{i'} \cdots a_i, b_{j'} \cdots b_j \mid i' \leq i, j' \leq j\}$, $0 \leq i \leq m$, $0 \leq j \leq n$.

Theorem 2.4.3 *Non-negative local alignment scores l_{ij} can be computed using the recurrence*

$$l_{ij} = \max\{0, l_{i-1,j-1} + s(a_i, b_j), l_{i-1,j} - d, l_{i,j-1} - d\}, \tag{2.5}$$

where $1 \leq i \leq m$, $1 \leq j \leq n$, and using initialization

$$\begin{aligned} l_{00} &= 0, \\ l_{i0} &= 0, 1 \leq i \leq m, \text{ and} \\ l_{0j} &= 0, 1 \leq j \leq n. \end{aligned}$$

It is easy to modify Algorithm 2.4.2 accordingly.

After the computation, one can locate $\max l_{ij}$ or maintain the maximum value during computation. In the latter case, $O(m)$ space is sufficient for finding the maximum (computing the recurrence e.g. column by column). To trace back the alignment, one cannot usually afford $O(mn)$ space to store the matrix. Instead one can easily modify the dynamic programming algorithm to maintain, in addition to maximum score, also the left-most positions in A and B, say i' and j', respectively, where a maximum scoring alignment ending at l_{ij} starts. Then it is easy to recompute a matrix of size $(i-i'+1) \times (j-j'+1)$ to output (all) local alignment(s) with maximum score ending at l_{ij}. The space is hence quadratic in the length of the longest local alignment with maximum score. The details are left for an exercise to the reader.

2.4.3 Overlap alignment

Overlap alignment is the problem of finding a maximum scoring alignment between any suffix of A and any prefix of B. Like in local alignment, we are interested only in alignments with score greater than zero. It is easy to see that again a variant of the first row to zero -trick works.

Let us denote $o_{ij} = \max\{S(a_{i'} \cdots a_i, b_1 \cdots b_j \mid i' \leq i\}$, $0 \leq i \leq m$, $0 \leq j \leq n$.

Theorem 2.4.4 *Non-negative overlap alignment scores o_{ij} can be computed using the recurrence*

$$o_{ij} = \max\{o_{i-1,j-1} + s(a_i, b_j), o_{i-1,j} - d, o_{i,j-1} - d\}, \tag{2.6}$$

where $1 \leq i \leq m$, $1 \leq j \leq n$, and using initialization

$$\begin{aligned} o_{00} &= 0, \\ o_{i0} &= 0, 1 \leq i \leq m, \text{ and} \\ o_{0j} &= -dj, 1 \leq j \leq n. \end{aligned}$$

Again, modifications to Algorithm 2.4.2 are minimal.

Overlap alignment is important for *fragment assembly*, where one tries to assemble a sequence given a random subset of its substrings as input: Computing the overlap alignments for all pairs of substrings and accepting those pairs with high enough score defines an *overlap graph* with substrings as nodes and edges weighted by the overlap score. Paths in the overlap graph represent possible partial assemblies for the target sequence. Many assembly methods have been developed building on top of the overlap graph.

2.4.4 Affine gap scores

In sequence evolution, simple mutations are more frequent than indels. However, indels often occur in blocks, and the simple linear scoring scheme with penalty d for each base is not well grounded. More realistic schemes have been developed. We cover here the *affine gap model*, as it, although not perfect model either, admits an efficient algorithm.

Recall the definition of alignment through sequences U and L containing A and B, respectively, as subsequences and gap symbols $'-'$ to form the alignment. We say there is a *run of gaps* $U[l,r], L[l,r]$ in the alignment if $U[i] =' -'$ or $L[i] =' -'$ for all $l \leq i \leq r$. Identically, we say there is a *run of matches* $U[l,r], L[l,r]$ in the alignment if $U[i]! =' -'$ and $L[i]! =' -'$ for all $l \leq i \leq r$. Any alignment can be partitioned into a sequence of runs of matches and gaps: Let $U = U_1U_2 \cdots U_k$ and $L = L_1L_2 \cdots L_k$ denote such a partitioning, where for all i $|U_i| = |L_i|$ and U_i, L_i is either a run of matches or a run of gaps. Let $\mathcal{P}(\mathcal{U}, \mathcal{L})$ denote the set of all possible such partitions. Let $W_G(U_i, L_i) = W(U_i, L_i)$ for a run of matches U_i, L_i and $W_G(U_i, L_i) = -\alpha - \beta(|U_i| - 1)$ for a run of gaps. The affine gap score $S_G(A, B)$ is defined as $S_G(A, B) = \max_{(U,L) \in \mathcal{A}(A,B)} \max_{(U_1 \cdots U_k, L_1 \cdots L_k) \in \mathcal{P}(U,L)} \sum_{i=1}^{k} W_G(U_i, L_i)$, where $\mathcal{A}(A, B)$ denotes the set of all valid alignments of A and B. Then the global alignment problem under affine gap model is to find an alignment with score $S_G(A, B)$. Here α is the penalty for opening a gap and β the penalty for extending a gap, with $\beta < \alpha$. The idea is that starting a gap always costs significantly, but its small extension not that much.

Let us denote $sg_{ij} = S_G(a_1 \cdots a_i, b_1 \cdots b_j)$, $0 \leq i \leq m$, $0 \leq j \leq n$.

Theorem 2.4.5 *Global alignment under affine gap scores sg_{ij} and especially $sg_{mn} = S_G(A, B)$ can be computed using the recurrence*

$$sg_{ij} = \max_{i' \leq i, j' \leq j, i'+j' < i+j} \{sg_{i-1,j-1} + s(a_i, b_j), sg_{i',j'} - \alpha - \beta(j - j' + i - i')\}, \quad (2.7)$$

where $1 \leq i \leq m$, $1 \leq j \leq n$, and using initialization

$$\begin{aligned} sg_{00} &= 0, \\ sg_{i0} &= -\alpha - \beta(i-1), 1 \leq i \leq m, \text{ and} \\ sg_{0j} &= -\alpha - \beta(j-1), 1 \leq j \leq n. \end{aligned}$$

Proof. Initialization is correct: For $U = A[1, i]$ and $L =' -'^i$, $W_G(U, L) = -\alpha - \beta(i - 1)$. Similarly for $U =' -'^j$ and $L = B[1, j]$, $W_G(U, L) = -\alpha - \beta(j - 1)$. These are the values stored at first row and first column of the matrix. For induction, assume the recurrence is correct for i', j' such that $i' \leq i, j' \leq j$, and $i' + j' < i + j$ holds. In any alignment the last operation is (1) $a_i \to b_j$, (2) $a_i \to' -'$, or (3) $'-' \to b_j$. In case (1) maximum score is given by $sg_{i-1,j-1} + s(a_i, b_j)$ because by induction assumption $sg_{i-1,j-1}$ is the maximum scoring alignment for $A_{1,i-1}$ and $B_{1,j-1}$. For (2) the alignment ends with a run of gaps with a_i in the end. The run can start at any position $i' < i$ in A and at any position $j' \leq j$ in B. Since by induction assumption $sg_{i',j'}$ is the maximum scoring alignment for $A_{1,i'}$ and $B_{1,j'}$, inserting $B_{j'+1,j}$ and deleting $A_{i'+1,i}$ costs $-\alpha - \beta(i - i' + j - j')$, value sg_{ij} is correctly computed. Case (3) is analogous to (2) and leads to the same recurrence with $j' < j$ and $i' \leq i$. These cases can be combined to the common maximization over i', j' for which $i' \leq i, j' \leq j$, and $i' + j' < i + j$. □

The running time for evaluating the recurrence (2.7) is $O((mn)^2)$. Notice that the recurrence is very general in the sense that $-\alpha - \beta(j - j' + i - i')$ could be replaced with any other function on the gap length. Also, it is easy to modify the recurrence to compute variants of run of gaps definition: One could define separately a run of insertions and a run of deletions. Then, the recurrence changes so that one takes separately $\max_{i'<i} sg_{i',j} - \alpha - \beta(i - i')$ and $\max_{j'<j} sg_{i,j'} - \alpha - \beta(j - j')$. The running time reduces to $O(mn(m+n))$.

There are several ways to speed-up the affine gap score recurrences. The most common is to fill two or more matrixes simultaneously, where one is for alignments ending at match state, and the others are for being inside a run of gaps (or run of insertions and run of deletions).

Let $S_G(A, B|\text{ match}) = \max_{(U,L) \in \mathcal{A}_{\mathcal{M}}(A,B)} \max_{(U_1 \cdots U_k, L_1 \cdots L_k) \in \mathcal{P}(U,L)} \sum_{i=1}^{k} W_G(U_i, L_i)$, where $\mathcal{A}_{\mathcal{M}}(A, B)$ denotes the set of all valid alignments of A and B ending at a match. Let $S_G(A, B|\text{ gap}) = \max_{(U,L)\in\mathcal{A}_\mathcal{G}(A,B)} \max_{(U_1\cdots U_k, L_1\cdots L_k)\in\mathcal{P}(U,L)} \sum_{i=1}^{k} W_G(U_i, L_i)$, where $\mathcal{A}_\mathcal{M}(A, B)$ denotes the set of all valid alignments of A and B ending at a gap.

Let m_{ij} denote $S_G(A_{1,i}, B_{1,j}|\text{ match})$ and g_{ij} denote $S_G(A_{1,i}, B_{1,j}|\text{ gap})$.

Theorem 2.4.6 *Global alignment under affine gap scores sg_{ij} and especially $sg_{mn} = S_G(A, B)$ can be computed using the recurrence*

$$\begin{aligned} sg_{ij} &= \max(m_{ij}, g_{ij}) \\ m_{ij} &= \max\{m_{i-1,j-1} + s(a_i, b_j), g_{i-1,j-1} + s(a_i, b_j)\} \\ g_{ij} &= \max\{m_{i-1,j} - \alpha, g_{i-1,j} - \beta, m_{i,j-1} - \alpha, g_{i,j-1} - \beta\} \end{aligned} \tag{2.8}$$

where $1 \le i \le m$, $1 \le j \le n$, and using initialization

$$\begin{aligned} m_{00} &= 0, \\ m_{i0} &= -\infty, 1 \le i \le m, \\ m_{0j} &= -\infty, 1 \le j \le n, \\ g_{00} &= 0, \\ g_{i0} &= -\alpha - \beta(i-1), 1 \le i \le m, \textit{(and)} \\ g_{0j} &= -\alpha - \beta(j-1), 1 \le j \le n. \end{aligned}$$

Proof. First, $sg_{ij} = \max(m_{ij}, g_{ij})$ follows from the definitions. Initialization works correctly as in match matrix the first row and first column do not correspond to any valid alignment, and hence the initialization of gap matrix is identical to that of matrix sg. The correctness of m_{ij} computation follows easily by induction, as $\max\{m_{i-1,j-1}, g_{i-1,j-1}\}$ is the maximum score of all alignments where a match can be appended. The correctness of g_{ij} computation can be seen as follows. An alignment ending in a gap is either (1) opening a new gap or (2) extending an existing one (2). In case (1) it is sufficient to take the maximum of scores for aligning $A_{1,i-1}$ with $B_{1,j}$ and $A_{1,i}$ with $B_{1,j-1}$ so that the previous alignment ends with a match, and then add the cost for opening a gap. These maxima are given (by induction assumption) by $m_{i-1,j}$ and $m_{i,j-1}$. In case (2) it is sufficient to take the maximum of scores for aligning $A_{1,i-1}$ with $B_{1,j}$ and $A_{1,i}$ with $B_{1,j-1}$ so that the previous alignment end with a gap, and then add the cost for extending a gap. These maxima are given (by induction assumption) by $g_{i-1,j}$ and $g_{i,j-1}$. □

The running time is now reduced to $O(mn)$. Similar recurrence can be derived for local alignment and overlap alignment under affine gap score.

2.4.5 Invariant technique

It is instructive to study an alternative $O(mn)$ time algorithm for alignment under affine gap score. This algorithm uses an *invariant technique* similar to what we used earlier in sparse dynamic programming for longest common subsequence problem.

Consider the formula $sg_{i',j'} - \alpha - \beta(j - j' + i - i')$ inside the maximization in Equation (2.7). This can be equivalently written as $-\alpha - \beta(j+i) + sg_{i',j'} + \beta(j' + i')$. The maximization goes over valid values of i', j' and the first part of the formula, $-\alpha - \beta(j+i)$, is not affected. Recall the search tree $\mathcal{T}$ of Lemma 2.2.6. With a symmetric change it can be transformed to support range maximum query instead of range minimum query. Then by storing values $sg_{ij} + \beta(j+i)$ together with key i to the search tree, we can query the maximum value and add the invariant. We directly obtain Algorithm 2.4.7, i.e., an $O(mn \log m)$ time algorithm for computing global alignment under affine gap score.

Algorithm 2.4.7 *$S_G(A, B)$ computation using dynamic programming and range maximum queries.*

```
Input: $A = a_1 a_2 \cdots a_m$, $B = b_1 b_2 \cdots b_n$
Output: Matrix $(sg_{ij})$, $0 \leq i \leq m$, $0 \leq j \leq n$, $sg_{mn} = S_G(A, B)$
(1)  $\mathcal{T}.Insert(0, 0)$; (* tree initialization *)
(2)  for $i := 0$ to $m$ do $sg_{i0} := -\alpha - \beta(i-1)$; (* initialization *)
(3)  for $j := 1$ to $n$ do $sg_{0j} := -\alpha - \beta(j-1)$;
(4)  for $j := 1$ to $n$ do
(5)      for $i := 1$ to $m$ do
(6)          $sg_{ij} := \max\{sg_{i-1,j-1} + s(a_i, b_j),$
(7)          $\mathcal{T}.Maximum(0, i) - \alpha - \beta(i+j)\}$;
(8)          $\mathcal{T}.Insert(sg_{ij} + \beta(i+j), i)$
```

The use of a data structure in Algorithm 2.4.7 is in fact unnecessary. It is sufficient to maintain maximum value on each row, and then maintain on each column the maximum of row maxima. The modified $O(mn)$ time algorithm is given in Algorithm 2.4.8.

Algorithm 2.4.8 *$S_G(A,B)$ computation using dynamic programming and row/column maxima updating.*

```
Input: $A = a_1 a_2 \cdots a_m$, $B = b_1 b_2 \cdots b_n$
Output: Matrix $(sg_{ij})$, $0 \le i \le m$, $0 \le j \le n$, $sg_{mn} = S_G(A,B)$
(1)  for $i := 0$ to $m$ do (* initialization *)
(2)      $sg_{i0} := -\alpha - \beta(i-1)$;
(3)      $r_i = sg_{i0} + \beta i$;
(4)  for $j := 0$ to $n$ do
(5)      $sg_{0j} := -\alpha - \beta(j-1)$;
(6)      $c_j = sg_{0j} + \beta j$;
(7)  for $j := 1$ to $n$ do
(8)      for $i := 1$ to $m$ do
(9)          $sg_{ij} := \max\{sg_{i-1,j-1} + s(a_i, b_j)$,
(10)         $\max(c_j, r_i) - \alpha - \beta(i+j)\}$
(11)         $r_i = \max(r_i, sg_{ij} + \beta(i+j))$
(12)         $c_j = \max(c_j, r_i)$
```

We leave for the reader to prove the correctness of Algorithms 2.4.7 and 2.4.8.

