Computational Genomics **Fall Semester, 2010**

Lecture 12: January 6, 2011

Lecturer: Ron Shamir *Scribe: Anat Gluzman and Eran Mick*

12.1 Algorithms for Next Generation Sequencing Data

12.1.1 Introduction

Ever since Watson and Crick elucidated the structure of the DNA molecule in 1953, thus proving that it carried the genetic information, the challenge of reading the DNA sequence became central to biological research. The earliest chemical methods for DNA sequencing were extremely inefficient, laborious and costly. For example, reliably sequencing the yeast gene for $tRNA^{Ala}$ (Holley, 1965) required the equivalent of a full year's work per person per basepair (bp) sequenced.[1]

Over the next few decades, sequencing became more efficient by orders of magnitude. In the 1970s, two classical methods for sequencing DNA fragments were developed by Sanger and Gilbert. In the 1980s these methods were augmented by the advent of partial automation as well as the cloning method, which allowed fast and exponential replication of a DNA fragment. By the time the human genome project was started in the 1990s, sequencing efficiency had already reached 200,000 bp/person/year, and when it concluded in 2002 this figure had gone up to 50,000,000 bp/person/year.

Over the past couple of years, new sequencing technologies - termed next generation sequencing (NGS) or deep sequencing - have emerged, which now allow the reliable sequencing of $100 \cdot 10^9$ bp/person/year. At the same time, the cost of sequencing has also sharply declined. While sequencing a single human genome cost $3 billion as part of the human genome project, NGS now allows compiling the full DNA sequence of a person for as little as $10,000 and efforts are underway to bring this down to $1000 within the next 3-5 years. Such diminished costs herald the age of personalized medicine, in which doctors will diagnose and treat each patient based on his or her unique genetic characteristics.

12.1.2 Principles of NGS Technology

Below we sketch the sequencing technology of Illumina, one of the leading companies in the field. Bear in mind that other available technologies are different and quite a few are currently under development.

[1] This lecture is based in part on lectures by Itsik Pe'er, Dept of CS, Columbia University.

First, the DNA is replicated, yielding millions of copies. Then, all the copies are randomly shredded using restriction enzymes or mechanical means into various fragments. These fragments form the input for the sequencing phase.

Hundreds of copies of each fragment are generated in one spot (cluster) on the surface of a huge matrix, though it is not initially known which fragment sits where. Then, the following stages are repeated:

1. A DNA replication enzyme is added to the matrix along with the 4 nucleotides. The nucleotides are slightly modified chemically so that each would emit a unique color when excited by laser, and so that each one would terminate the replication. Hence, the growing complementary DNA strand on each fragment in each cluster is extended by one nucleotide at a time.

2. The laser is used to obtain a read of the nucleotide now added in each cluster. The multiple copies in a cluster provide an amplified signal.

3. The solution is washed away, and with it the chemical modification on the last nucleotide which prevented further elongation and emitted the unique signal.

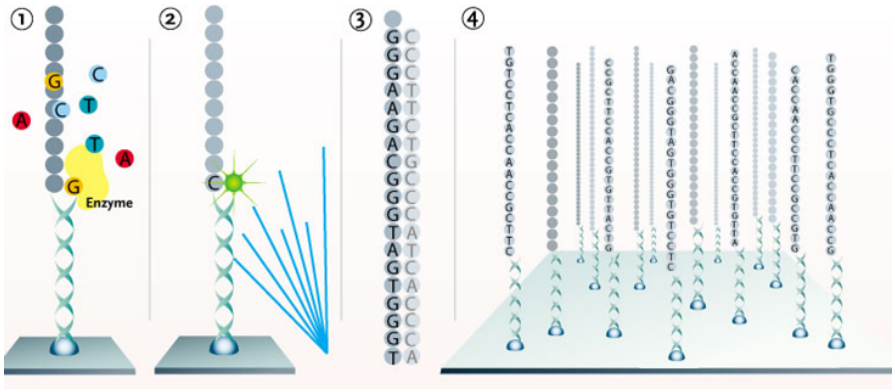

Figure 12.1: (1) A modified nucleotide is added to the complemntary DNA strand by a DNA polymerase enzyme. (2) A laser is used to obtain a read of the nucleotide just added. (3) The full sequence of a fragment thus determined through successive iterations of the process. (4) A visualization of the matrix where fragment clusters are attached to flow cells.

In this way, we can sequence millions of fragments efficiently and in parallel. These fragment sequences are now called *reads*, and they form the input for the computational problems presented in the next sections.

12.2 Mapping

We first focus on the problem of aligning the reads to the genome.

