10 Read alignment

Recall the high-throughput sequencing applications from Chapter 1. For example, in whole-genome resequencing we obtain a set of reads, that is, short extracts of DNA sequences from random positions in a donor genome. Assuming that the reference genome of the species has been assembled, one can align the reads to the reference in order to obtain an approximation of the content of the donor genome. We will consider this process in Chapter 14.

We remind the reader that reads come from both strands and, in diploid organisms, from both copies of each chromosome. This means that either the read or its reverse complement should align to the reference genome, but in what follows we do not explicitly repeat this fact. Let us now focus on how to efficiently align a set of (short) reads to a large genome.

The first observation is that, if a position in the reference genome is covered by several reads, one could try to exploit multiple alignment methods from Section 6.6 to obtain an accurate alignment. This gives an enormous semi-local multiple alignment problem with the reads and the reference sequence as input, and a multiple alignment as output (where reads can have an arbitrary amount of cost-free gaps before and after). Owing to the hardness of finding an optimal multiple alignment, shown for example in Section 6.6.3, we leave this approach just as a thought experiment. However, Chapter 14 deals with an application of this idea on a smaller scale.

Let us consider an incremental approach, which consists of aligning each read independently to the reference. This approach can be justified by the fact that, if the read is long enough, then it should not align to a random position in the genome, but exactly to that unique position from where it was sequenced from the donor. This assumes that this position is at least present in the reference genome, and that it is not inside a repeat. See Insight 10.1 for a more detailed discussion on this matter.

Insight 10.1 Read filtering, mapping quality

Since the incremental approach to read alignment is justified by the uniqueness of the expected occurrence positions, it makes sense to *filter* reads that are expected to occur inside a random sequence of length n. This filtering is aided by the fact that most sequencing machines give predictions on how accurately each base was measured. Let $\mathbb{P}(p_i)$ be such a probability for position i inside read P containing p_i. With probability $(1-\mathbb{P}(p_i))$ this position contains an arbitrary symbol. We may view

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

P as a conditional *joker* sequence, with position i being a joker matching any symbol with probability $(1 - \mathbb{P}(p_i))$. Such a conditional joker sequence P matches a random sequence of length m with probability

$$\prod_{i=1}^{m} \left(\mathbb{P}(p_i) q_{p_i} + (1 - \mathbb{P}(p_i)) \right), \tag{10.1}$$

where q_c is the probability of symbol c in a random sequence. An approximation of the expected number of occurrences of a conditional joker sequence P in random sequence $T = t_1 t_2 \cdots t_n$ is

$$\mathbb{E}(P, T) = (n - m + 1) \prod_{i=1}^{m} \left(\mathbb{P}(p_i) q_{p_i} + (1 - \mathbb{P}(p_i)) \right). \tag{10.2}$$

One can use the criterion $\mathbb{E}(P, T) > f$, for some $f \leq 1$, to filter out reads that are likely to match a random position, under our conditional joker sequence model.

Owing to repeats, some reads will match multiple positions in the genome, no matter how stringent the filtering which is used. In fact, it is impossible to know which of the positions is the one where the sequence originates. This phenomenon can be captured by the *mapping quality*; if a read has d equally good matches, one can assign a probability $1/d$ for each position being correct. This measure can be refined by taking into account also alignments that are almost as good as the best: see Exercise 10.1.

In Chapters 8 and 9 we learned that we can build an index for a large sequence such that exact pattern matching can be conducted in optimal time. In the read alignment problem, exact pattern matching is not enough, since we are in fact mainly interested in the *differences* between the donor and the reference. Furthermore, DNA sequencing may produce various measurement errors in the reads. Hence, a best-scoring semi-local alignment (in which an entire read matches a substring of the reference) is of interest. This was studied as the k-errors problem in Section 6.3.

In this chapter we will extend indexed exact pattern matching to *indexed approximate pattern matching*. For simplicity of exposition, we will focus on approximate string matching allowing mismatches only. Extending the algorithms to allow indels is left for the reader. In what follows, we describe several practical methods for this task, but none of them guarantees any good worst-case bounds; see the literature section on other theoretical approaches with proven complexity bounds.