2.5 Gene Alignment

Consider the problem of having an unknown protein sequence and looking for its counterpart (gene) in DNA. With prokaryotes, the task, call it *Prokaryote gene alignment*, is relative easy: just modify the approximate string matching version of global alignment so that $s_{i-1,j-1} + s(a_i, b_j)$ is replaced by $s_{i-1,j-3} + \max_{b \in \text{aminoacids}[B_{j-2,j}]} s(a_i, b)$, where A is the protein sequence, B is the DNA sequence and aminoacids$[xyz]$ is the set of amino acids coded by codon xyz. For symmetry, also $s_{i,j-1} - d$ should be replaced by $s_{i,j-3} - d$ and initialization modified accordingly.

With eukaryotes, introns need to be taken into account. Most common way to do this is to use affine gap penalties and assign large gap opening cost and very small gap extension cost in the DNA side, and use linear gap cost in the protein side. Call this approach *Eukaryote gene alignment with affine gaps*. We leave as an exercise for the reader to formulate an $O(mn)$ algorithm for solving this variant. However, this approach is clearly problematic because intron lengths can vary from tens to tens of thousands of nucleotides: there is no good choice for gap extension cost.

For simplicity of exposition, we consider in the sequel that the pattern $P = A$ is an RNA sequence instead of a protein and denote the DNA sequence $T = B$; the protein to DNA versions follow using the idea explained above.

Let us now develop a better approach for gene alignment in eukaryotes exploiting the fact that the number of introns in genes is usually quite small. For example, on human genome the average number of introns is 7.8 with maximum being 148[1]. We can fix the maximum number of introns allowed in an alignment as follows: Define e_{ijk} as the maximum scoring alignment of $P_{1,i}$ and $T_{j',j}$ for some j', using exactly k free runs of gaps in $T_{j',j}$. Call the problem of computing values $e_k = \max_j e_{mjk}$ for $0 \leq k \leq$ MAXINTRONS as *Eukaryote gene alignment with limited introns*. Then the following result follows easily for solving the problem.

Theorem 2.5.1 *Eukaryote gene alignment with limited introns problem can be solved computing the scores e_{ijk} using the recurrence*

$$e_{ijk} = \max\{e_{i-1,j-1,k} + s(a_i, b_j), e_{i-1,j,k} - d, e_{i,j-1,k} - d, \max_{j'<j} e_{i,j',k-1}\} \tag{2.9}$$

where $1 \leq i \leq m$, $1 \leq j \leq n$, and using initialization

$$\begin{aligned} e_{000} &= 0, \\ e_{i00} &= -di, 1 \leq i \leq m, \\ e_{0jk} &= 0, 1 \leq j \leq n, 0 \leq k \leq \min(j, \mathit{MAXINTRONS}), \ \textit{and} \end{aligned}$$

$e_{ijk} = -infty$ otherwise.

The proof is left for the reader. The running time is $O(mn^2\text{MAXINTRONS})$, but this can be easily improved to $O(mn\text{MAXINTRONS})$ by replacing $\max_{j'<j} e_{i,j',k-1}$ with $m_{i,k-1}$ that maintains the row maximum for each k during the computation of e_{ijk}'s. That is, $m_{ik} = \max(m_{ik}, e_{ijk})$. It is left as an exercise for the reader to find a correct evaluation order for the values e_{ijk} and m_{ik} so that they get computed correctly.

Obviously the values j giving the maximum for $\max_j e_{mjk}$ are plausible ending positions for the alignment. The alignment(s) can be traced back with the normal routine. It is left as an exercise to modify this recurrence to take more properties of genes into account, e.g. the start/stop codons and the conserved dinucleotides at intron/exon boundaries.

[1]`http://www.bioinfo.de/isb/2004040032/`

2.6 Exercises

1. Modify Algorithm 2.1.4 at page 7 to use only one vector of length $m+1$ in the computation.

2. Give pseudocode for local alignment using space $O(m)$.

3. Give an example of perfectly balanced binary search tree storing 8 (value,key) pairs in its leaves as described in Lemma 2.2.6. Give an example of a range minimum query for some non-empty interval.

4. A *van Emde Boas tree* (vEB tree) supports in $O(\log\log n)$ time insertions, deletions, and *predecessor queries* for values in interval $[1,n]$. Predecessor query returns the largest element i' stored in the vEB tree smaller than query element i. Show how the structure can be used instead of a balanced search tree of Lemma 2.2.6 to solve range minimum queries for semi-infinite intervals $(-\infty, i]$ (i.e. for the type of queries we used e.g. in the LCS algorithm).

5. Give pseudocode for tracing an optimal path for maximum scoring local alignment, using space quadratic in the alignment length.

6. Prove the correctness of Algorithms 2.4.7 and 2.4.8 at pages 23 and 24.

7. `SOLiD`[2] sequencing produces short reads of DNA in *colour-space* with a two-base defined by the matrix (row=first base, column=second base):

   ```
     A C G T
   A 0 1 2 3
   C 1 0 3 2
   G 2 3 0 1
   T 3 2 1 0
   ```

   For example, `T012023211202102` equals `TTGAAGCTGTCCTGGA` (first base is always given). Modify overlap alignment to work properly in the case where one of the sequences in `SOLiD` read and the other is a normal sequence.

8. Give pseudocode for prokaryote gene alignment.

9. Give pseudocode for eukaryote gene alignment with affine gaps.

10. Give pseudocode for eukaryote gene alignment with limited introns.

[2] TM Applied Biosystem

11. Modify the recurrence for gene alignment with limited introns so that the alignment must start with a start codon, end with an end codon, and contain `GT` and `AG` dinucleotides at intron boundaries. Can you still solve the problem in $O(mnk)$ time? Since there are rare exceptions when dinucleotides at intron boundaries are something else, how can you make the requirement softer?

12. Develop an algorithm for tracing an optimal path for maximum scoring local alignment, using space linear in the alignment length. *Hint.* Let $[i', i] \times [j', j]$ define the rectangle containing a local alignment. Assume you know j_{mid} for row $(i - i')/2$ where the optimal alignment goes through. Then you can independently recursively consider rectangles defined by $[i', (i - i')/2] \times [j', j_{\text{mid}}]$ and $[(i - i')/2] \times [j_{\text{mid}}, j]$. To find j_{mid} you may consider similar algorithm as was used in `MSA` to find alignments going through certain coordinate pair.

13. Develop a sparse dynamic programming solution for computing the longest common subsequence of multiple sequences. What is the expected size of the match set on a random sequences from an alphabet of size σ? *Hint. The search tree can be extended to higher dimensional range queries using recursion — see range trees from computational geometry literature).*

Chapter 3

Compressed Data Structures

In the next chapter, we will need some concepts from data compression & compressed data structures introduced next.

The goal in data structure compression is to represent the structure in small space, but at the same time preserve its *functionality*.

Example 3.0.1

There are $\binom{2n}{n}/(n+1)$ different binary trees of n nodes. Hence it is possible to construct a bijection f:binary trees $\rightarrow \{0, 1, \dots \binom{2n}{n}/(n+1)-1\}$. For each binary tree T one can efficiently compute $f(T)$, and vice versa([KM06]). Coding ($f(T)$ as binary number) is optimal, if all binary trees are as likely. However, the coding is not functionality preserving, because one cannot implement procedures such as "proceed to the left child of node v" without decompressing the whole encoding.

Surprisingly, a binary tree can be represented using $2n + o(n)$ bits so that the transitions from parent to children and back can be simulated in constant time [Jac89]. This follows from an easy reduction from trees to bitvector *rank* and *select* queries (see [Jac89]) covered in the sequel.

These kind of structures are the basis in replacing widely used bioinformatics data structures like suffix trees with practical alternatives that can handle much larger data sets. Since compressed suffix trees are quite technical, and also they are not (yet) heavily exploited in bioinformatics, we will focus on just compressed suffix arrays, that are more limited in use, but have been adopted widely in high-throughput sequencing data analysis.

Before the actual data structures, we will go through some basics of data compression.

3.1 Huffman coding

Huffman coding assigns a variable length *prefix code* to each symbol, i.e., each symbol has a code that is not a prefix of any other code. The algorithm works as follows:

1. Set the code of each symbol s_i to ϵ.
2. Construct sets $\{s_1\}, \ldots, \{s_\sigma\}$.
3. Find sets A and B, whose sum of probabilities is smallest.
4. Add a bit 0 in front of the codes of symbols in A and bit 1 in front of the codes of symbols in B
5. Combine sets A and B.
6. Repeat steps 3-5, until only one set exists.

For example:

combine					s_i	code
{e} 0,3	{e} 0,3	{e} 0,3	{a,o} 0,4	•{u,y,i,e} 0,6	a	10
{a} 0,2	{a} 0,2	{u,y,i} 0,3	•{e} 0,3	•{a,o} 0,4	e	00
{o} 0,2	{o} 0,2	•{a} 0,2	•{u,y,i} 0,3		i	011
{i} 0,1	•{u,y} 0,2	•{o} 0,2			o	11
•{u} 0,1	•{i} 0,1				u	0100
•{y} 0,1					y	0101

It can be shown that Huffman algorithm minimizes $\sum_{i=1}^{\sigma} p(s_i)\ell_i$ over all prefix codes, and that each ℓ_i satisfies $\ell_i \leq -\log p(s_i) + 1$. Hence, for a text $T[1,n]$, Huffman coding requires at most $n(H(T)+1)$ bits plus the representation of the code table, where *empirical entropy* $H(T)$ is defined as

$$H(T) = -\sum_{i=1}^{\sigma} p(s_i) \log p(s_i), \tag{3.1}$$

and $p(s_i)$ is defined as the number of times s_i occurs in T divided by $|T|$.

Huffman tree is a keyword trie with binary egde labels, where each root to leaf path spells a different code word in the Huffman code table.

3.2 Elias codes

To code a sequence of (large) integers, one can use *self-delimiting* integer codes like Elias γ- and δ-coding [Eli75]. For integer $x = x_{10}$ taken in its binary expression x_2, we have $\gamma(x) = 0^{|x_2|-1}1x_2$. E.g. $\gamma(5) = 001101$, since $5_{10} = 101_2$. To decode $\gamma(x)$, one can read zeros until encountering the first one to reveal the length y of x_2, and then the y next bits contain $x_2 = x_{10}$. Hence, a sequence of γ-coded integers can be decoded uniquely. Notice that $\gamma(x)$ occupies at most 2 times the number of bits required for binary expression of x. To have asymptotically smaller codes, the idea can be applied recursively one step further to obtain δ-code: $\delta(x) = 0^{|O^{|x_2|-1}|-1}1(|x_2|-1)_2x_2$. E.g. $\delta(5) = 0^1 1 2_2 5_2 = 0110101$.

3.3 Bit-vector operations *rank* and *select*

The following operations on bit-vector $B[1 \ldots n]$ are key components of many more complex compressed data structures:

- $rank(B, i)$ tells how many 1-bits there is up to position i in B, and
- $select(B, i)$ tells which position contains the i-th 1-bit.

$rank$-operation in constant time:

Storing all values $rank(B, i)$ would take $O(n \log n)$ bits, where $n = |B|$.

Partial solution 1: Let us store each ℓ-th $rank(B, i)$ as is and scan the rest of the bits (at most ℓ), during the query. We then have an array $first$, where $first[i/\ell] = rank(B, i)$ when $i \ mod \ \ell = 0$ ($/$ is here integer division). If we choose $\ell = (\lceil \log n \rceil)^2$, we need (about) $n \log n/(\log^2 n) = n/(\log n)$ bits space for the array $first$. We can answer $rank(B, i)$ in $O(\log^2 n)$ time: $rank(B, i) = first[i/\ell] + rank(B, \ell*(i/\ell)+1, i)$, where $rank(B, i', i)$ computes the amount of 1-bits in the range $B[i' \ldots i]$.

Partial solution 2: Let us store more answers. We store inside each area of length ℓ answers for each k-th position (how many 1-bits from the start of the are). We obtain an array $second$, where $second[i/k] = rank(B, \ell * (i/\ell) + 1, i)$, when $i \ mod \ k = 0$. This uses overall space $n \log \ell/k$ bits. Choosing $k = \lceil \log n \rceil$ gives $O(n \log\log n/(\log n))$ bits space usage. Now we can answer $rank(B, i)$ in $O(\log n)$ time, as $rank(B, i) = first[i/\ell] + second[i/k] + rank(B, k * (i/k) + 1, i)$.

Final solution: We use so-called *four Russians Trick* to improve the $O(\log n)$ query time into constant. This is based on an observation that there are only $\sqrt{n}$ bit-vectors of length $k/2 = \lceil \log n \rceil/2$. We store for each position j in each of the $(\log n)/2$ size bit-vector C a value $rank(C, j)$ as is. This takes overall $O(\sqrt{n}\log n \log\log n)$ bits. Let a table $third[0\ldots\sqrt{n}-1][0\ldots\lceil\log n\rceil/2-1]$ store the above values, where the first index equals C as an integer. Let c_i and d_i be the first and second half of the bit-vector $rank(B,k*(i/k),i)$ as $k/2$ bit integers (one has to zero the first bit of c_i, because that bit is already taken into account in the table $first$). We obtain the final formula to compute $rank(B,i)$

$$\begin{aligned} rank(B,i) &= first[i/\ell] + second[i/k] \\ &\quad + third[c_i][\min(i \ mod \ k, k/2-1] \\ &\quad + third[d_i][\max((i \ mod \ k)-k/2,-1)], \end{aligned} \qquad (3.2)$$

where $third[d_i][-1]=0$. Integers c_i and d_i can be read in constant time from the bit-vector B, if the model of computation is chosen properly (RAM-model, where $w=\Omega(\log n)$). E.g. using C-language, B can be represented as an array `unsigned B[n/32+1]`. Then each c_i and d_i can be read from the bit-vectors representing integers `B[i/32]` and `B[i/32+1]` (Exercise: write a C-program that reads c_i and d_i in constant time).