Problem 12.1 Short read mapping problem
INPUT: m l-long reads $S_1, \ldots, S_m$ and an approximate reference genome R.
QUESTION: What are the positions $x_1, \ldots, x_m$ along R where each read matches?

An example of this problem is when we sequence the genome of a person and wish to map it to an existing sequence of the human genome. The new sample will not be 100% identical to the reference genome due to the natural variation in the population, and so some reads may align to their position in the reference with mismatches or gaps. In diploid organisms, such as human beings, different alleles on the maternal and paternal chromosomes can lead to two slightly different reads mapping to the same location (some perhaps with mismatches). Additional complications may arise due to sequencing errors or repetitive regions in the genome which make it difficult to decide where to map the read.

12.2.1 Possible Solutions for the Mapping Problem

At face value, the problem of aligning a short read to a long genome sequence is exactly the problem of local alignment. However, the large parameters involved make such an approach impractical. In the human genome example, the number of reads m is usually 10^7-10^8, the length of a read l is 50-200 bp and the length of the genome $|R|$ is $3{\cdot}10^9$ bp (or twice, for the diploid genome).

Let us consider some other possible solutions:

1. The most naive algorithm would scan R for each S_i, matching the read at each position p and picking the best match. **Time complexity:** $O(ml|R|)$ for exact or inexact matching. Considering the parameters mentioned above, this is clearly impractical.

2. A less naive solution uses the Knuth-Morris-Pratt algorithm to match each S_i to R. **Time complexity:** $O(m(l+|R|)) = O(ml + m|R|)$ for exact matching. This is a substantial improvement but still not enough.

3. Building a suffix tree for R provides another solution. Then, for each S_i we can find matches by traversing the tree from the root. **Time complexity:** $O(ml+|R|)$, assuming the tree is built using Ukkonen's linear-time algorithm. This time complexity is certainly practical, and it has the additional advantage that we only need to build the tree for the reference genome once. It can then be saved and used repeatedly for mapping new sequences, at least until an updated version of the genome is issued.

However, space complexity now becomes the obstacle. Since the leaves of the suffix tree also hold the indices where the suffixes begin, saving the tree requires $O(|R|\log|R|)$ bits just for the binary coding of the indices, compared with $|R|\log|\Sigma|$ bits for the original text. The constants are also large due to the additional layers of information required for the tree (such as suffix links, etc.). Thus, we can store the text of the human genome using ∼750MB, but we'd need ∼64GB for the tree! The resultant size is much greater than the cache memory of most of today's desktop computers. Another problem is that suffix trees allow only for exact matching.

4. A fourth solution is to preprocess the reference genome into a hash table H. The keys of the hash are all the substrings of length l in R, and the value of each key is the position p in R where the substring ends. Then, given S_i the algorithm returns $H(S_i)$. **Time complexity:** $O(ml+l|R|)$, which is pretty good.

   The space complexity, however, remains too high at $O(l|R| + |R|\log|R|)$ since we must also hold the binary representation of each substring's position. A practical improvement which can be applied is packing the substrings into bit-vectors, that is representing each nucleotide as a 2-bit code. This reduces the space complexity by a factor of four. Further improvement can be achieved by partitioning the genome into several chunks, each of the size of the cache memory, and running the algorithm on each chunk in turn. Again, this approach only allows exact matching.

12.3 The MAQ Algorithm

The MAQ (Mapping and Alignment with Qualities) algorithm, presented in 2008 by Li, Ruan and Durbin[5], provides an efficient solution to the short read mapping problem which allows for inexact matching, up to a predetermined maximum number of mismatches, while taking into account practical cache memory limitations. It also accounts for base and read quality (and for identifying sequence variants, which will not be discussed here).

12.3.1 Algorithm Outline

The key insight is that if a read has, for example, one mismatch vis-à-vis its correct position in the reference genome, then partitioning that read into two makes sure one part is still exactly matched. Hence, if we perform an exact match twice, each time checking only a subset of the read bases, one subset is enough to find the match. We call such a subset a *template*.

Similarly, for two mismatches, it is possible to create six templates, some of which are noncontiguous, so that each template covers half the read and has a partner template that complements it to form the full read. If the read has no more than two mismatches, at

least one template will not be affected by any mismatch. Twenty such templates guarantee a fully matched template for any 3-mismatched read, etc. Figure 12.2 below demonstrates this property on an 8-bp read.