10.1 Pattern partitioning

Assume we want to search for a read $P = p_1 p_2 \cdots p_m$ in a reference sequence $T = t_1 t_2 \cdots t_n$, allowing k errors.

Pattern partitioning exploits the pigeonhole principle, under the form of the following observation.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

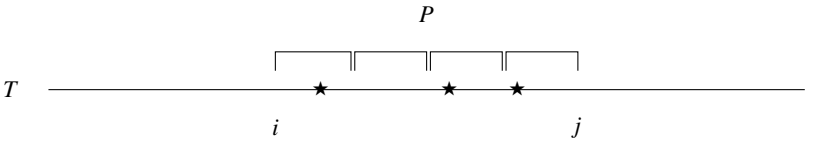

Figure 10.1 Pigeonhole principle in pattern partitioning. Here the pattern matches with three-errors whose positions in T are marked with ★ and partitioning of the pattern into four pieces is necessary and sufficient to ensure that one of the pieces does not contain an error.

The pigeonhole principle. If the pattern P is partitioned into $k+1$ pieces $P = P^1P^2 \cdots P^{k+1}$, then one of the pieces must occur exactly (without errors) inside $T_{i..j}$, if $T_{i..j}$ is a k-error occurrence of P in T.

This observation follows from the fact that if all pieces $P^1 \cdots P^k$ have an occurrence with one or more errors in $T_{i..j}$, then they can form only an occurrence with at least $k+1$ errors of P: see Figure 10.1.

Notice that the existence of an exact occurrence of a piece of P does not guarantee that there would be a k-error occurrence for P; the other pieces might not even occur one next to the other. Thus, pattern partitioning can be used only as a *lossless filter*. The final method needs to find the exact occurrences of each P^i, and then *verify* whether there is a k-error occurrence of P overlapping this exact occurrence position.

Since the sequence T is static, we can afford to build any of the indexes from the previous chapters on it. Then, we search for all the pieces P^i exactly. Using, for example, a succinct suffix array with a sampling factor $r = \log^{1+\epsilon} n / \log \sigma$ (for some constant $\epsilon > 0$), these searches and verifications take $O(m \log \sigma + (\log^{1+\epsilon} n + \lceil m/w \rceil m)\texttt{cand})$ time, where `cand` is the number of *candidate* occurrences reported by the filter. The multiplicative factor comes from extracting each candidate position from the succinct suffix array, and verifying each of the occurrences in $O(\lceil m/w \rceil m)$ time, assuming that Myers' bitparallel algorithm from Section 6.1.3 is used for verification.

Pattern partitioning does not give good worst-case guarantees for approximate searches, and the expected case behavior is limited to small error levels; see Insight 10.2. For this reason, one could also partition the pattern into fewer pieces, but then one needs to search each piece allowing some errors: see Exercise 10.2. These approximate searches can be done as explained in the next section.

Insight 10.2 Average case bound for pattern partitioning

Obviously, pattern partitioning makes sense only if the number of candidates is not much bigger than the amount of actual occurrences. On a random sequence T it makes sense to partition a pattern $P[1..m]$ into as equally sized pieces as possible. Assuming a threshold k on the number of errors and a partitioning of the pattern into $k+1$ pieces of length at least d, the probability that a given piece of the pattern matches at a given position in the text is at most $1/\sigma^d$. Then the expected number

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

of random matches between pieces of the pattern and positions of the text will be roughly $\texttt{cand} = (k+1)n/\sigma^d \leq mn/\sigma^d$. Since the time needed to check every random match is $O(m)$ (assuming we allow only mismatches), we may wish to have $\texttt{cand} = O(1)$ in order for the expected checking time of those occurrences to be $O(m)$. Equal partitioning means (in our discrete world) a mixture of pieces of length $d = \lfloor m/(k+1) \rfloor$ and pieces of length $d+1$, from which it follows than that filter is expected to work well for

$$k < \frac{m}{\log_\sigma(mn)} - 1. \tag{10.3}$$

A corollary is that, for small enough k, a suffix tree built in $O(n)$ time can be used as a data structure for the k-errors problem such that searching for a pattern takes $O(m \log \sigma + (\texttt{cand}+\texttt{occ})m)$ time. Since the expected number of candidates is $O(1)$, the expected search time simplifies to $O(m(\log \sigma + \texttt{occ}))$. In order for the checking time per occurrence to be $O(m)$, one needs to avoid checking the same candidate position more than once (since the same candidate can match more than one piece). A simple solution for the problem is to use a bitvector of size n in which the positions of the candidate positions are marked. The stated time bound assumes mismatches, but a slightly worse bound can also be achieved when one allows both indels and mismatches. Allowing indels will increase the time needed to check occurrences, but in practice, for reasonable values of m, a significant speedup can be achieved using Myers' bitparallel algorithm.