Theorem 3.3.1 *Bit-vector rank operation for a given bit-vector $B[1\ldots n]$ can be supported in constant time on a RAM-model when the size of the computer word is $w=\Omega(\log n)$. In addition to the bit-vector B, one needs a dictionary of size $o(n)$ to support the operation.*

***select*-operation in constant time:**

Notice that *select* can be implemented in $O(\log n)$ time by making a binary search on the *rank*-dictionary. Constant time solution is possible using techniques like above [Mun96, Cla96]:

Theorem 3.3.2 *Bit-vector select operation for a given bit-vector $B[1\ldots n]$ can be supported in constant time on a RAM-model when the size of the computer word is $w=\Omega(\log n)$. In addition to the bit-vector B, one needs a dictionary of size $o(n)$ to support the operation.*

3.4 Wavelet trees

Wavelet tree generalizes *rank* and *select* queries to sequences from any alphabet size [GGV03]. Let us denote by $rank_s(T,i)$ the count of symbols s upto position i in T, and by $select_s(T,j)$ the position of the j-th s in T. Here

$T \in \Sigma^*$. We will next show several ways to reduce the problem of *rank/select* computation on general sequences to computation on binary sequences.

3.4.1 Linear representation

Let us represent a string $T[1 \ldots n] \in \Sigma^*$ as σ binary strings $B_s[1 \ldots n]$ for all $s \in \Sigma$, such that $B_s[i] = 1$ if $T[i] = s$ otherwise $B_s[i] = 0$. Now, $rank_s(T, i) = rank(B_s, i)$ and $select_s(T, j) = select(B_s, j)$. After preprocessing binary strings B_s for *rank/select* queries, we can answer $rank_s(T, i)$ and $select_s(T, j)$ in constant time using $\sigma n(1 + o(1))$ bits of space.

3.4.2 Balanced representation

Consider a perfectly balanced binary tree where each node corresponds to a subset of the alphabet. The children of each node partition the node subset into two. A bitmap B_v at the node v indicates to which children does each sequence position belong. Each child then handles the subsequence of the parent's sequence corresponding to its alphabet subset. The root of the tree handles the sequence $T[1 \ldots n]$. The leaves of the tree handle single alphabet symbols and require no space.

To answer query $rank_s(T, i)$, we first determine to which branch of the root does s belong. If it belongs to the left, then we recursively continue at the left subtree with $i \leftarrow rank_0(B_{root}, i)$. Otherwise we recursively continue at the right subtree with $i \leftarrow rank_1(B_{root}, i)$. The value reached by i when we arrive at the leaf that corresponds to s is $rank_s(T, i)$.

The character t_i at position i is obtained similarly, this time going left or right depending on whether $B_v[i] = 0$ or 1 at each level, and finding out which leaf we arrived at.

Query $select_s(T, j)$ is answered by traversing the tree bottom-up.

The above hierarchical structure is called *wavelet tree* [GGV03]. The size of the structure is $n \log \sigma(1 + o(1))$ bits. Wavelet tree supports *rank/select* queries in $O(\log \sigma)$ time.

3.4.3 Huffman-shaped representation

Let the alphabet symbol probabilities be defined by the relative symbol frequencies in T. The *Huffman-shaped wavelet tree* is a wavelet tree, where the underlying balanced binary tree is replaced by the Huffman tree of the given empirical probability distribution. It is easy to see that the space-requirement of Huffman-shaped wavelet tree is $n(H(T) + 1)(1 + o(1))$.

3.5 Burrows-Wheeler transformation

Burrows-Wheeler transformation (BWT) [BW94] offers an efficient way to achieve high-order compression. The idea is simple: The text is transformed into a better compressible form. BWT works as follows:

- Construct the n *cyclic shifts* $t_1 t_2 \cdots t_n$, $t_2 t_3 \cdots t_n t_1$, …, $t_n t_1 \cdots t_{n-1}$ of a string $T = t_1 \cdots t_n$ and sort them into the lexicographic order.
- Let the string be `mississippi`. After sorting the cyclic shifts, we get a matrix M.

```
1  imississipp
2  ippimississ
3  issippimiss
4  ississippim
5  mississippi
6  pimississip
7  ppimississi
8  sippimissis
9  sissippimis
10 ssippimissi
11 ssissippimi
```

- The transform is the last column L of M, `pssmipissii`, and row number, 5, where the original string appears in M. Notice that the transformed text is a permutation of the original.

The transform $BWT(T)$ is reversible, i.e., one can obtain T from $BWT(T)$. It is enough to be able to (virtually) walk through the rows of M in the order of right-shifts of T: `5 mississippi`, `1 imississipp`, `6 pimississip`, … , `4 ississippim`. Once this walk order is found, the original text is revealed backwards: $L[5] =$`i`, $L[1] =$`p`, $L[6] =$`p`, …, $L[4] =$`m`. To find this walk just based on the last column L may seem difficult, but in fact it remains to observe that one can just follow where the revealed symbol maps in the first column of M, which is well-defined as the i-th occurrence of symbol s in L becomes the i-th occurrence in the first column F of M: To see this, let Xs and Ys be two rows of M ending with symbol s and $X < Y$, where $<$ denotes lexicographic order of strings X and Y. Now, sX and sY are right-shifts of Xs and Ys and clearly $sX < sY$, that is, the relative order of all occurrences of s must be the same in F and L, proving the claim. The mapping from k-th occurrence of symbol s in L to its k-th occurrence in F is called *LF-mapping*. It is easy to compute by stable sorting L and recording where the symbols map in the sorted order.

With the LF-mapping, the algorithm to reverse the Burrows-Wheeler transform can be stated as follows. Set i the stored row number. Repeat n times:

- Read $L[i]$. Set $i := LF(i)$.

With our example, this gives:

```
i    5 1 6 7 2 8 10 3 9 11 4
     i p p i s s i  s s i  m
```

3.5.1 Construction

The construction of the Burrows-Wheeler transform requires an efficient algorithm to sort the cyclic shifts of the text. Sorting the cyclic shifts is essentially equivalent to sorting the suffixes of the text. In fact, in our example the order becomes the same:

```
1  i
2  ippi
3  issippi
4  ississippi
5  mississippi
6  pi
7  ppi
8  sippi
9  sissippi
10 ssippi
11 ssissippi
```

In general, one can add an endmarker to the end of the text. This special symbol is considered to be smaller than any other alphabet symbol. Then, sorting cyclic shifts is the same as sorting suffixes. The loss in compressibility due to the endmarker is negligible. From now on we assume there is a virtual endmarker at the end of the text. The sorted order of the suffixes is called *suffix array*. Notice that it is enough to store an array of integers denoting the starting positions of the suffixes:

```
i  Pos      m i s s i s s i p p  i
------      1 2 3 4 5 6 7 8 9 10 11
1  11
2  8
3  5
4  2
5  1
6  10
7  9
8  7
9  4
10 6
11 3
```

It is possible to construct the suffix array in $O(n)$ time on a text of length n [KSB06, KSPP05, KA05]. We omit the details here.

3.6 Compressed suffix array

We will next see how to exploit the Burrows-Wheeler transform and wavelet trees to develop a compressed suffix array. See [NM07] for a survey on the topic.

Recall the suffix array $Pos[1 \ldots n]$. Given $Pos[1 \ldots n]$, the occurrences of the pattern $P = p_1 p_2 \ldots p_m$ can be counted in $O(m \log n)$ time: The occurrences form an interval $Pos[sp, ep]$ such that suffixes $t_{Pos[i]} t_{Pos[i]+1} \ldots t_n$, for all $sp \leq i \leq ep$, contain the pattern P as a prefix. This interval can be searched for using two binary searches in time $O(m \log n)$ [MM93]. Once the interval is obtained, the starting positions of the *occ* occurrences can be listed in $O(occ)$ time.

In the following, we develop a mechanism to find the interval $[sp, ep]$ matching a given pattern without requiring the full storage array $Pos[1 \ldots n]$.

3.6.1 Succinct LF-mapping.

Let L be the Burrows-Wheeler transformed text, i.e. L is the last column of matrix M, and let F be the sorted order of symbols of the text, i.e. F is the first column of M. Let the k-th occurrence of some symbol s in L be at row i. Recall that we defined $LF(i) = j$ as the mapping of k-th occurrence of symbol $s = L[i]$ to its k-th occurrence $F[j]$ in F.

Let us consider how to succinctly represent function $LF()$, since it is a central part of compressed full-text indexes. Instead of precomputing LF-values into a table, we can rewrite LF-mapping in the form $LF(i) = C[L[i]] + k$, where the array $C[1, \sigma]$ stores in $C[s]$ the number of occurrences of characters $\{\$, 1, \ldots, s-1\}$ in the text T. Notice that $C[s]+1$ is the position of the first occurrence of s in F (if any). To compute the value of k, we can exploit wavelet tree for L: $k = rank_{L[i]}(L, i)$. Hence we have the identity $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$.

3.6.2 Backward search.

We exploit the connection of suffix array and Burrows-Wheeler transform to derive a pattern search algorithm that works in $O(m)$ steps [FM00, FM05]. The algorithm for counting the pattern occurrences is shown below.

```
**Algorithm** Count($P[1 \ldots m]$,$L[1 \ldots n]$)
(1)  $i \leftarrow m$;
(2)  $sp \leftarrow 1$; $ep \leftarrow n$;
(3)  while ($sp \leq ep$) and ($i \geq 1$) do
(4)      $s \leftarrow P[i]$;
(5)      $sp \leftarrow C[s] + rank_s(L, sp-1)+1$;
(6)      $ep \leftarrow C[s] + rank_s(L, ep)$;
(7)      $i \leftarrow i - 1$;
(8)  if ($ep < sp$) then return "not found"
     else return "found ($ep - sp + 1$) occurrences".
```

The correctness of the above algorithm is easy to see by induction: At each phase i $[sp, ep]$ gives the maximal interval of suffix array Pos pointing to suffixes prefixed by $P[i \ldots m]$. Figure 3.1 gives an example.

The time requirement of the $Count()$ algorithm is clearly $O(m)$ if function $rank_s()$ can be computed in constant time. Using the results from Sect. 3.4, we can support $rank_s()$ in constant time using $\sigma n(1 + o(1))$ bits of space, OR we can support $rank_s()$ in $O(\log \sigma)$ time using $n \log \sigma(1 + o(1))$ bits of space. Other space-time tradeoffs appear in the literature.

3.6.3 Self-indexing.

So far we can only support counting queries. We are not able to locate the occurrence positions. One way to do this is to *sample* suffix array values

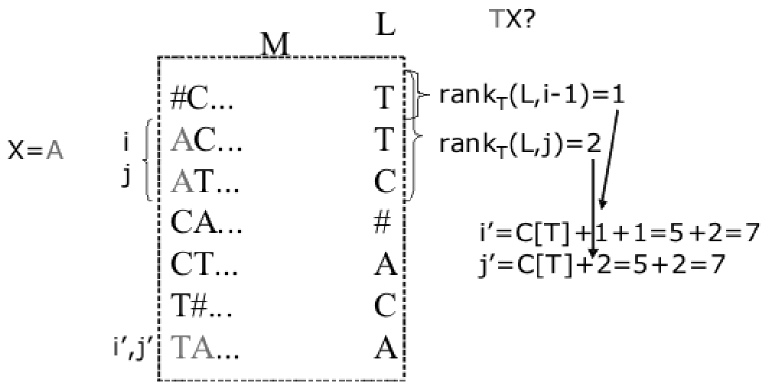


Figure 3.1: Backward search step on the Burrows-Wheeler transform of $S =$ `CATACT#`.

and retrieve the rest using LF-mapping. Adjusting the sample rate gives different space/time tradeoffs. The details follow.

Let r be the sample rate. We sample values $Pos[i]$ such that $Pos[i] = rk$ for $0 \leq k \leq n/r$. We need a bitvector $B[1, n]$ to mark the positions that are sampled, i.e., $B[i] = 1$ if $Pos[i] = rk$ otherwise $B[i] = 0$. Then the samples can be stored in an array $SSA[0, n/r]$: $SSA[rank(B, i) - 1] = Pos[i]$ for $Pos[i] = rk$.

Now, if $B[i] = 1$, then $Pos[i] = SSA[rank(B, i)]$ restores the sampled suffix array value. If $B[i] = 0$, one can apply $j = LF(j) = C[L[j]] + rank_{L[j]}(L, j)$, say d times, until $B[j] = 1$ holds, where $j = i$ in the beginning. Then $Pos[i] = d + SSA[rank(B, j)]$. Here $L[1, n]$ is the same Burrows-Wheeler transform as used for supporting counting queries.

To retrieve one value $Pos[i]$ takes $O(r \log \sigma)$ time, since each of the $d \leq r$ steps requires one $rank$-computation (via wavelet tree) on the Burrows-Wheeler transform L.

One can choose $r = \log_\sigma^{1+\epsilon} n$ for any given $\epsilon > 0$ to have the samples in $SSA[]$ fit in $(n/r) \log n = n \log \sigma / \log^\epsilon n = o(n \log \sigma)$ bits, which is of the same order as required for counting queries. This setting enables retrieval of $Pos[i]$ in time $O(\log^{1+\epsilon} n)$.

Bitvector B and its $rank$-structure require $n + o(n)$ bits if implemented as described in Sect. 3.3. However, since B is sparse (with r bits set), it can be run-length compressed into $O(r \log(n/r) = o(n)$ bits using Elias codes [Eli75]. It is easy to extend the $rank$-solution to such a compressed representation [MN07] (use the same two-level mechanism to store pointers into the compressed representation as the partial $rank$ answers). Alternatively, one can compress B with the identifier coding [Pag99, RRR02] for which

$rank$-structures are also easy to store.

The resulting structure is called *compressed suffix array*. (Other implementations for compressed suffix array exist; in fact, the original proposal [GV06] uses quite a different mechanism for compression and achieves slightly better running time for retrieval of suffix array values.)

Our compressed suffix array can be further developed into a *self-index*. A self-index replaces the text with a compressed representation, so that the text itself can be discarded. For this functionality, we should be able to access the text substrings efficiently. Notice, that we can extract the whole text from the (wavelet tree of the) Burrows-Wheeler transform using the reverse transformation as explained earlier. To access arbitrary substring efficiently, we can exploit sampling once more. It is enough to store inverse sampled suffix array values in an array $ISSA[0, n/r]$: $ISSA[k] = i$ for $Pos[i] = rk$. Then, given a text interval $[e, f]$, $i = ISSA[k]$ for smallest $rk \geq f$ gives us the suffix array index containing pointer to suffix $T_{rk,n}$. Applying LF-mapping starting from i analogously as in extracting the whole text backwards, reveals the substring $T_{e,rk}$ backwards and hence $T_{e,f}$ as its prefix. Running time is $O((f - e + r) \log \sigma)$ and the space is the same as for array SSA.

3.7 Compressed suffix trees

Suffix tree is a classical full-text index, that extends suffix arrays with several useful functionalities. It is a tree whose leaves are the suffixes of the text, and each path from root to a leaf spells the corresponding suffix. The edges are labeled with pointers to text substrings. The edge labels are of maximal length so that the tree branches on each internal node. Hence, with n leaves, the tree has at most $n-1$ internal nodes. Each node and edge stores constant number of values / pointers that each can be represented using $\log n$ bits. A concept often required in the construction and applications of suffix tree is *suffix link*: Let aX be the substring (prefix of suffix) leading to node v and let X be the substring leading to node w. Then we say that there is suffix link from v to w, denoted $sl(v) = w$. Another useful notion is the *string depth* of a node w, defined as the length $|X|$ of the string X leading to w. Figure 3.2 illustrates the definitions.