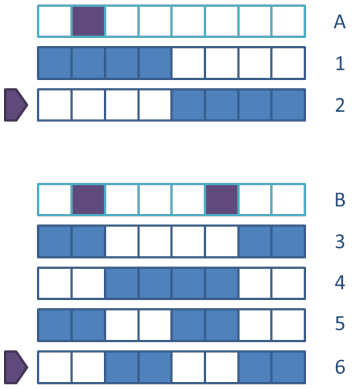

Figure 12.2: A and B represent reads, with purple boxes signifying positions of mismatches with respect to the reference. The numbered templates have blue boxes in the positions they cover. When comparing the genomic segment and read A through template 1, they will not match. However, template 2 must still be fully matched. Similalry, read B has two mismatches, but we see template 6 is fully matched despite this. Any combination of up to two mismatched positions will be avoided by at least one of the six templates. In fact, six templates guarantee 57% of all 3 mismatches will also have at least one unaffected template.

The algorithm can then generate the number of templates required to guarantee at least one full match for the desired maximal number of mismatches, and use that exact match as a seed for extending across the full read. Identifying the exact matches is done by *hashing the read templates* into template-specific hash tables, and then scanning the reference genome against each table.

In fact, it is not even necessary to generate templates and hash tables for the full read, since the initial seed will undergo extension, e.g. using the Smith-Waterman algorithm. Rather, the algorithm initially processes only the first 28 bases of each read. The first bases are chosen since they are the most accurately read ones.

Algorithm summary (for the case of up to two mismatches):

1. Index the first 28 bases of each read with complementary templates 1, 2; generate hash tables *H1*, *H2*.

2. Scan the reference: for each position and each orientation, query a 28-bp window through templates 1, 2 against the appropriate tables. If hit, extend and score the

complete read based on mismatches. For each read, keep the two best scoring hits and the number of mismatches therein.

3. Repeat steps 1+2 with complementary templates 3, 4, then 5, 6.

Remark The reason for indexing against a pair of complementary templates each time has to do with a feature of the algorithm regarding paired-end reads (covered in section 12.3.3 below), which we will not discuss.

Complexity:

- **Time Complexity:** $O(ml)$ for generating the hash tables in step 1, and $O(l|R|)$ for scanning the genome in step 2. We repeat steps 1+2 three times in this implementation but that does not affect the asymptotic complexity, using O notation.
- **Space Complexity:** $O(ml)$ for holding the hash tables in step 1, and $O(ml+|R|)$ total space for step 2. However, we only hold $O(ml)$ space in cache at any one time.

More accurately, we are not dealing with the full read length l, but rather just a window of length $l' \leq l$ (28-bp in the implementation presented above). The size of each key in a template hash table is only $l'/2$, since each template covers half the window. We therefore can provide a more precise term for the time and space complexity of step 1: $O(2 \cdot m \cdot l'/2) = O(ml')$. This reduction in space makes running the algorithm with the cache memory of a desktop computer feasible.

Note also that we did not take into account the time and space required for extending a match in step 2 using the Smith-Waterman algorithm. Let p be the probability of a hit, then the time complexity for this step becomes $O(l'|R| + p|R|l'^2)$. The space complexity becomes $O(ml' + l'^2)$. The value of p is small, and decreases drastically the longer l' is, since most l'-long windows in the reference genome will likely not capture the exact coordinates of a true read.

12.3.2 Read Mapping Qualities

Another feature of the MAQ algorithm is that it provides a measure of the confidence level for the mapping of each read, denoted by Q_S and defined as:

$$Q_S = -10 \, log_{10}(Pr\{\text{read } S \text{ is wrongly mapped}\}) \tag{12.1}$$

For example, $Q_S = 30$ if the probability of incorrect mapping of read S is 1/1000. This confidence measure is called *phred*-scaled quality, following the scaling scheme originally introduced by Phil Green and colleagues[3] for the human genome project.

We now present a simplified statistical model for inferring Q_S. In this simple case, we know the reads are supposed to match the reference precisely, and so any mismatches must be the result of sequencing errors. Assuming that sequencing errors along a read are independent, we can define the probability $p(z|x,u)$ of obtaining the read z given that its position in reference x is u as the product of the error probabilities of the observed mismatches in the alignment of the read at that position. These base error probabilities, which can also be presented on the *phred* scale, are empirically determined during the sequencing phase based on the characteristics of the emitted signal at each step.

Example If read z aligned at position u yields 2 mismatches that have a *phred*-scaled base quality of 10 and 20, then:

$$p(z|x,u) = 10^{-(20+10)/10} = 0.001$$

We now wish to calculate the posterior probability $p_s(u|x,z)$ of position u being the correct mapping for an observed read z. Applying Bayes' theorem, we get:

$$p_s(u|x,z) = \frac{P(z|x,u)P(u)}{P(x,z)} = \frac{P(z|x,u)P(u)}{\sum_{v=1}^{|R|-l+1} P(z|x,v)P(v)} = \frac{P(z|x,u)}{\sum_{v=1}^{|R|-l+1} P(z|x,v)} \tag{12.2}$$

The denominator was expanded to the sum of probabilities of seeing read z at any position along x, and we cancelled the probabilities of the positions since we assume all positions are equiprobable.