10.2 Dynamic programming along suffix tree paths

We will now focus on the problem of an approximate search for a pattern P, using an index built on a text T.

Consider the suffix tree of the text T. Each path from the root to a leaf spells a suffix of T. Hence, a semi-local alignment of a read to each suffix of T can be done just by filling a dynamic programming matrix along each path up to depth $2m$. The bound $2m$ comes from the fact that the optimal alignment must be shorter than that obtained by deleting a substring of T of length m and inserting P. The work along prefixes of paths shared by many suffixes can be done only once: the already-computed columns of the dynamic programming matrix can be stored along the edges of the suffix tree while traversing it by a depth-first search. This simple scheme is illustrated in Figure 10.2, and Exercise 10.3 asks the reader to develop the details.

10.3 Backtracking on BWT indexes

The problem with our approach so far is that the suffix tree of a whole genome sequence requires a rather large amount of space. Luckily, just like with exact pattern matching,

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

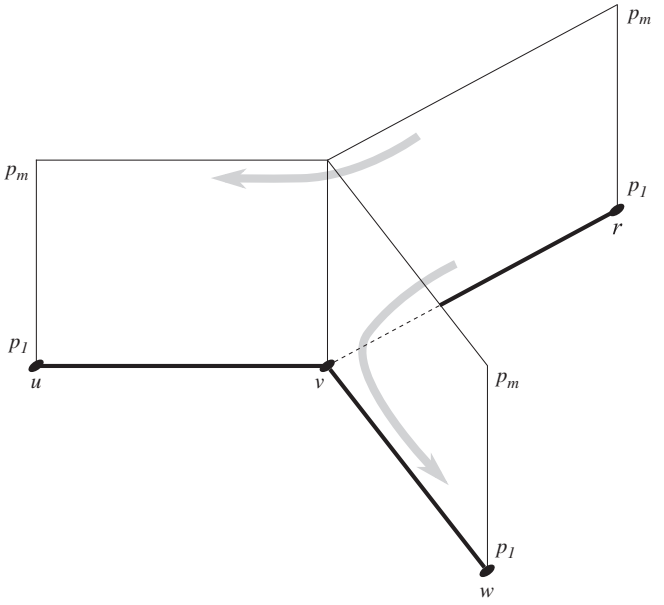

Figure 10.2 Dynamic programming along suffix tree paths.

there is a way to arrange the search with the more space-efficient Burrows–Wheeler indexes instead.

Consider the *backtracking* algorithm given in Algorithm 10.1 that uses the notions from Definition 9.12 at page 171.

Algorithm 10.1: Backtracking with the BWT index

Input: Forward BWT index `idx` of a sequence T.
Output: Search space states.

```
1 def Branch(d,[i..j]):
2     for c ∈ idx.enumerateLeft(i,j) do
3         Print (c, d);
4         Branch(d + 1,idx.extendLeft(c, [i..j]));
5 Branch(0,[1..n]);
```

Assume (by induction) that $[i..j]$ corresponds to a prefix α of length d of one or more suffixes of $\underline{T}$ in Algorithm 10.1. The operation `idx.enumerateLeft`(i,j) gives all those c such that αc is a prefix of length $d+1$ of one or more suffixes of $\underline{T}$.

Observation. Backtracking on the succinct suffix array (the forward BWT index) of a sequence T explores the same search space as a depth-first traversal of the suffix tree of $\underline{T}$.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

This means that we can almost verbatim implement dynamic programming along suffix tree paths by using backtracking on the BWT index, by reading the pattern backwards.