For more details on basic concepts related to suffix tree, see e.g. [Gus97].

A careful implementation of suffix tree requires $(3x+2n) \log n$ bits, where $x < n$ is the number internal nodes. This is without suffix links, parent pointers, auxiliary information, etc. In many practical implementations, $\log n$ is the computer word size (32 or 64 bits).

For ≈ 3 gigabases sequence of Human Genome, which can be represented

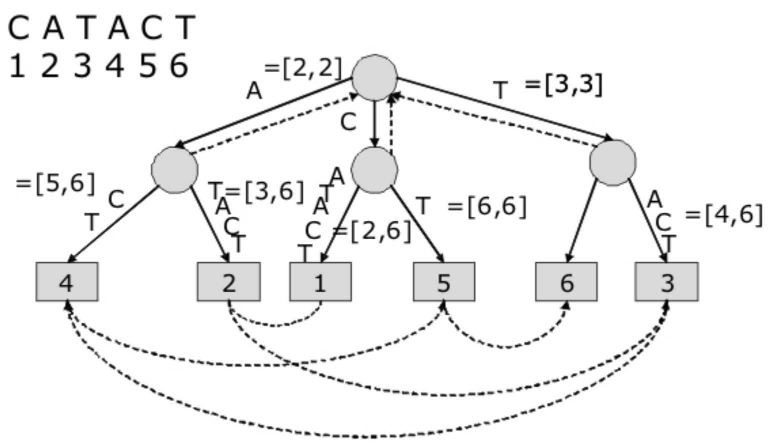


Figure 3.2: Suffix tree of S = `CATACT`. Dashed arrows denote suffix links. Numbers in brackets are pointers to text substring giving the edge labels. The edge with no label denotes the end of the sequence; in practice, a specific endmarker is appended for this purpose.

in 6 gigabits using 2 bits per base, the difference in space-requirement is more than 65-fold, assuming $x = 0.7n$ as shown to hold for DNA sequences [PZ07]. While for many applications it is enough to have suffix tree built for some collections of short sequences, there are tasks that require the tree to be built for the complete sequence collection. We will see some examples in the sequel. Therefore it is of interest to study whether it is possible to reduce the 65-fold difference.

3.7.1 Sadakane's structure.

Sadakane [Sad07] has proposed a mechanism to simulate suffix tree by several compressed data structures. The idea is to split suffix tree into different logical units that can be compressed independently. These logical units are (i) leaves, (ii) tree hierarchy, (iii) edge labels, and (iv) suffix links. For (i) one can exploit the connection to suffix array; In suffix tree the leaves can be represented in arbitrary order, i.e., also in the lexicographic order of the suffixes they represent. Once this order is fixed (as we assume in the sequel), the leaves can be replaced by any implementation of compressed suffix array. For (ii) one can use the balanced parentheses sequence. For (iii) and (iv) the solutions are new compressed data structures for *longest common prefix information*, *range minimum queries* and *lowest common ancestor queries*, explored next.

Longest common prefix information is well-known associate of suffix array; it is defined by an array $LCP[2, n]$ with $LCP[i] = |lcp(T_{Pos[i],n}, T_{Pos[i-1],n})|$, where $lcp(X, Y)$ denotes the longest prefix common to X and Y. The lowest common ancestor -query $c = lca(v, w)$ on a tree returns the common ancestor c of given two nodes v and w such that c resides lowest in the tree. There is an interesting connection between LCP and lca on a suffix tree (assuming lexicographic order of suffixes in leaves): Let v and w be two leaves of suffix tree, given as integers such that $Pos[v]$ and $Pos[w]$ give the suffix positions, then $c = lca(v, w)$ is the node reached after following length $RMQ_{LCP}(v+1, w) \min\{LCP[v+1], LCP[v+1], \ldots, LCP[w]\}$ prefix of the path label from root to leaf v (or equivalently to leaf w). It is easy to see that each internal node of suffix tree corresponds to some *range minimum query* $RMQ(i, j)$ on LCP array. With the interplay between balanced parentheses, RMQ, and LCP, it is possible to efficiently extract the edge labels in suffix tree when requested (see the details in [Sad07]).

Finally, suffix links can be simulated using the above structures as well. Suffix link on a leaf node is in fact an inverse function of LF-mapping, whose computation is easy to include in the compressed suffix array. On an internal node, suffix link computation uses a transitivity property. Let $c = lca(v, w)$, $sl(c) = c'$, $sl(v) = v'$, and $sl(w) = w'$. Then $lca(v', w') = c'$. Given the position of c in the balanced parentheses sequence, the positions of v and w can be revealed with constant time operations on it. Then it is enough to compute v' and w' from v and w using compressed suffix array, and then use RMQ on LCP to reveal the string depth of c' and again balanced parentheses operations to find out the position of c' in it.

The time-requirement of operations sketched above depend on the underlying data structures. Using the compressed suffix array described earlier and the new data structures in [Sad07], the maximum running time on any suffix tree operation is $O(\log^{1+\epsilon} n)$.

The space-requirement is $6n + sizeof(CSA) + o(n)$ bits, where $sizeof(CSA)$ is the space-requirement of the underlying compressed suffix array. The $6n$ consist of $2n$ bits from the balanced parentheses, $2n$ from compressed LCP, and $2n$ from range minimum query structure. The $o(n)$ comes from the sublinear data structures associated with the representations (much alike the earlier $rank$-structure).

The first implementation of Sadakane's compressed suffix tree can be found in `http://www.cs.helsinki.fi/group/suds/cst`. The space-requirement is 8.8 GB in that case, i.e., about 12-times larger than the compressed genome sequence. Yet, this is over 5-times less than a normal suffix tree, and actually closer to 10-times less than a suffix tree with equivalent functionality; the compressed suffix tree has in addition to standard functions, access to parent, computation of subtree size, number of leaves in subtree, suffix links, *lca*, possibility to attach auxiliary data in compressed form. These extended functionalities are in fact crucial to most advanced algorithms exploiting suffix tree [Gus97].

There are alternative proposals for compressed suffix trees providing different time/space tradeoffs and dynamic updates [RNO08b, RNO08a, FMN08].

Many compressed suffix tree variants have been recently implemented by Simon Gog, and can be found inside a generic succinct data structure library `http://www.uni-ulm.de/en/in/institute-of-theoretical-computer-science/research/sdsl.html`.

Chapter 4

High-throughput sequence mapping and analysis

4.1 Sequence mapping

Sequence mapping is a fundamental primitive required in many sequence-related studies. For example, the recent *ChIP-sequencing* technology can be used to produce large number of short DNA sequences tagging protein binding sites [JMMW07]. The underlying mapping problem can be stated essentially as a *multiple pattern matching* problem: Given a set $\mathbb{P}$ of short sequences (patterns extracted from protein binding sites), find their occurrence positions in a long sequence T (the whole genome). In the best case, the occurrence is exact (pattern matches a substring of T), but typically some errors must be allowed (pattern matches approximately a substring of T).

Figure 4.1 shows the different application areas of high-throughput (next-generation) sequencing. In addition to ChIP-sequencing discussed above, one can sequence complementary DNA of RNA trancsripts using *RNA-sequencing*. There the problem is that reads should be mapped to genome allowing an intron to split the read; we will discuss this case separately. Other applications are *targeted resequencing* and *whole genome resequencing*, where in the former some areas of the genome are spliced out, enriched, and then sequenced, and in the latter the whole genome is sequenced. Sequencing can also be applied to genomes whose sequence is not yet known, and then the process is called *de novo sequencing* and the problem to be solved is *fragment assembly*. When the tissue contains several organisms, the task of identifying them is called *metagenomics*. This usually involves both known

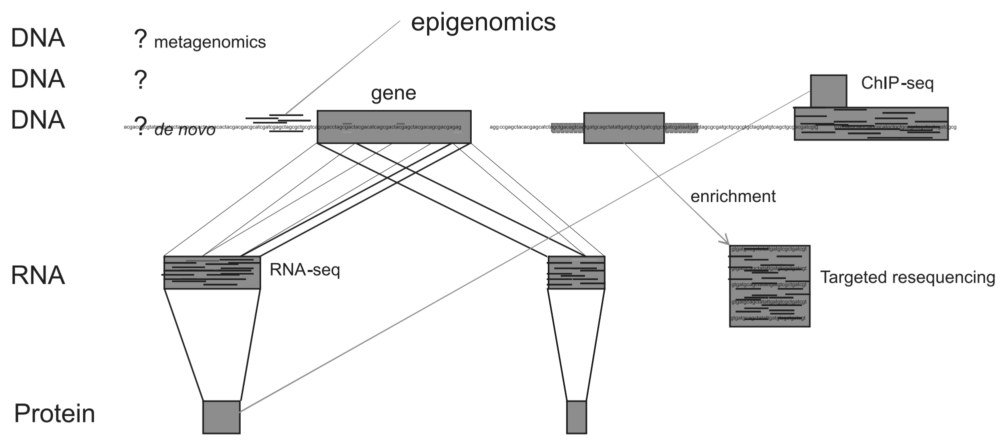


Figure 4.1: Atlas of high-throughput sequencing applications.

and unknown genomes.

Very recently also *epigenomics* studies have joint this atlas. It is possible to measure the chemical changes in DNA that do not change the sequence. Such changes affect gene regulation and are then likely to constitute the main mechanism why e.g. identical twins are not really identical in their phenotype. There is also evidence that these chemical changes are inheritable. One such chemical change is *DNA methylation*, where some cytosines have a modification such that a methyl group is bound to them. In promoter regions of eukaryote genes DNA methylation level is inversely proportional to the transcription activity. There is a sequencing technology called *bisulfite sequencing*, to target the promoter areas (or in fact *CpG islands*) with high methylation levels.

As a summary, ChIP-sequencing, methylation sequencing, and targeted resequencing are quite similar in the analysis perspective; some areas are targeted and the task is to find them based on sequence mapping. RNA sequencing adds the difficulty of split reads. In de novo sequencing, sequence mapping is not an option, so the task is to find overlaps between the reads and construct the consensus sequence based on the observed overlaps.

4.1.1 Read types, significance of hits, filtering

Different sequencing technologies produce different kinds of reads, e.g.:

- `Illumina Solexa`[1] reads are typically of length $35-150$ basepairs (getting longer all time with advances in technology). Each base is asso-

[1] TM Illumina, Inc.

ciated with an encoded probability of the observation being correct. Typical sequencing error is a mutation; indels occur less frequently.

- `SoLID`[2] are similar in length to Solexa reads, but the coding is in *colour-space* with a two-base code defined by the matrix (row=first base, column=second base):

	A	C	G	T
A	0	1	2	3
C	1	0	3	2
G	2	3	0	1
T	3	2	1	0

  Consider the values in the matrix as colours. For example, `T012023211202102` equals `TTGAAGCTGTCCTGGA` (first base is always given). Error probabilities are like in solexa, but the advantage of colour code is that if there is a sequencing error, the whole suffix of the read is incorrect; for a long read with an error in the middle it is unlikely that the read maps to any part of the genome. This separates most measurement errors from SNPs that only affect the code locally (two consecutive colours); a read including a SNP can be mapped to the correct location in the genome allowing basically two errors in the alignment.

- `454 sequencing`[3] reads are about 400 basepairs long. The errors produced by the sequencer are different from the two above; the sequence is constructed as *runs* of symbols mapping light intensity to the length of the run. Raw data files actually keep the observed intensities and sequence mapping could exploit these values (approximate string matching with run-length encoded strings).

Classical `Sanger sequencing` could be included to the picture, providing much longer and more accurate reads, but the advantage of the new technology is simply their much better bases per euro proportion. For the same reason, 454 reads are typically only used in de novo sequencing projects, where the read length is important. In resequencing projects, Solexa and SoLID are the prevailing techniques.

All the three high-throughput techniques generalize to providing *paired end* and *mate pair* reads. Paired end read is a suffix, prefix pair of a fragment of DNA, with read length smaller than the fragment length:

[2]TM Applied Biosystem

[3]TM 454 Life Science, Roche Diagnostics

```
TGACGACTGCTAGCTA..........................AGCTACGTAGCATCGT
```

Mate pair is the same thing but produced with a different sequencing technique. The most notable difference for the analysis perspective is that the fragment for paired end reads is at maximum about 1000 base pairs, as mate pairs can be produced from longer fragments (upto several thousands).

In what follows, we mostly discuss reads of Solexa type; generalizations to other types are left for the reader (see exercises).

First thing to consider when mapping reads is the probability of finding the correct occurrence location. Let $A = a_1a_2\cdots a_m$ be the consensus sequence of a read with an associated sequence of probabilities $p(A) = p_1p_2\cdots p_m$, where $0.25 \leq p_i \leq 1$ tells the probability of i-th position in the read being a_i. Denote $M[c,i] = p_i$ if $a_i = c$ and $M[c,i] = (1-p_i)/(|\Sigma|-1)$ if $c \neq a_i$. Then we have a *positional weight matrix (PWM)* M representing the read. We say that matrix M *occurs* in position j in a genome sequence $T = t_1t_2\cdots t_n$ if $p(M,T,j) = \prod_{i=1}^{m} M[t_{j+i-1}, i] > t$, where t is a predefined threshold. Consider a random T from an i.i.d. model with q_c denoting the probability of symbol c. We wish to compute whether M is expected to occur in the random T. The probability of M matching a random sequence of length m is $\sum_{C=c_1c_2\cdots c_m\in\Sigma^m : p(M,C,1)>t} \prod_{i=1}^{m} q_{c_i}$. By the linearity of expectation, the expected number of occurrences for M in T is

$$E(M,T) = (n-m+1) \sum_{C=c_1c_2\cdots c_m\in\Sigma^m : p(M,C,1)>t} \prod_{i=1}^{m} q_{c_i}. \tag{4.1}$$

We can now set thresholds $t \leq 1$ and $f \leq 1$ such that reads with PWM matrix M having $E(M,T) > f$ can be filtered out.

Computing $E(M,T)$ is problematic because it requires enumeration over all sequences of lengh m. To obtain an easier to compute filter, we use an alternative modeling: We model position i in the read as a_i with probability p_i and as a *joker symbol* $*$ matching any symbol with probability $(1-p_i)$. Then A matches a random sequence of length m with probability $\prod_{i=1}^{m} p_iq_{a_i} + (1-p_i)$, and the expected number of occurrences of A in random sequence $T = t_1t_2\cdots t_n$ is

$$E(A,T) = (n-m+1)\prod_{i=1}^{m} p_iq_{a_i} + (1-p_i). \tag{4.2}$$

Again we can use filtering $E(A,T) > f$, and now the filtering is easy to do in linear time in the read length. Notice that we have on purpose overused here the measurement error probability to obtain as stringent filter as possible; the error includes the case that joker matches a_i. More accurate interpretation would be that joker matches only other symbols but a_i, and then the match probability would be $p_i q_{a_i} + (1 - p_i)(1 - q_{a_i})$.