We have already seen how to calculate the nominator $P(z|x,u)$. However, the exact calculation of the denominator would require summing over all positions in R, which is impractical. Instead, it is estimated from the best hit, the second best and all other hits (we will not present an exact derivation of this). Having calculated the posterior probability, we can now give our confidence measure as:

$$Q_s(u|x,z) = -10\ log_{10}(1 - p_s(u|x,z)) \tag{12.3}$$

Obviously, MAQ uses a more sophisticated statistical model to deal with the issues we neglected in this simplification, such as true mismatches that are not the result of sequencing errors (SNPs = single nucleotide polymorphisms).

12.3.3 Paired-End Reads

Paired-end sequencing is a technology which provides additional information on the reads and allows for an even more reliable confidence measure of read mapping. The approach is based on circularization of genomic DNA

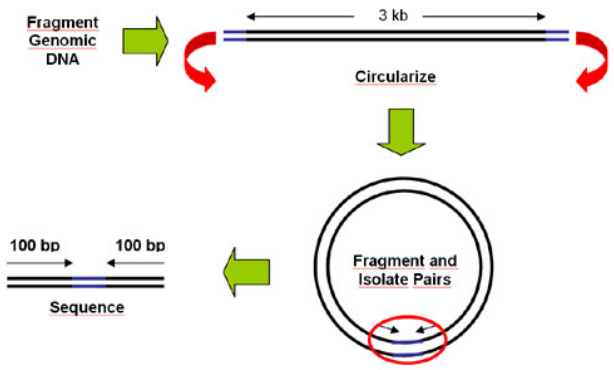

Figure 12.3: Paired-end sequencing. In this example, $L = 3000$ and $l = 100$. Source: http://www.lifesequencing.com/pages/estrategia-paired-end.

fragments of length $\sim L$, and sequencing two reads, each of length l, centered around the site of circularization (see Figure 12.3).

The two reads are called mate-pairs in the sense that we know they are supposed to be mapped to positions in the genome approximately $L - 2l$ apart. In the case of paired-end sequencing, MAQ jointly aligns the two reads in a pair. Sequences that fail to reach a mismatch score threshold but whose mate is mapped are searched with a gapped alignment algorithm in the region defined by the mate pair.

Paired-end mapping qualities are derived from single end mapping qualities. In MAQ, if there is a unique pair mapping in which both ends hit consistently (i.e., in the right orientation within the proper distance), we give the mapping quality $Q_P = Q_{S_1} + Q_{S_2}$ to both reads since we are now more certain of the correct mapping and we assume errors are independent. If there are multiple consistent hit pairs, we cannot know (based on the read pair alone) which pair is correct, so we take their single end mapping qualities as the final mapping qualities.

12.4 The Bowtie Algorithm

We have seen that one way to map reads to a reference genome is to index into a hash table either all l-long windows of the genome or of the reads. Holding these indices in memory requires a great deal of space, as discussed in section 12.2.1.

The Bowtie algorithm, presented in 2009 by Langmead et al.[1], solves problem 12.1 through a more space-efficient indexing scheme. This scheme is called the Burrows-Wheeler transform[2] and was originally developed for data compression purposes. In the following section, we will describe the transform and its uses by following a specific example.

12.4.1 Burrows-Wheeler Transform

To demonstrate the process, we shall apply the transform BW(T) to T = "the_next_text_that_i_index.":

1. First, we generate all cyclic shifts of T.

2. Next, we sort these shifts lexicographically. In this example we define the character '.' as the minimum and we assume that it appears exactly once, as the last symbol in the text. It is followed lexicogrpahically by '_', which is followed by the English letters according to their natural ordering. We call the resulting matrix M.

3. We now define the transform BW(T) as the sequence of the *last characters* in the rows of M. Figure 12.4 shows an example for the first few shifts. Note that this last column is a permutation of all characters in the text since each character appears in the last position in exactly one cyclic shift.

```
.the_next_text_that_i_index
_i_index.the_next_text_that
_index.the_next_text_that_i
_next_text_that_i_index.the
_text_that_i_index.the_next
_that_i_index.the_next_text
at_i_index.the_next_text_th
dex.the_next_text_that_i_in
e_next_text_that_i_index.th
ex.the_next_text_that_i_ind
ext_text_that_i_index.the_n
ext_that_i_index.the_next_t
hat_i_index.the_next_text_t
he_next_text_that_i_index.t
```

Figure 12.4: Some of the cyclic shifts of T sorted lexicographically and indexed by the last character.