In the sequel, we assume that one of the filters above, or some more advanced filtering, has been done so that random hits are unlikely. Notice that a read should originate from exactly one location in the donor DNA, but if this location happend to be inside a repeat, we can still expect to see many occurrences when aligning the read to the reference sequence.

4.1.2 Problem modeling and theoretical solutions

Indexed multiple approximate string matching

Typically the problem is modeled as a k mismatch problem or as a k errors problem (see Sect. 2.3), ignoring the probabilities. Using that modeling, the best online solutions are the $O(rmn/w)$ time bitparallel algorithms, where r is the number of reads. One can do much better by using average-optimal *multiple approximate string matching* [FN04], which gives running time $O(rm \log_{|\Sigma|}(rm) + (k + \log_{|\Sigma|}(rm))n/m)$ holding for small k, where the first part comes from preprocessing the read set and second part from matching. However, since the approach builds an index for the read set, the space requirement will be a bottleneck ($O(r|\Sigma|^{\ell})$, where ℓ needs to be set to $O(\Theta(\log_{|\Sigma|})$ to obtain the optimal running time).

The above solutions do not yet exploit the fact that the reference is a static sequence and can be preprocessed into an index structure, to be used with many read sets. Let us first assume that there are no errors in the reads and we are interested only on exact matches (applications without the need of SNP detection). For such indexed exact multiple pattern matching the obvious solution is as follows. Build suffix tree of T, and match each pattern in $\mathbb{P}$ against a path in the tree. Report the occurrence positions lying in the subtree leaves of the found path (if exists). The solution is takes (after building the suffix tree in $O(T)$ time) linear (optimal) time in proportion to the total length of sequences in $\mathbb{P}$ and to the number of occurrences. In practice, binary search on suffix array is usually faster for the same task and much more space-economic. To get more space savings, one can use compressed suffix array of Chapter **??** with some sacrifice in running time.

For *indexed multiple approximate pattern matching* there is no obvious solution. In fact, this problem was the Holy Grail of pattern matching for long time, until Cole, Gottlieb, and Lewenstein [CGL04] finally were able to develop an index with guaranteed running time for approximate searches. However, their index requires superlinear space making it useless for genome-scale sequences. This bottleneck was soon overcome by Chan et al. [CLS$^+$06]; they provide an $O(n)$ words index with running time $O(m+\texttt{occ}+polylog(n))$, where m is the length of the pattern and `occ` is the number of occurrences with upto k-errors. The space can be further decreased to $O(n)$ bits by using the compressed suffix tree as part of the index. However, the result works only for very small values of k, as the $O()$ space terms hide 3^k and $\log^{k^2} n$ factors. In short read mapping, the sequences are quite accurate and allowing only a few errors usually suffice in the search. It would be very interesting to see if a practical implementation of this index would work better than the more heuristic approaches covered next.

4.1.3 BWT-based approaches

One practically efficient solution to indexed approximate multiple pattern matching is the *simulation of backtracking on suffix tree*. This technique is behind the fastest and most memory efficient solutions to sequence mapping up-to-date: `BWT-SW`[4], `BWA`[5], `SOAP2`[6], `Bowtie`[7], and `readaligner`[8].

Let us first study backtracking on suffix tree and then see how it can be simulated using backward search in compressed suffix array. To get the idea, it is enough to consider *k-mismatches* problem, where an occurrence is a substring of T that can be converted into the pattern with at most k substitutions, i.e., the *Hamming distance* of the pattern and its occurrence is at most k.

The backtracking for k-mismatch occurrences of a pattern $P[1,m]$ in $T[1,n]$ works as follows. Make a depth-first traversal to the suffix tree of T upto string depth m. Count the mismatches between P and each path label. Report suffixes in the subtrees whose path labels have at most k mismatches. It is obvious that this algorithm can be sped up by maintaining the current number of mismatches at the nodes, and retracting on paths with more than k mismatches encountered.

The same backtracking idea can be applied for e.g. the k-errors problem by filling in the familiar dynamic programming tables along the backtracking paths (or even faster by running the Myers' bitparallel algorithm instead [Mye99]). It works also for local alignment (Smith-Waterman algorithm)

[4]`http://i.cs.hku.hk/$\sim$ckwong3/bwtsw/`
[5]`http://maq.sourceforge.net/bwa-man.shtml`
[6]`http://soap.genomics.org.cn/index.html`
[7]`http://bowtie-bio.sourceforge.net/`
[8]`http://www.cs.helsinki.fi/group/suds/readaligner/`

[LST+08] and for matching position-specific scoring matrices (PSSMs) (see the SUDS Genome Browser, `http://www.cs.helsinki.fi/group/suds/cst`), or even for read mapping versions of profile HMMs (not covered here). In all cases, backtracking can be replaced by branch-and-bound mechanism; for example in the latter two, the maximum tail probability can be used to bound the best match in the current subtree, limiting the amount of branching.

Backtracking is exponential in the length of the pattern and in the number of errors in the worst case. For small error-levels and errors concentrating on tails (like in read data typically), the branching is in practice limited, and the approach is rather efficient.

Due to its practicality with respect to time efficiency, let us focus on the space issue. The backtracking on suffix tree is easy to simulate using compressed suffix tree, which means that, with the current implementations, 8.5 GB for human genome is sufficient for the task. However, it is possible to use much more simplified mechanism to perform backtracking, as we will learn next.

Recall the Burrows-Wheeler transformed text L and the backward search algorithm using it: At step i of the algorithm $[sp, ep]$ gives the maximal interval of suffix array Pos pointing to suffixes prefixed by $P[i \ldots m]$. To perform backtracking instead of exact search, one can perform a backward search step *with all the alphabets symbols* instead of just with $P[i-1]$. For k-mismatches problem, if the chosen alphabet symbol is other than $P[i-1]$, mismatch counter is incremented for the path (range) taken, otherwise the counter stays the same. The procedure is applied recursively until too many errors have been encountered. The result is a set of suffix array ranges containing the k-mismatches occurrences. With the added suffix array samples to form the compressed suffix array, these occurrences can be reported in $O(\log^{1+\epsilon} n)$ time each. The pseudocode is given below. The first call to the recursive procedure is `kmismatches`$(P, C, L, k, m, 1, n)$.

```
Algorithm kmismatches($P, C, L, k, j, sp, ep$)
(1) if ($j = 0$) then
(2)         Report occurrences $Pos[sp], \ldots, Pos[ep]$; return ;
(3) for each $s \in \Sigma$ do
(4)         $sp' \leftarrow C[s] + rank_s(L, sp-1)+1$;
(5)         $ep' \leftarrow C[s] + rank_s(L, ep)$;
(6)         if ($P[j] \neq s$) then $k' \leftarrow k-1$; else $k' \leftarrow k$;
(7)         if ($k' \geq 0$) kmismatches($P, C, L, k', j-1, sp', ep'$);
```

Just like with (compressed) suffix tree, the above backtracking / branch-

and-bound mechanism can be applied to many other forms of approximate matching. For example, any dynamic programming algorithm for alignment can be one transformed to build one column of the dynamic programming table on each recursive step. The following pseudocode illustrates the idea:

```
Algorithm kerrors($sp, ep, count, oldcol$)
(1)  if ($sp > ep$) return ;
(2)  if ($oldcol[patlen] \leq klimit$) then
(3)      Report occurrences $Pos[sp], \ldots, Pos[ep]$; return ;
(4)  for each $s \in \Sigma$ do
(5)      $sp' \leftarrow C[s] + rank_s(L, sp-1)+1$;
(6)      $ep' \leftarrow C[s] + rank_s(L, ep)$;
(7)      $curcol[0] \leftarrow count$; $minval \leftarrow count$;
(8)      for $i = 1$ to $count$ do
(9)          if ($P[patlen - count + 1] = s$) then $diag \leftarrow 0$; else $diag \leftarrow 1$;
(10)         $curcol[i] \leftarrow min(oldcol[i] + 1, curcol[i-1] + 1, oldcol[i-1] + diag)$;
(11)         if ($curcol[i] < minval$) then $minval \leftarrow curcol[i]$;
(12)     if ($minval \leq klimit$) kerrors($sp', ep', count+1, curcol$);
```

This procedure is first called with `kerrors`$(1, n, 1, firstcol)$ where $firstcol[i] = i$ (and we have dropped the static variables from the call). The arrays *oldcol* and *curcol* contain values of two consecutive columns of the dynamic programming table. The minimum value of the current column (*minval*) is calculated to terminate the search if no further matches are possible.

To get an idea of the practical performance, let us consider some experiments on a compressed suffix array closely analogous to the one described in Sect. 3.6. The index requires 2.1 GB for human genome. It performs Algorithm `kmismatches()` (limited to counting the number of occurrences) in 0.3,8.2, and 121 milliseconds on average for parameters $k = 0$, $k = 1$, and $k = 2$, respectively, on a query of length 32. Locating one (e.g. the best) occurrence takes 0.9 milliseconds. The test was run on a random set of 10000 substrings of the genome on a 3.0 GHz Intel Xeon CPU with 128 GB of main memory.

Branch-and-bound technique of `BWA`

The basic backtracking mechanism we studied is not efficient enough for large error levels. Therefore it is important to look at different search space pruning techniques.

Li and Durbin [LD09] use a branch-and-bound technique to limit the search space. To implement the approach, they need FM-index also for the reverse text $T^r = t_n t_{n-1} \cdots t_1$. Let us call *forward FM-index* and *reverse FM-index* the FM-index of T and FM-index of T^r, respectively.

Then they precompute for each prefix α of the pattern, its splitting to maximum number of pieces such that no piece occurs in the text. Let us denote the maximum number of splits $\kappa(\alpha)$. The computation of $\kappa(\alpha)$ for all prefixes α is analogous to the backward search algorithm applied to reverse pattern on reverse FM-index, and hence works in linear number of steps in the pattern length [LD09]: Backward search on reverse FM-index is applied as long as the interval $[sp, ep]$ gets empty. Then the process is repeated with the remaining suffix of the pattern until the whole pattern is processes. In detail, if backward step from $P[1, i]$ to $P[1, i + 1]$ results into non-empty interval $[sp, ep]$ with $sp \leq ep$, then $\kappa(P[1, i + 1]) = \kappa(P[1, i])$. Otherwise, $\kappa(P[1, i + 1]) = \kappa(P[1, i])$ and $\kappa(P[1, i + 2]) = \kappa(P[1, i + 1]) + 1$ and the backward search is started from beginning with pattern $P[i + 2, m]$, and so on.

The computation is also possible to do with forward FM-index alone by simulating the suffix array binary search (with roughly logarithmic slowdown to the linear preprocessing).

Value $\kappa(\alpha)$ works as a lower bound for the number of errors that must be allowed in any approximate match for the prefix α (see exercise for proof). This estimate can be used to prune the search space of backward backtracking as follows. Each search state knows the number of errors, say $\eta(\beta, sp, ep)$, between a suffix β of the pattern and the longest common prefix of suffixes $Pos[sp \ldots ep]$; if $\kappa(\alpha) + \eta(\beta, sp, ep) > k$, the branch can be ignored.

Case analysis technique of `Bowtie`

Langmead et al. [LTPS09] extend the standard filtering technique of splitting pattern into $k + 1$ pieces [Nav01]: The original idea of pattern splitting is to be able to search each piece exactly, since k-errors/-mismatches cannot affect all pieces simultaneosly. Then the surrounding of each exact occurrence of each piece is checked for possible k-errors/-mismatches match. This filter is trivial to implement using exact search in FM-index, but for large error levels too many candidate matches need to be checked.

The extension proposed in [LTPS09] is to consider separately all cases how k mismatches can be distributed in the $k+1$ pieces, and perform backtracking for the whole pattern for each case either from forward or from reverse FM-index, depending on which one is likely to prune better the search space. Too see how it works, let us consider the simplest case $k = 1$ first. Pattern P is split into two pieces $P = \alpha\beta$. One error can be either (a) in α or (b) in β. In case (a), it is preferable to search for $P = \alpha\beta$ using backward backtracking on the forward FM-index, since β must appear exactly and branching is only needed after reading the $|\beta|$ first symbols. In case (b), it is affordable to search for $P^r = \beta^r\alpha^r$ using backward backtracking on the reverse FM-index, since α^r must appear exactly and branching is only needed after reading the $|\alpha|$ first symbols. For obvious reasons $|\alpha| \approx |\beta|$ is a good choice for pruning efficiency. Let us then consider $k = 2$ to see the limitations of the approach. The different ways to distribute two errors into three pieces are (a) 002, (b) 020, (c) 200, (d) 011, (e) 101, and (f) 110. Obviously in cases (a) and (d) it makes sense to use backtracking on the reverse FM-index and in cases (c) and (f) backtracking on the forward FM-index. For cases (b) and (e) either choice is as good or bad. Obviously for any k, there is always the bad case where both ends have at least $k/2$ errors. Hence, there is no strategy to start the backtracking with 0 errors, other than in case $k = 1$.

Two-way BWT approach of SOAP2

Li et al. [LYL+09] solve the bottleneck of the case analysis technique: They develop *Two-way BWT* method such that the search using forward and reverse FM-index can be interleaved. For example, on case 101 above, they are able to start the search, say, with forward FM-index searching blocks $\underbrace{10}_{\leftarrow}1$, and then continue the search from reverse FM-index with block $10\underbrace{1}_{\rightarrow}$.

In detail, the technique works as follows. Let $[sp(\alpha), ep(\alpha)]$ denote the interval of suffix array (or Burrows-Wheeler transform) of T matching α, and let $[sp'(\alpha^r), ep'(\alpha^r)]$ denote the interval of suffix array (or Burrows-Wheeler transform) of T^r matching α^r. Notice that it holds $ep(\alpha) - sp(\alpha) = ep'(\alpha^r) - sp'(\alpha^r)$. Also it holds $ep(c\alpha) - sp(c\alpha) = ep'((c\alpha)^r) - sp'((c\alpha)^r)$ for any $c \in \Sigma$. The goal is to compute interval $[sp'((c\alpha)^r), ep'((c\alpha)^r)]$ given $[sp'(\alpha^r), ep'(\alpha^r)]$ and $[sp(c\alpha), ep(c\alpha)]$, where the latter term is easy to compute from $[sp(\alpha), ep(\alpha)]$ with one backward search step with c on forward FM-index of T. The trick is to compute $[sp(c'\alpha), ep'(c'\alpha)]$ for *all* $c' \in \Sigma, c' \leq c$. Because of the lexicographic order used for constructing the Burrows-Wheeler transform, it must hold

$$\begin{aligned} sp'((c\alpha)^r) &= sp'((\alpha)^r) + \sum_{c'<c} ep(c'\alpha) - sp(c'\alpha) \\ ep'((c\alpha)^r) &= sp'((c\alpha)^r) + ep(c\alpha) - sp(c\alpha). \end{aligned} \tag{4.3}$$

Hence, it costs a $|\Sigma|$ multiplicative factor to maintain both forward and reverse FM-index simultaneously updated. Moreover, Eq. (4.3) can be implemented also in $O(\log|\Sigma|)$ if the FM-index uses wavelet trees; it is possible to add less-than counting to wavelet tree as an extension of *rank*-function.

Now it is easy to see how the two-way approximate search works in `SOAP2`: Once a state $[sp, ep]$ is found matching a prefix of the pattern with the fixed distribution of errors, one can continue the search from the original start position with the reverse FM-index with the interval $[sp', ep']$ that has been maintained in each step of the backtracking search with forward FM-index.

Suffix filter and overlap matching

Kärkkäinen and Na [KN07] developed an extension of the pattern splitting filter. Instead of splitting the pattern into pieces to be searched for exactly, the suffixes starting from the start positions of the pieces are considered. More concretely, let pattern P be partitioned into pieces $P = \alpha_1\alpha_2\cdots\alpha_{k+1}$, then the set of suffixes considered is $\mathcal{S} = \{\alpha_1\alpha_2\cdots\alpha_{k+1}, \alpha_2\alpha_3\cdots\alpha_{k+1}, \ldots, \alpha_{k+1}\}$. Then each $S \in \mathcal{S}$ is searched for from the text so that zero errors are allowed before reaching the end if first piece in S, one error is allowed before reaching the end of second piece of S, and so on. Obviously this search can be done e.g. using backtracking on FM-index (to be precise, backward backtacking on reverse FM-index in order to backtrack on suffixes). This idea is implemented in `readaligner` tool by [MVLK10]. The benefit of suffix filter compared to other search space pruning methods is that the technique extends easily to finding approximately matching overlaps between strings [VLM10][9]. Such overlap computation is an important precomputation step for de novo fragment assembly tasks.

4.1.4 RNA-sequencing

Recall that RNA transcript alignment to the genome consists of exons and introns. Mapping RNA-sequencing reads to the reference genome is otherwise identical to DNA read mapping except for reads whose prefix overlaps end of one exon and suffix overlaps the start of next exon. A way to go over this problem is to align first all reads that map as a whole. With this initial alignment one can identify the exons. Let $T[j' \ldots j]$ and $T[k' \ldots k]$ be two nearby exons found this way. One can then align all remaining reads to $T[j-m \ldots j]T[k' \ldots k'+m]$, and get coverage of this *splice variant* candidate. To do this efficiently, one can in fact concatenate all such exon pairs into one long sequence (adding a special marker in between the pairs), create FM-index (and reverse FM-index) for the concatenation, and proceed to read

[9]`http://www.cs.helsinki.fi/group/suds/sfo/`

mapping as normally. The result is the coverage counts for all splice variant candidates [TPS09]. This approach has two weaknesses: (1) exon boundaries may not be accurately identifiable from initial alignment, and (2) one cannot afford to take distant splice variant candidates into account as there would be too many pairs to be consider.

One can also approach the splice variant alignment problem by directly extending the backtracking methods studied above [MVLK10]. Again using both forward and reverse FM-index turns out to be crucial: Any read spanning two exons splits into two parts where one is at least of length $m/2$. Hence, given a read $P[1 \ldots m]$, search $P[1 \ldots i]$ from reverse FM-index and $P[i' \ldots m]$ from forward FM-index, where $i \geq m/2$ and $i' \leq m/2$. Consider now an occurrence of $P[1 \ldots i]$ with $k' \leq k$ errors in $T[j' \ldots j]$ found using the approach (where one could limit to occurrences such that $T[j+1 \ldots j+2] = \mathrm{GT}$, e.g. an intron starting dinucleotide). It is sufficient to search for $P[i+1 \ldots m]$ in $T[j \ldots j+\alpha]$, where α is the maximum intron length. For example, using Myers' bitparallel algorithm, this takes $O(\alpha m/w)$ time (again one could speed-up by considering only candidate occurrences preceding intron ending dinucleotide AG). It is also possible to do the same computation inside the backtracking algorithm without repeating the same computation for each occurrence, but this requires a different dynamic programming speed-up technique (*Cartesian tree*) [MVLK10], which we omit here. The problem with the approach is that the search pattern length is not fixed and not all pruning mechanisms studied above work. In fact, only the branch-and-bound of `BWA` extends easily.

To go over the problem of non-fixed pattern length, one could directly search for patterns $P[1 \ldots m/2]$ and $P[m/2 \ldots m]$ using any of the pruning mechanisms. This will not yield exact exon/intron boundary locations, but the extension is easy to do once an occurrence of $P[1 \ldots m/2]$ or $P[m/2 \ldots m]$ is given. Then the search for the remaining part of the pattern in another exon can be done as above.

The key question for the usefulness of this approach is that how many occurrences one is expected to be found for the half of the pattern, and what are the odds for finding the correct occurrence of the tail, say $P[i+1 \ldots m]$? Let us consider exact occurrences. The probability of $P[i \ldots m/2]$ matching a random string with each symbol equally likely in every position is $\left(\frac{1}{|\Sigma|}\right)^{m/2}$. Expected number of occurrences in string of length n is less than $n\left(\frac{1}{|\Sigma|}\right)^{m/2}$. To have this value smaller than 1, m should be at least $2\log_{|\Sigma|} n$ which is 33 in the case of human genome and its reverse complement. The same reasoning gives that to have expected number of occurrences for the tail

$P[i+1\dots m]$ in area of length α, one should have $m-i > \log_{|\Sigma|}\alpha$, which is 9 when $\alpha = 100000$. When allowing errors in the search, the expected number of occurrences grows, but with small error levels reasonable m is still sufficient (see exercises). Nowadays even length 150 reads are provided, and this is clearly sufficient length even when halved in this RNA-sequencing read alignment approach.

Let us finally consider the remaining bottleneck of locating the tail $P[i+1\dots m]$. With high-enough coverage, one could limit to reads that split nearby the midpoint; then $P[i+1\dots m]$ is long enough to be unlikely to have random occurrences. With the above calculations, $m-i > 16$ is enough, but to allow errors in the search, higher limit should be used (see exercises). For curiosity, let us consider if it is possible to find an occurrence of $P[i+1\dots m]$ in a given range $T[j+1\dots j+\alpha]$ any faster than online scanning. This kind of search is known as *position-restricted substring searching* and for example an efficient $O(m+\log\log n)$ solution exist [MN06] for exact pattern matching with the cost of much bigger index structure; it is not known if this problem can be solved efficiently in $o(n\log n)$ bits of space. However, the solution extends to the backtracking / search space pruning techniques we studied above, to support approximate search in an interval. If one is willing to use e.g. $n\log n(1+o(1))$ bits of extra space, then there is a relatively easy-to-implement solution that slows down the backtracking solutions at most by $\log n$ factor (see exercises).

4.1.5 Paired end mapping

Aligning pair end reads is typically done using straighforward intersection approach: map each end separately, sort the occurrence positions, and choose the occurrence pair with span closest to the expectation. This is obviously not time efficient when there are many occurrences for each end separately.

The alternative is to use any of the approaches discussed above for RNA-sequencing. The difference is that now the task is much easier since patterns are fixed. Also the span between the two reads in the pair is more accurately known. Therefore it makes sense to use the following concrete scheme: Let P^1 and P^2 be the paired end read pair. Assume that P^1 has less occurrences in T (the other case is symmetric). For each occurrence position j of P^1, check whether P^2 appears in $T[j+c-\beta\dots j+c+\beta]$, where c is the average span and β is a parameter depending on the variance of the spans. Again, if space permits, one can support the search in $T[j+c-\beta\dots j+c+\beta]$ faster using the position-restricted search techniques [MN06] (see exercises for an easy implementation).

When studying rearrangements in genomes (like somatic mutations caused by tumors), paired end mapping is used in a completely different way. Then one is especially interested in anomalies in paired end mapping. For example P^1 mapping to different chromosome than P^2. For this problem, the approach is just to find all occurrences of P^1 and P^2 separately. When this is done for all pairs, one can analyse the mutual evidence for specific rearrangements.

4.2 Lower level sequence analysis

High-throughput sequence analysis can be separated in two categories: (1) lower level analysis tasks and (2) higher level analysis tasks. First category consists of general properties one can compute from the sequence, which can be an input for the second category tasks. For example, searching for *maximal exact matches (MEMs)* between two sequences can be considered a lower level analysis task. Maximal exact match is a substring in sequence A that also occurs as a substring in sequence B and has the property that it cannot be extended left or right without loosing occurrences in A or B. The amount and length of maximal exact matches can work as a simple similarity measure between sequences, but more often MEMs and their variants are used as a first step of some higher level task, such as whole genome alignment. We will next explore problems in the two categories in the context of high-throughput sequencing. For first category problems compressed data structures are essential since the analysis starts from the sequence level. For second category, the inputs are already smaller as they are outputs of sequence level analysis, and we can concentrate more on the biological aspects.

For sequence mapping we ended up observing that compressed suffix array is enough for the task.

For the more complex sequence analysis tasks, the situation is slightly different, as we will learn next.

4.2.1 Maximal repeats

Let us first consider the *maximal repeats* problem, which is a special case of maximal exact matches problem: Find substrings of A that occur at least twice in A and cannot be extended to the left or right such they would not loose any occurrences. This problem can be solved using suffix tree in many ways. First notice that paths ending at suffix tree nodes correspond to *right-maximal repeats*, i.e., substrings that cannot be extended to the right so that they would not loose any occurrences. Analogously, buiding suffix tree for the reverse of A will give all *left-maximal repeats*. Maximal repeats are then the intersection of right-maximal and left-maximal repeats. There are linear number of both kind of repeats, but finding the intersection in linear time is not quite trivial. However, this basic idea gives us a tool to develop a simple algorithm exploiting two-way BWT: Let $[sp(\alpha), ep(\alpha)]$ denote the interval in suffix array (or Burrows-Wheeler transform) of A matching α, and let $[sp'(\alpha^r), ep'(\alpha^r)]$ denote the interval in suffix array (or Burrows-Wheeler transform) of A^r matching α^r. Let $ep(\alpha) - sp(\alpha) = ep'(\alpha^r) - sp'(\alpha^r) > 1$. If $ep(a\alpha) - sp(a\alpha) = ep(a\alpha) - sp(a\alpha)$ for some $a \in \Sigma$, then α is not left-maximal, otherwise it is. If $ep'((\alpha b)^r) - sp'((\alpha b)^r) = ep'(\alpha^r) - sp'(\alpha^r)$ for some $b \in \Sigma$, then α is not right-maximal, otherwise it is. That is, given $[sp(\alpha), ep(\alpha)]$ and $[sp'(\alpha^r), ep'(\alpha^r)]$, we can test in one backward step in both forward and reverse FM-index whether α is maximal or not! It remains to backtrack all candidates for α and maintain the intervals. The pseudocode of the resulting algorithm is given below.

```
Algorithm maximalRepeatsTwoWayBWT($P, C, L, L', sp, ep, sp', ep'$)
(1)  if ($ep - sp < 2$) then
(2)      return ;
(3)  for each $s \in \Sigma$ do
(4)      $sp(s) \leftarrow C[s] + rank_s(L, sp-1)+1$;
(5)      $ep(s) \leftarrow C[s] + rank_s(L, ep)$;
(6)      $sp'(s) \leftarrow C[s] + rank_s(L', sp'-1)+1$;
(7)      $ep'(s) \leftarrow C[s] + rank_s(L', ep')$;
(8)  if ($ep(s) - sp(s) \neq ep - sp$ and $ep'(s) - sp'(s) \neq ep' - sp'$ for all $s$) then
(9)      $P$ is a maximal repeat
(10) for each $s \in \Sigma$ do
(11)     $sp'(s) \leftarrow sp' + LessThan_s(L, ep) - LessThan_s(L, sp-1)$;
(12)     $ep'(s) \leftarrow sp'(s) + ep(s) - sp(s)$;
(13)     maximalRepeatsTwoWayBWT($sP, C, L, L', sp(s), ep(s), sp'(s), ep'(s)$);
```

First call to the algorithm is `maximalRepeatsTwoWayBWT("",`$C, L, L', 1, m, 1, m$`)` on string $A[1, m]$, where L is the BWT of A, L' is the BWT of A^r, and C the corresponding count array. At line (11), $LessThan_s(L, i)$ gives the number of symbols smaller than s in $L[1, i]$ and implements the elements of the sum at Eg. 4.3. This function can be implemented in $O(\log |\Sigma|)$ time on wavelet tree of L.

It is easy to see that the algorithm takes linear time in the overall length of all maximal repeats (ignoring alphabet-factors, which could be improved). Yet, this not optimal, as there could be an algorithm working in linear time in the number of maximal repeats; each repeat can be represented as an interval in A. Such algorithm exists using suffix tree: Let v be a node representing right-maximal repeat α. If there are two nodes v' and v'' with suffix links

$sl(v') = sl(v'') = v$, then α is also left-maximal. This follows from the same observation we used above with the two-way BWT; if there are two suffix links, then there are at least two occurrences of α preceded by some characters $a, b \in \Sigma$ such that $a \neq b$. The algorithm is then trivial. Build suffix tree of A (with suffix links). Initialize counter $rsl(v) = 0$ to each node. For all nodes v' apply $rsl(sl(v')) = rsl(sl(v')) + 1$. Nodes with $rsl(v) > 1$ represent maximal repeats. Total running time is linear, as the suffix tree can be computed in linear time [Gus97].

The above optimal algorithm is trivial to make space efficient just by using a *compressed suffix tree* [NM07] as a black box. However, it is instructive to see the connection to the above version with the two-way BWT. Consider the optimal algorithm on suffix tree of A^r. Essentially the optimal algorithm is able to directly skip all recursion steps of `maximalRepeatsTwoWayBWT` that have $ep(s) - sp(s) = ep - sp$ for some s, that is, all cases where there is only one character to follow in the backtracking step. These cases correspond to edges of the suffix tree. Hence, the essential feature exploited in the optimal algorithm is to be able to visit only nodes of the suffix tree, and to retrieve the corresponding suffix array interval $[sp', ep']$, so that intervals $[sp'(s), ep'(s)]$ can be computed as in lines (6-7).