Saving BW(T) requires the same space as the size of the text T since it is simply a permutation of T. In the case of the human genome, we saw that each character can be represented by 2 bits, so we require $\sim 2 \cdot 3 \cdot 10^9$ bits for storing the permutation instead of $\sim 30 \cdot 3 \cdot 10^9$ for storing all indices of T.

Thus far, we have seen how to transform a text T into BW(T). Let us now consider what information can be gleaned from BW(T), assuming we do not see T or M:

1. The first question we can ask is: given BW(T), how many occurrences in T of the character 'e'? We can easily answer this by counting the number of occurrences of 'e' in BW(T) since we have shown that this is simply a permutation of the text.

2. Can we also recover the first column of the matrix M? Certainly! All we have to do is sort BW(T) since the first column is also a permutation of all characters in the text, sorted lexicographically. Figure 12.5 demonstrates this.

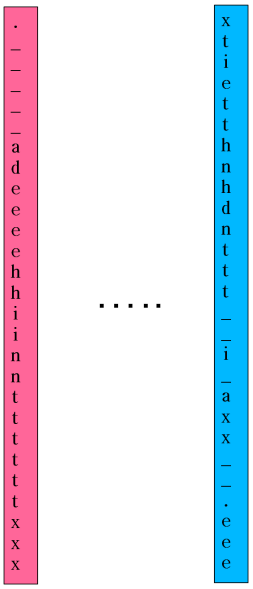

Figure 12.5: Recovering the first column (left) by sorting the last column.

3. How many occurrences of the substring 'xt' do we have in T? Remember that BW(T) is the last column of the lexicographical sorting of the shifts. Hence, the character at the last position of a row appears in the text T immediately prior to the first character in the same row (each row is a cyclical shift). So, to answer this question, we consider the interval of 't' in the first column, and check whether any of these rows have an 'x' at the last position. In Figure 12.5, we can see that there are two such occurrences.

4. Now we can recover the second column as well. We know that 'xt' appears twice in the text, and we see that 3 rows start with an 'x'. Two of those must be followed by a 't', but which ones? The lexicographical sorting determines this as well. In the above example, another 'x' is followed by a '.' (see first row). Therefore, '.' must follow the first 'x' in the first column since '.' is smaller lexicographically than 't'. The second and third occurrences of 'x' in the first column are therefore followed by 't'. We can

use the same process to recover the characters at the second column for each interval, and then the third, etc.

We have thus shown two central properties of the transform, which we now state formally following a formulation by Ferragina and Manzini[4].

Lemma 12.1 *(Last-First Mapping): Let M be the matrix whose rows are all cyclical shifts of T sorted lexicographically, and let L(i) be the character at the last column of row i and F(i) be the first character in that row. Then:*

1. *In row i of M, L(i) precedes F(i) in the original text: T =...L(i) F(i)... .*

2. *The j-th occurrence of character X in L corresponds to the same text character as the j-th occurrence of X in F.*

Proof:

1. Follows directly from the fact that each row in M is a cyclical shift.

2. Let X_j denote the j-th occurrence of char X in L, and let α be the character following X_j in the text and β the character following X_{j+1}. Then, since X_j appears above X_{j+1} in L, α must be equal or lexicographically smaller than β. This is true since the order is determined by lexicographical sorting of the full row and the character in F follows the one in L (property 1). Hence, when character X_j appears in F, it will again be above X_{j+1}, since α and β now appear in the second column and $X\alpha \leq X\beta$ (Figure 12.6 demonstrates this).

■

12.4.2 Reconstructing the Text

We now present an algorithm for reconstructing a text T from its Burrows-Wheeler transform BW(T) utilizing lemma 12.1[4]. In this formulation, we assume the actual text is of length u and we append a unique \$ character at the end, which is the smallest lexicographically (the '.' character played the role of \$ in our example above):

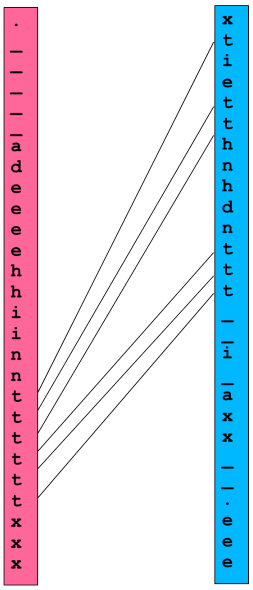

Figure 12.6: Last-first mapping. Each 't' character in L is linked to its position in F and no crossed links are possible.