4.2.2 Maximal exact matches and variants

The algorithms for maximal repeats considered above can be modified to solve maximal exact matches problem. We will discuss the modification required for the two-way BWT version as the modifications to the other algorithms are analogous. For example, one can concatenate A and B into $A\#B$, store an indicator vector I such that $I[i] = 1$ iff $Pos[i] > |A| + 1$, that is, mark the suffixes of B in suffix array Pos of the concatenation. Then the condition $(ep - sp < 2)$ at line (1) in Algorithm `maximalRepeatsTwoWayBWT` can be changed to $(rank(I, ep) - rank(I, sp - 1) = 0$ or $rank(I, ep) - rank(I, sp - 1) = ep - sp + 1)$. Line (9) can be changed to form sets $Q = \{(Pos[i], Pos[i] + |P| - 1) \mid Pos[i] \leq |A|, sp \leq i \leq ep\}$ and $R = \{(Pos[i], Pos[i] + |P| - 1) \mid Pos[i] > |A| + 1, sp \leq i \leq ep\}$ and to output all tuples (P, q, r) for $q \in Q$ and $r \in R$.

One can also extend the algorithms for *maximal unique matches* (MUM) problem. A unique match is a substring that appears only once in both A and B. Maximal unique match is a unique match that cannot be extended left or right without removing any occurrence. This problem naturally extends to a set of sequences. Let us consider how to solve it with two sequences A and B. Make the same modification to condition at line (1) in Algorithm `maximalRepeatsTwoWayBWT` as with MEMs. In addition, add condition $ep - sp = 2$ to the condition at line (8). With d strings $A^1, A^2, \ldots, A^d$, the latter condition at line (8) becomes $ep - sp = d$, but the former condition at line (1) becomes more complicated. One approach is to replace the indicator vector I with an array D such that $D[i] = j$ iff $Pos[i]$ belongs to the suffix of A^j in the concatenation $A^1\#A^2\#\cdots A^d$. Wavelet tree of D can be used for computing $\text{RangeColorCount}(sp, ep) = |\{D[j] \mid sp \leq j \leq ep\}|$ (see exercises for an analogous operation). We can then replace the condition at line (1) with $\text{RangeColorCount}(sp, ep) < d$. This test will take $O(\log d)$ time. However, space is the real bottleneck in this approach, as wavelet tree of D occupies $n \log d(1+o(1))$ bits, which can be much larger than the $O(n \log |\Sigma|)$ required for compressed suffix tree that is able to simulate the optimal algorithm. Using a mechanism by [Hui92], one can obtain all the RangeColorCount values in overall $O(n)$ time with no dependency on the number d of sequences in the collections. This requires additional storage, but it turns out that the values can be computed on-the-fly during the construction of compressed suffix tree in $O(n \log n \log |\Sigma|)$ time using $O(n \log |\Sigma| + d \log n)$ bits of space [FMV08]. We omit the details here.

4.2.3 Overlap computation

The overlap-layout-consensus approach to fragment assembly starts by finding all overlaps between fragments. With high-throughput sequencing, we have huge number of relatively short reads, making this task enormous. For this reasons k-mer / de Bruijn -based approaches have been more popular recently. However, one can use BWT-based indexes to solve the problem of exact overlap computation efficiently, as we will learn next.

Let $r_1, r_2, \ldots r_n$ be set of reads of total length N. We wish to find all pairs (r_i, r_j) such that they can be decomposed as $r_i = \alpha\beta$ and $r_j = \beta\eta$ with $|\beta| \geq K$, where K is the minimum overlap length threshold. In addition, we would like to find maximal such β for each pair.

Consider the following algorithm to solve the problem. Build a compressed suffix array for sequence $T = \#r_1\#r_2\#\cdots\#r_n$. For all suffixes starting with $\#$ associate the corresponding read number i following it. Then do backward search for each r_i separately. After K steps, check if one could proceed with $\#$. If so, report the overlap(s) and proceed with backward search repeating the same check at every step. This way one obtains all overlaps of length at least K.

The problem with the above approach is that we may report some redundant overlaps; we would like to report only the maximal ones. A better algorithms exists [Gus97], using suffix trees. We will now review it, and leave as an exercise to think about possible problems in implementing this algorithms with a compressed suffix tree instead.

Consider a suffix tree built on $T = r_1\#_1 r_2\#_2 \cdots \#_{n-1} r_n \#_n$. Initialize an empty stack for each read. Make a depth-first search on suffix tree and when visiting node v first time add its string depth (if it is at least K) to all stacks i such that there an edge from v starting with symbol $\#_i$. When visiting node v last time, reverse the situation by popping from the corresponding stacks. Now, when visiting a leaf node corresponding to suffix $r_i\#i\cdots$, the top-most values in the non-empty stacks give the longest overlaps (those that are at least of length K). Non-emtpty stacks can be maintained in double-linked lists during the traversal, and with a table of pointers $P[1\dots n]$ such that $P[i]$ gives the pointer to the corresponding stack in the double-linked list one can update the list of stacks in constant time when stacks get empty or non-empty.

It is also possible to extend the above approaches to compute the transitive closure of the overlap/string graph directly [SD10].

4.3 Higher level sequence analysis

4.3.1 Variation calling

Read mapping alignments provide information about SNPs in donor genome; if position j in T is covered by r_j reads of which p percent say that there is nucleotide a and rest say there is nucleotide $b = T[j]$, one can reason whether this is because of heterozygous polymorphism or because of a measument error. However, measurement errors are easy to rule out; they are independent events, so probability of observing many in the same position decreases exponentially. Say q is the measurement error probability, then the probability that there is no polymorphism at position j is $q^{r_j p/100}$.

Hence, using sequencing with high enough coverage, one can easily obtain the SNP profile of donor genome with reasonable accuracy.

The same straightforward idea can basically be applied to short indels, but some more care needs to be taken here. Reads are aligned independently of each others, so in case of indel areas the alignments can disagree on the exact location; same alignment score can be obtained with slight variations of the alignment. To go over this problem, take the r_j reads covering position j and compute the optimal multiple alignment. Then indels will be assigned to same columns. Computing optimal multiple alignment is plausible using the Carillo & Lipman approach [CL88, LAK89], bacause it is sufficient to

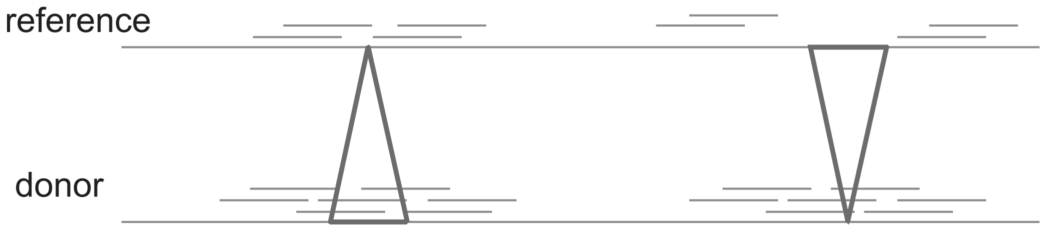


Figure 4.2: Read mapping behaviour on insertions (left) and deletions (right).

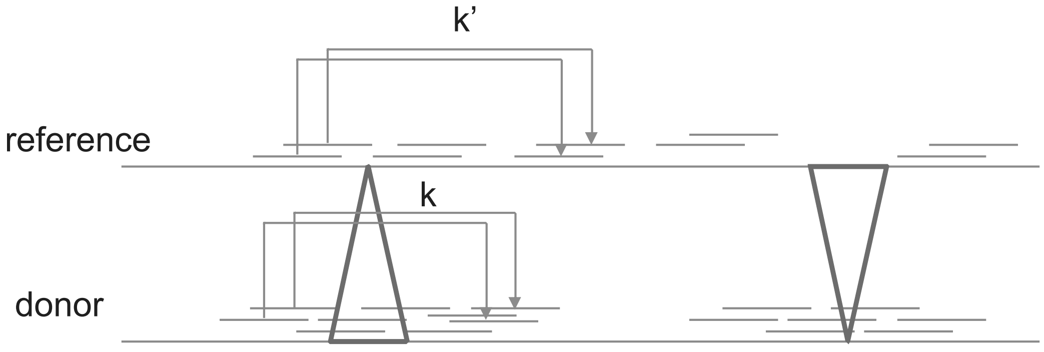


Figure 4.3: Paired end read mapping behaviour on insertions (left) and deletions (right).

extract only small region surrounding the indel cluster.

Larger variations are more difficult to detect. If there is a deletion in the donor genome, then there should be an uncovered region in the reference of the length of the deletion. If there is an insertion in the donor genome, then there should be a pair of consecutive positions $(j, j+1)$ in the reference such that j and $j+1$ are not covered by any same read r (see Fig. 4.2). Moreover, there should be reads whose prefix matches $t_{j'...j}$ and reads whose suffix matches $t_{j+1...j''}$; in case of deletions, both conditions should hold for some reads. To fill the inserted regions, one can apply fragment assembly: Consider all reads that have not mapped anywhere in the genome as well as $t_{j-m...j}$ and $t_{j+1...j+m+1}$. Try to create a *contig* that starts with $t_{j-m...j}$ and ends with $t_{j+1...j+m+1}$. The above is a simplistic view of indel detection; in practice one must prepare for background noise.

Another approach to large scale variation calling is to use paired end or mate pair reads. Consider paired end read pairs $(L_1, R_1), (L_2, R_2), \ldots, (L_r, R_r)$ such that reads L_i cover position j in the reference. The average distance of L_i and R_i should be k with some known variance. Now one can compute the average k' and variance of the observed distances of all L_i and R_i, and by comparing the observed distribution to the given one can decide whether the deviation is statistically significant. The expected indel length is $|k - k'|$ (see Fig. 4.3). Notice that now the length of the insertions can also been observed. This information is useful for the fragment assembly approach above since it gives a constraint to the contig to be constructed. In detail, consider an *overlap graph* with unmapped reads, $t_{j-m\ldots j}$ and $t_{j+1\ldots j+m+1}$ as nodes, and with edges representing the overlap lengths of the reads. Then the task is to find a path from $t_{j-m\ldots j}$ to $t_{j+1\ldots j+m+1}$ with length (counted as total length of the reads encountered minus the overlaps) close to the given estimate. Like many tasks related to fragment assembly, this is NP-hard [NU02]. However, the task is in practice not hopeless since the overlap graph contains only the unmapped reads; a (birectional) level-wise search may work fast enough.

4.3.2 Transcript expression and splice-site detection

Recall that from RNA-sequencing read alignment we obtain not only coverage counts for each position in the genome, but also for each splice-site alternative. Given a gene annotation (chains of exon locations for each transcript of each gene), we can easily count the average coverage for each gene and hence derive gene expression values analogous to microarray gene regulation experiments. With the positional and splice-site coverage counts, it is possible to go deeper in the analysis. First, we can count the average coverage $C(e)$ for each exon $e \in \mathcal{E}$ and the average coverage $C(e', e)$ for each splice-site alternative (e', e), where $e' \in \mathcal{E}$ and $e \in \mathcal{E}$ are two exons getting votes from split reads, and $e \in \mathcal{E}$ is the set of all exons. Let us denote by $\mathcal{T}$ the set of annotated transcripts. Denote $e \in t$ if exon e belongs to the transcript $t \in \mathcal{T}$, and $(e', e) \in t$ if $e' \in \mathcal{T}$ and $e \in \mathcal{T}$. A reasonable assumption on how transcript sequencing should behave is that the coverage does not depend on the position inside the transcript. We could hence assume that each transcript has an unknown average coverage $C(t)$. We obtain the *Annotated Transript Expression* problem: Estimate average coverages c_t, $t \in \mathcal{T}$, that best explain the measured coverages $C(e)$ and $C(e', e)$, $e', e \in \mathcal{E}$. One way to solve this is to use *least squares method*: minimize

$$\sum_{e \in \mathcal{E}} \left(C(e) - \sum_{t \in \mathcal{T}: e \in t} c_t \right)^2 + \sum_{e', e \in \mathcal{E}: C(e', e) > 0} \left(C(e', e) - \sum_{t \in \mathcal{T}: (e', e) \in t} c_t \right)^2$$

with the conditions $c_t \geq 0$, $t \in \mathcal{T}$ [KBD10, Hon11]. Least squares has standard closed form matrix-algebraic solutions that are out of the scope of this course.

The gene annotation was used here just to define the set of transcripts. One could be more ambitious by defining the possible transcripts from the data, that is, to solve *Unannotated Transcript Expression* problem. Consider the *exon chaining graph* implicit above, where exons are nodes, and exon-exon pairs corresponding to split reads are the directed edges. Then any path from an exon containing a start codon to an exon containing a stop codon is a possible transcript. Nodes and edges have weights $C(e)$ and $C(e', e)$, respectively. For simplicity of exposition, we assume that different genes share no exons (like above). As this is a directed acyclic graph, some path problems on it can be solved by simple dynamic programming. For example, heaviest path from exon e' to exon e is easy to compute, but the result is more like a greedy choice for a transcript that explains maximally the coverage. A better way to model the problem is to fix the number of transcripts k and search for k paths in the exon chaining graph and an average coverage for each such that the least squares estimate gets minimized. This is a hard optimization problem, but can be solved efficiently under some assumptions on the input (see exercises).

Trapnell et al. [Tea10] use a graph similar to the exon chaining graph, with one significant difference; they construct the graph directly from read alignments. The graph is then *overlap graph* and is constructed by assigning reads to nodes and having an edge between two reads if their alignment to the reference overlaps and is *compatible*, i.e., reads split to different exons do not have an edge. Then they seek for *minimum path cover* (minimal set of paths that cover all reads) in this overlap graph, which can be solved in polynomial time by reducing the problem to maximum matching on bipartite graph. These paths are in principle the proposed transcripts; the actual computations are much more involved [Tea10]. The major difference to the exon chaining approach we studied above is that the objective is not directly to optimize squared error of expression levels. However, they do assign costs to the edges based on the proportion of compatible overlapping reads versus incompatible overlapping reads, and therefore the resulting minimum weight minimum path cover may well be resulting to transcripts that implicitly minimize closely the same criteria.

More recently, the problem has been approached using *lasso-regression modeling* [LFJ11]. Let us number nodes and edges of the exon chaining graphs with unique numbers from 1 to M, and all possible paths (transcripts) from 1 to N. One can write the optimization problem as follows:

$$\begin{aligned} \min f(X) &= \min \sum_{i=1}^{M} \left(c_i - \sum_{j=1}^{N} a_{ji} x_j \right)^2 \qquad (4.4) \\ \texttt{s.t.} \quad & x_j \geq 0, 1 \leq j \leq N \\ & \sum_{j=1}^{N} x_j \leq \delta, \end{aligned}$$

where c_i's are the coverage values of nodes and edges, $a_{ji} = 1$ if node/edge i belongs to path number j, otherwise $a_{ji} = 0$, and x_j's are the coverage values of the paths (transcripts). The idea is that setting parameter δ small restricts the choice of x_j's; the hypothesis is that the optimal solutions then choose couple of non-zero x_j's and set other to zero. This formulation can be solved using any standard quadratic programming (QP) solver (e.g. using matlab). The problem with the approach is that one still needs to enumerate all paths and also the the λ-restriction does not directly minimize the size of the non-zero assigned paths.