UNPERMUTE[BW(T)]

1. Compute the array C$[1, \ldots, |\Sigma|]$: C(c) is the no. of characters $\{\$, 1, \ldots, c-1\}$ in T.

2. Construct the last-first mapping, tracing every character in L to its corresponding position in F:

   LF$[i]$ = C(L$[i]$) + r(L$[i]$, i) + 1, where r(c, i) is the number of occurrences of character c in the prefix L$[1, i-1]$.

3. Reconstruct T backwards:

   - $s = 1$, $T(u)$ = L$[1]$;
   - for $i = u-1, \ldots, 1$
     s = LF$[s]$, $T[i]$ = L$[s]$;

Figure 12.7: Example of running UNPERMUTE to recover the original text. Source: [1].

In the above example, T = acaacg\$ ($u = 6$) was transformed to BW(T) = gc\$aaac, and we now wish to reconstruct T from BW(T) using UNPERMUTE:

1. First, the array C is computed. For example, C(c) = 4 since there are 4 occurrences of characters smaller than 'c' in T (in this case, the '\$' and 3 occurrences of 'a'). Notice, that C(c) + 1 = 5 is the position of the first occurrence of 'c' in F.

2. Second, we perform the LF mapping. For example, $LF[c_2] = C(c) + r(c, 7) + 1 = 6$, and indeed the second occurrence of 'c' in F sits at F[6].

3. Now, we determine the last character in T: $T(6)$ = L(1) = 'g'.

4. We iterate backwards over all positions using the LF mapping. For example, to recover the character $T(5)$, we use the LF mapping to trace L(1) to F(7), and then $T(5)$ = L(7) = 'c'.

Remark We do not actually need to hold F in memory, which would double the space we use. Instead, we only keep the array C defined above, of size $|\Sigma|$, which we can easily obtain by looking at L alone.

12.4.3 Exact Matching

Next, we present an algorithm for exact matching of a query string P to T, given BW(T)[4]. The principle is very similar to UNPERMUTE, and we use the same definitions presented above for C and r(c, i). We denote by sp the position of the first row in the interval of rows in M we are currently considering, and by ep the position of the first row *beyond* this interval. So, the interval is defined by the rows $sp, \ldots, ep - 1$.

EXACTMATCH[$P[1, \ldots, p]$, BW(T)]

1. $c = P[p]$; sp = C[c] + 1; ep = C[c+1] + 1; $i = p - 1$;

2. while $sp < ep$ and $i \geq 1$
   $c = P[i]$;
   $sp = \mathrm{C}[c] + \mathrm{r}(c, sp) + 1$;
   $ep = \mathrm{C}[c] + \mathrm{r}(c, ep) + 1$;
   $i = i - 1$;

3. if($sp == ep$) return "no match";
   else return sp, ep;

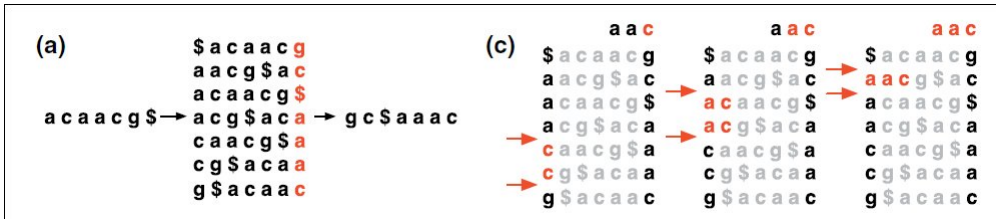

Figure 12.8: Example of running EXACTMATCH to find a query string in the text. Source: [1].

In the above example we use the same text as in Figure 12.7, while searching for $P =$ 'aac':

1. First, we initialize sp and ep to define the interval of rows beginning with the last character in P, which is 'c':
   - $sp = \mathrm{C}(\mathrm{c}) + 1 = 5$.
   - $ep = \mathrm{C}(\mathrm{g}) + 1 = 7$.
2. Next, we consider the preceding character in P, namely 'a'. We redefine the interval as the rows that begin with 'ac' utilizing the LF mapping. Specifically:
   - $sp = \mathrm{C}(\mathrm{a}) + \mathrm{r}(\mathrm{a}, 5) + 1 = 1 + 1 + 1 = 3$.
   - $ep = \mathrm{C}(\mathrm{a}) + \mathrm{r}(\mathrm{a}, 7) + 1 = 1 + 3 + 1 = 5$.
3. Now, we consider the preceding character, namely 'a'. We now redefine the interval as the rows that begin with 'aac'. Specifically:
   - $sp = \mathrm{C}(\mathrm{a}) + \mathrm{r}(\mathrm{a}, 3) + 1 = 1 + 0 + 1 = 2$.
   - $ep = \mathrm{C}(\mathrm{a}) + \mathrm{r}(\mathrm{a}, 5) + 1 = 1 + 1 + 1 = 3$.
4. Having thus covered all positions of P, we return the final interval calculated (whose size equals the number of occurrences of P in T).