We have omitted here the statistical corrections required for doing robust gene regulation studies based on the transcript expression levels. However, such corrections are analogous to the microarray gene expression level studies (out of the scope of this course), yet dedicated analysis methods have been developed taking into account the characteristics of RNA-seq data [ORY10].

4.3.3 Co-linear chaining

Consider the Unlabeled Transcript Expression problem above, where no gene annotation is given as input for estimating the transcript coverages. A possible alternative way to solve it is to do *de novo* fragment assembly with the reads, hope that the resulting contigs represent the transcripts, and align the predicted transcripts to the genome using the approach in Sect. 2.5. There are two bottlenecks: (1) fragment assembly of short reads from (almost identical) transcripts is likely to result into misassemblies; (2) alignment of transcripts to genome using the simple dynamic programming algorithms is not a high-throughput approach. For (1), the partial solution is to use longer reads; these are called *Expressed sequence tags (ESTs)*, and are produced using e.g. Sanger sequencing. However, this is much more costly and is not used in practice for high-throughput studies but for studying e.g. genomes whose reference sequence is unknown. For (2), the solution is to use *co-linear chaining* studied next.

Co-linear chaining exploits the fact that pieces of RNA should match almost perfectly the corresponding genomic DNA. Therefore one can first compute quickly all good local alignments and proceed to collecting the global alignment. For example, one could start from the list of MEMs between RNA and genome. Let us assume that such local alignment has been done, and we have a set of tuples $V = \{(x, y, c, d)\}$ such that $T[x, y]$ matches $P[c, d]$, where $T[1, n]$ is the genome and $P[1, m]$ the RNA transcript. The goal is to find a sequence of tuples $S = s_1 s_2 \cdots s_p \in V^p$ such that $s_j.y > s_{j-1}.y$, $s_j.d > s_{j-1}.d$, for all $1 \leq j \leq p$, and coverage$(P, S) = |\{i \mid i \in [s_j.c, s_j.d]$ for some $1 \leq j \leq p\}|$ is maximized. That is, find tuples preserving order in both T and P such that the coverage of P is maximized. In short, we call this *ordered* coverage of P. The solution uses dynamic programming and the invariant technique we learned in Sect. 2. First sort tuples in V by the coordinate y into sequence $v_1 v_2 \cdots v_N$. Then fill a table $C[1 \ldots N]$ such that $C[j]$ gives the maximum ordered coverage of $P[1, v_j.d]$ using any subset of tuples from $\{v_1, v_2, \ldots v_j\}$. Hence $\max_j C[j]$ gives the total maximum ordered coverage of P. It remains to derive the recurrence for computing $C[j]$. There are two cases: (a) Either the previous tuple does not overlap v_j in P; or (b) the previous tuple overlaps v_j in P. For (a) we can see that the recurrence is

$$C^{\mathrm{a}}[j] = \max_{j' : v_{j'}.d < v_j.c} C[j'] + (v_j.d - v_j.c + 1). \tag{4.5}$$

For (b) we can see that the recurrence is

$$C^{\mathrm{b}}[j] = \max_{j' : v_j.c \leq v_{j'}.d \leq v_j.d} C[j'] + (v_j.d - v_{j'}.d), \tag{4.6}$$

which works correctly unless there is a tuple $v_{j'}$ satisfying the condition such that $v_{j'}.c > v_j.c$. Such containments can however be ignored when computing the final value $C[j] = \max(C^{\mathrm{a}}[j], C^{\mathrm{b}}[j])$, because case (a) gives always a better result (exercise). Now we can use the invariant technique to obtain range maximum queries, which can be solved using the search tree in

Lemma 2.2.6 (or its dual version with minimum replaced by maximum):

$$\begin{aligned}
C^{\mathrm{a}}[j] &= (v_j.d - v_j.c + 1) + \max_{j' : v_{j'}.d < v_j.c} C[j'] \\
&= (v_j.d - v_j.c + 1) + \mathcal{T}.Maximum(0, v_j.c - 1), \\
C^{\mathrm{b}}[j] &= v_j.d + \max_{j' : v_j.c \leq v_{j'}.d \leq v_j.d} C[j'] - v_{j'}.d \\
&= v_j.d + \mathcal{I}.Maximum(v_j.c, v_j.d), \\
C[j] &= \max(C^{\mathrm{a}}[j], C^{\mathrm{b}}[j]).
\end{aligned}$$

For these to work correctly, we need to have updated the trees $\mathcal{T}$ and $\mathcal{I}$ properly for $j' : 1 \leq j' < j$: $\mathcal{T}.Insert(C[j'], v_{j'}.d)$ and $\mathcal{I}.Insert(C[j'] - v_{j'}.d, v_{j'}.d)$. Running time is $O(N \log N)$. The pseudocode is given below.

```
Algorithm CoLinearChaining($V$ sorted by $y$-coordinate: $v_1, v_2, \ldots, v_N$)
(1) $\mathcal{T}.Insert(0,0)$; $\mathcal{I}.Insert(0,0)$;
(2) for $j \leftarrow 1$ to $N$ do
(3)         $C^{\mathrm{a}}[j] \leftarrow (v_j.d - v_j.c + 1) + \mathcal{T}.Maximum(0, v_j.c - 1)$;
(4)         $C^{\mathrm{b}}[j] \leftarrow v_j.d + \mathcal{I}.Maximum(v_j.c, v_j.d)$;
(5)         $C[j] \leftarrow \max(C^{\mathrm{a}}[j], C^{\mathrm{b}}[j])$;
(6)         $\mathcal{T}.Insert(C[j], v_j.d)$;
(7)         $\mathcal{I}.Insert(C[j] - v_j.d, v_j.d)$;
(8) return $\max_j C[j]$;
```

4.3.4 Whole genome alignment

Pair-wise global alignment of two complete genomes is an enermous task using optimal dynamic programming algorithms. Let us consider a high-throughput algorithm exploiting the maximal unique matches (MUMs) studied earlier, and directly the case of having multiple genomes to be aligned. Let $A^1, A^2, \ldots, A^d$ be the sequences of length n_i for $1 \leq i \leq d$. The algorithm uses divide and conquer strategy: Find the maximum length MUM, say α, shared by all sequences. This MUM has exactly one location in each sequence, say j_i for $1 \leq i \leq d$, so it can be used for splitting the set into two independent parts: $A^1_{1\ldots j_1}, A^2_{1\ldots j_2}, \ldots, A^d_{1\ldots j_d}$ and $A^1_{j_1+|\alpha|\ldots n_1}, A^2_{j_2+|\alpha|\ldots n_2}, \ldots, A^d_{j_d+|\alpha|\ldots n_d}$. Now apply the same recursively for each part until getting sufficiently short sequences in each part; then apply optimal multiple alignment algorithm for each part. The MUMs calculated this way work as anchors for the multiple alignment.

4.4 Exercises

1. Some mate pair sequencing techniques work by having an adapter where the two tails of a long DNA fragment bind to, forming a circle. This circle is cut in one random place and then X nucleotides apart from it again, forming one long fragment and another shorter one (assuming X much smaller than circle length). The fragments containing the adapter are fished out from the pool (together with some background noise). Then these adapter containing fragments are sequenced from both ends to form the mate pair. Because of the random process of cutting, some of the mate pair reads may overlap the adapter. Such overlaps should be cut before using the reads any further.

   a) Give an algorithm to cut the adapter from the reads. Take into account that short overlaps may appear by chance and that the read positions have the associated quality values denoting the measurement error probability.

   b) How can you use the information about how many reads overlap the adapter to estimate the quality of fishing?

2. Construct the Burrows-Wheeler transform of `ACATGATCTGCATT` and simulate backward search on it with pattern `CAT`.

3. Construct the Burrows-Wheeler transform of `ACATGATCTGCATT` and simulate 1-mismatch backward backtracking search on it with pattern `CAT`.

4. Show that the values $\kappa(\alpha)$ are computed correctly with the backward search on reverse FM-index algorithm variation described in the context of BWA, i.e. the values the algorithm computes have the property that there is no splitting of α to more pieces such that no piece would occur in the text.

5. Give pseudocode for k-mismatches search using two-way BWT. You may assume that a partitioning of the pattern is given together with the number of errors allowed in each piece. Start the search from a piece allowed to contain fewest errors.

6. What the read length m should be so that there is expected to be no occurrences of the read in a random sequence of length n, when allowing k mismatches?

7. What the read length m should be so that there is expected to be no occurrences of the read in a random sequence of length n, when allowing k errors? (use approximation, the exact formula is difficult)

8. Consider different large scale variations in genome, like gene dublication, copy number variation, inversions, translocations, etc. How they can be identified using read mapping? Is there an advantage of using paired end reads?

9. Consider an array $A[1, n]$ and a query $A.\text{RangeCount}(l, r, i, j) = |\{k \mid i \leq A[k] \leq j, l \leq k \leq r\}|$.

   a) Show that the balanced wavelet tree of Sect. 3.4 build on A can support the RangeCount query in $O(\log n)$ time.

   b) How can this query be exloited in RNA-sequencing read alignment and in paired end read alignment? *Hint. Use the structure on suffix array of the reference genome.*

10. Consider the overlap computation with stacks and suffix tree as described in Sect. 4.2.3. Assume you have a compressed suffix tree representing the concatenation of reads in small space, and it support depth-first travelsal, retrieval of string depth, etc. common suffix tree operations efficiently. How much extra space you need in the worst case for the stacks, doubly-linked lists, etc. structures to implement the overlap computation on top of the given compressed suffix tree? Do you find any compression methods to improve the extra space requirement? *Hint.* There are dynamic bit-vectors taking $O(N)$ bits space to represent a bit-vector of length N allowing logarithmic time inserts and deletions. One can imagine representing stacks (whose values are increasing) with them, and concatenation of stacks with another boundary vector.

11. Prove that the co-linear chaining algorithm works correctly even when there are tuples containing other tuples in T or in P, i.e., tuples of type (x, y, c, d) and (x', y', c', d') such that either $x < x' \leq y' < y$ or $c < c' \leq d' < d$ (or both).

12. Modify the co-linear chaining algorithm to solve the following variations of the ordered coverage problem.

    a) Find the maximum ordered coverage of P such that all the tuples involved in the coverage must overlap in P.

   b) Find the maximum ordered coverage of P such that the distance in P between two consecutive tuples involved in the coverage is at most a given threshold value α.

   b) Find the maximum ordered coverage of P such that the distance in T between two consecutive tuples involved in the coverage is at most a given threshold value β.

13. Show that the values $\kappa(\alpha)$ in `BWA` are correct lower-bounds, i.e., there cannot be any occurrence missed when using the rule $k' + \kappa(\alpha) > k$ to prune search space.

Bibliography

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CGL04] R. Cole, L. A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors. In *Proceedings of the Symposium on Theory of Computing*, pages 91–100, 2004.

[CL88] H. Carillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48:1073–1082, 1988.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[CLS$^+$06] Ho-Leung Chan, Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching*, LNCS 4009, pages 49–59. Springer-Verlag, 2006.

[CLZU02] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted cost matrices. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA'2002)*, pages 679–688, 2002.

[EGGI92] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM*, 39(3):519–545, July 1992.

[Eli75] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.

[FM05] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005.

[FMN08] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165, 2008.

[FMV08] J. Fischer, V. Mäkinen, and N. Välimäki. Space-efficient string mining under frequency constraints. In *Proc. Eighth IEEE International Conference on Data Mining (ICDM 2008)*, pages 193–202. IEEE Computer Society, 2008.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9(1.4):1–47, 2004.

[GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GP92] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92:49–76, 1992.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV06] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2006.

[Hon11] A. Honkela. personal communication, 2011.

[HS77] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, May 1977.

[Hui92] Lucas Chi Kwong Hui. Color set size problem with application to string matching. In *Proc. Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[JMMW07] David S. Johnson, Ali Mortazavi, Richard M. Myers, and Barbara Wold. Genome-wide mapping of in vivo protein-dna interactions. *Science*, 316:1497–1502, 2007.

[KA05] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, 2005.

[KBD10] Hyunsoo Kim, Yingtao Bi, and Ramana V. Davuluri. Estimating the expression of transcript isoforms from mrna-seq via non-negative least squares. *Bioinformatic and Bioengineering, IEEE International Symposium on*, 0:296–297, 2010.

[KM06] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1(6):246–251, 2006.

[KN07] Juha Kärkkäinen and Joong Chae Na. Faster filters for approximate string matching. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX07)*, pages 84–90. SIAM, 2007.

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[KSPP05] D. Kim, J. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2–4):126–142, 2005.

[LAK89] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence analignment. In *Proceedings of the National Academy of Sciences of the USA*, volume 86, pages 4412–4415, 1989.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 2009. Advance access.

[LFJ11] W. Li, J. Feng, and T. Jiang. IsoLasso: a LASSO regression approach to RNA-Seq based transcriptome assembly. *J. Comput. Biol.*, 18(11):1693–707, 2011.

[LST$^+$08] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24:791–797, 2008.

[LTPS09] Ben Langmead, Cole Trapnell, Maihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37:63–78, 1988.

[LYL$^+$09] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2. *Bioinformatics*, 25(15):1966–1967, 2009.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[MN06] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, LNCS 3887, pages 703–714, 2006.

[MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[MNU03] V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for transposition invariant string matching. In *Proc. 20th International Symposium on Theoretical Aspects of Computer Science (STACS'03)*, volume 2607 of *LNCS*, pages 191–202. Springer-Verlag, 2003.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[Mun96] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.

[MVLK10] V. Mäkinen, N. Välimäki, A. Laaksonen, and R. Katainen. Unifying view of backward backtracking in short read mapping. In *Algorithms and Applications, Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, volume 6060 of *LNCS Festschrifts*, pages 182–195. Springer, 2010.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surveys*, 33(1):31–88, 2001.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[NU02] M. Nykänen and E. Ukkonen. The exact path length problem. *J. Algorithms*, 42(1):41–52, 2002.

[ORY10] Alicia Oshlack1, Mark D Robinson, and Matthew D Young. From rna-seq reads to differential expression results. *Genome Biology*, 11(220):1–10, 2010.

[Pag99] R. Pagh. Low redundancy in dictionaries with $O(1)$ worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

[PZ07] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 833–844, New York, NY, USA, 2007. ACM.

[RNO08a] L. Russo, G. Navarro, and A. Oliveira. Dynamic fully-compressed suffix trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 191–203, 2008.

[RNO08b] L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 362–373, 2008.

[RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[SD10] Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics [ISMB]*, 26(12):367–373, 2010.

[Tea10] Cole Trapnell and et al. Transcript assembly and quantification by rna-seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28:511–515, 2010.

[TPS09] Cole Trapnell, Lior Pachter, and Steven L. Salzberg. Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, 25(9):1105–1111, 2009.

[Ukk85] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.

[VLM10] Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM'10)*, volume 6129 of *LNCS*, pages 76–87. Springer, 2010.