Note that EXACTMATCH returns the indices of rows in M that begin with the query, but it does not provide the offset of each match in T. If we kept the position in T corresponding to the start of each row we would waste a lot of space. Instead, we can mark only some rows with pre-calculated offsets. Then, if the row where EXACTMATCH found the query has this offset, we can return it immediately. Otherwise, we can successively use LF mapping to find a row that has a precalculated offset, and then we simply need to add the number of times we applied this procedure to obtain the position in which we are interested. There is a simple time-space tradeoff associated with this process.

Example In figure 12.8 we found the row beginning with 'aac'. Assuming that row has no offset, we can use LF mapping to reach row 5. If that row has the offset 2, then the desired offset of 'aac' is $2 + 1 = 3$.

Note also that we did not describe how to efficiently compute $\mathrm{r}(c, i)$, which is an operation we repeat many times while running the algorithm. Again, it would be wasteful to save the value for each occurence of every character in the text. Instead, we can use a similar solution to that used above for finding the exact index of a match. We store only a subset of the values and locally compute back from an unkown value to a stored one. Ferragina and Manzini provide a more efficient (and complicated) solution for this issue[4].

12.4.4 Inexact Matching

So far, we have seen how to find exact matches of a query using BW(T). However, to map reads to the genome we must allow for mismatches. Recall (section 12.3.2) that each character in a read has a numeric quality value, with lower values indicating a higher likelihood of a sequencing error. Bowtie defines an alignment policy that allows a limited number of mismatches and prefers alignments where the sum of the quality values at all mismatched positions is low.

The search proceeds similarly to EXACTMATCH, calculating matrix intervals for successively longer query suffixes. If the range becomes empty (a suffix does not occur in the text), then the algorithm may select an already-matched query position and substitute a different base there, introducing a mismatch into the alignment. The EXACTMATCH search resumes from just after the substituted position. The algorithm selects only those substitutions that are consistent with the alignment policy and that yield a modified suffix that occurs at least once in the text. If there are multiple candidate substitution positions, then the algorithm greedily selects a position with a maximal quality value. Figure 12.9 demonstrates this. In the full Bowtie algorithm, backtracking can allow more than one mismatch, but the size of the backtracking stack is bounded by a parameter for efficiency.

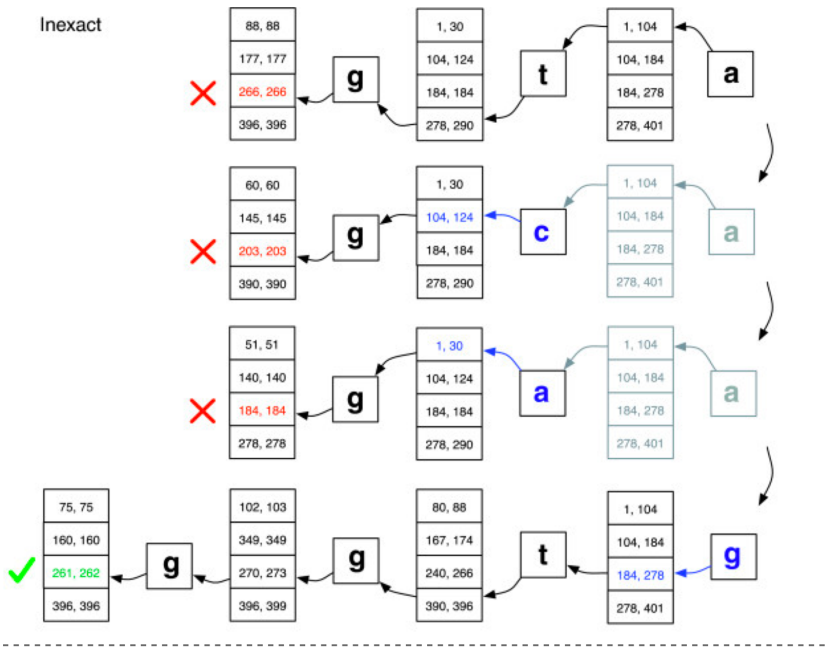

Figure 12.9: Example of running the inexact match variant of the Bowtie algorithm. In this example, we try to map the string 'ggta' to the genome, but we only succeed at mapping 'ggtg'. The array at each level of the backtracking shows the row intervals corresponding to suffixes with the 4 nucleotides at that position (in the order a, c, g, t). Source: [1].

12.5 Assembly

The next problem we wish to discuss breifly is how to assemble an unknown genome based on many highly overlapping short reads from it.

Problem 12.2 Sequence assembly
INPUT: m l-long reads $S_1, \ldots, S_m$.
QUESTION: What is the sequence of the full genome?

The crucial difference between the problems of mapping and assembly is that now we do not have a reference genome, and we must assemble the full sequence directly from the reads. To solve this problem, we first introduce a graph-theoretical concept.

Definition A k-dimensional *de Bruijn graph* of n symbols is a directed graph representing overlaps between sequences of symbols. It has n^k vertices, consisting of all possible k-tuples of the given symbols. The same symbol may appear multiple times in a tuple. If we have

the set of symbols $A = \{a_1, \ldots, a_n\}$ then the set of vertices is:

$$V = \{(a_1, ..., a_1, a_1), (a_1, ..., a_1, a_2), \ldots, (a_1, ..., a_1, a_n), (a_1, ..., a_2, a_1), \ldots, (a_n, ..., a_n, a_n)\}$$

If one of the vertices can be expressed by shifting all symbols of another vertex by one place to the left and adding a new symbol at the end, then the latter has a directed edge to the former vertex. Thus the set of directed edges is:

$$E = \{(\ (v_1, v_2, ..., v_k), (w_1, w_2, ..., w_k)\ ) | v_2 = w_1, v_3 = w_2, ..., v_k = w_{k-1}\}$$

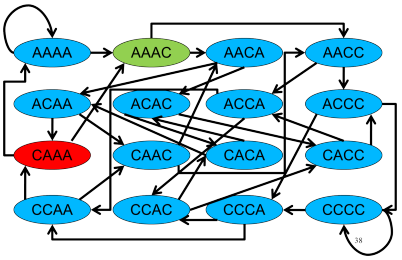

Figure 12.10: In this portion of a 4-dimensional de Bruijn graph, the vertex CAAA (in red) has a directed edge to vertex AAAC (in green) because if we shift the label of the former to the left and add the symbol C we get the label of the latter.

Given a read, every contiguous (k+1)-long word in it corresponds to an edge in the de Bruijn graph. We form a subgraph of the full de Bruijn graph by introducing only the edges that correspond to (k+1)-long words in some read. This is the graph we shall be working with. A path in this graph defines a potential subsequence in the genome. Hence, we can convert a read to its corresponding path:

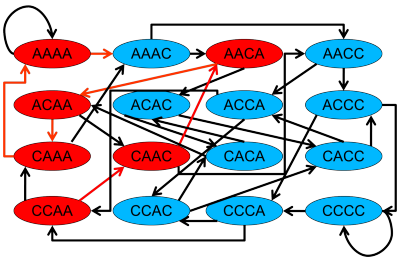

Figure 12.11: Using the same graph as in 12.10, we construct the path corresponding to CCAACAAAAC. We form it by shifting the sequence one position to the left each time, and marking the vertex with the label of the 4 first nucleotides.

After forming paths for all the reads, we now attempt to merge them through common vertices into one long sequence. For example:

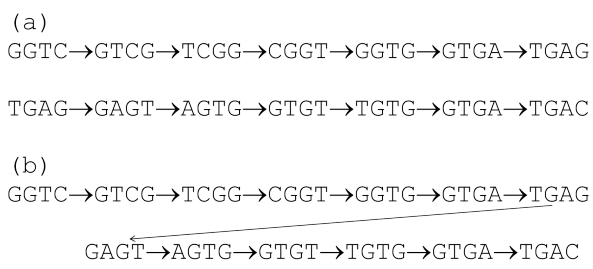

Figure 12.12: The two reads in (a) are converted to paths in the graph, and the common vertex TGAG is identified. Then, we can combine these two paths into (b).

Velvet, presented in 2008 by Zerbino and Birney[6], is an algorithm which utilizes de Bruijn graphs to assemble reads in this way. However, this assembly process can encounter some difficulties. For instance, repeats in the genome will manifest themselves as cycles in the merged path that we form. It is impossible then to tell how many times we must traverse each cycle in order to form the full sequence of the genome. Even worse, if we have two cycles starting at the same vertex, we cannot tell which one to traverse first. Velvet attempts to address this issue by utilizing the extra information we have in the case of paired-ends sequencing. However, we will not discuss the resolution of these issues here.

Bibliography

[1] Ben Langmead, Cole Trapnell, Mihai Pop and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 2009.

[2] Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.

[3] Brent Ewing and Phil Green. Base-calling of automated sequencer traces using *phred*. II. Error probabilities. *Genome Research*, 1998.

[4] Paulo Ferragina and Giovani Manzini. Opportunistic data structures with applications. *FOCS '00 Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000.

[5] Heng Li, Jue Ruan and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 2008.

[6] Daniel R. Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 2008.