10.3.1 Prefix pruning

Dynamic programming along backtracking paths is nevertheless time consuming, and one needs to consider ways to *prune* the search space in order to obtain efficient algorithms in practice. Exercise 10.4 asks the reader to consider pruning under k-errors search.

For simplicity of exposition, we will now focus on the formulation of the k-mismatches problem as the problem of finding the suffixes of T that start with some string P', such that the Hamming distance between P and P' is at most k (formally, $D_{\mathsf{H}}(P, P') \leq k$). The basic algorithm using backtracking on the BWT index is given in Algorithm 10.2. That algorithm prunes branches of the search tree as soon as the Hamming distance threshold is reached.

Algorithm 10.2: Backtracking with the BWT index for solving the k-mismatches problem

Input: Forward BWT index `idx` of a sequence T, pattern $P = p_1 p_2 \cdots p_m$, and error threshold k.

Output: Ranges of $\mathsf{BWT}(T)$ corresponding to the k-mismatch occurrences of P in T.

```
1  def Branch(d,k,[i..j]):
2      for c ∈ idx.enumerateLeft(i,j) do
3          if p_d ≠ c then
4              k ← k − 1;
5          if k ≥ 0 then
6              if d = 1 then
7                  Print idx.extendLeft(c,[i..j]);
8              else
9                  Branch(d − 1,k,idx.extendLeft(c,[i..j]));
10 Branch(m,k,[1..n]);
```

It is possible to do a more effective *prefix pruning* by some preprocessing: precompute, for each prefix $P_{1..i}$ of the read, its partitioning into the maximum number of pieces such that no piece occurs in the text. Let us denote the maximum number of splits $\kappa(i)$. The value $\kappa(i)$ works as a lower bound for the number of mismatches that must be allowed in any approximate match for the prefix $P_{1..i}$: see Exercise 10.12. This estimate can be used to prune the search space as shown in Algorithm 10.3.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

Algorithm 10.3: Prefix pruning backtracking with the BWT index for solving the k-mismatches problem

Input: Forward BWT index `idx` of a sequence T, pattern $P = p_1 p_2 \cdots p_m$, error threshold k, and prefix pruning function $\kappa()$.

Output: Ranges of $\mathsf{BWT}(T)$ corresponding to k-mismatch occurrences of P in T.

```
1  def Branch(d,k,[i..j]):
2      for c ∈ idx.enumerateLeft(i,j) do
3          if p_d ≠ c then
4              k ← k − 1;
5          if k − κ(d − 1) ≥ 0 then
6              if d = 1 then
7                  Print idx.extendLeft(c,[i..j]);
8              else
9                  Branch(d − 1,k,idx.extendLeft(c,[i..j]));
10 Branch(m,k,[1..n]);
```

The computation of $\kappa(i)$ for all prefixes is analogous to the backward search algorithm (Algorithm 9.1 on page 161) applied to the reverse of a pattern on a reverse BWT index: the backward search on the reverse BWT index is applied as long as the interval $[sp..ep]$ does not become empty. Then the process is repeated with the remaining suffix of the pattern until the whole pattern has been processed. In detail, if the backward step from $P_{1..i}$ to $P_{1..i+1}$ results in a non-empty interval $[sp..ep]$ with $sp \leq ep$, then $\kappa(i+1) = \kappa(i)$. Otherwise, $\kappa(i+1) = \kappa(i)$ and $\kappa(i+2) = \kappa(i+1) + 1$, and the backward search is started from the beginning with $P_{i+2..m}$ as pattern, and so on. Thus, all $\kappa(\cdot)$ values can be computed in $O(m \log \sigma)$ time.

Insight 10.3 discusses a different pruning strategy, based on a particular hash function.

Insight 10.3 Pruning by hashing and a BWT-based implementation

For a random text T, an alternative way of reducing the search space is the following one. Suppose we want to solve the k-mismatches search on strings of fixed length m, with $k = O(\log_\sigma n)$ and σ a power of 2. Suppose that we have indexed all the m-mers of the text. Then a naive search strategy for a pattern $P \in \Sigma^m$ is to generate the Hamming ball of distance k, $H_k(P) = \{X \mid D_\mathsf{H}(X,P) \leq k\}$, around the pattern P. For every $X \in H_k(P)$ we search for X in T, which generates $\texttt{cand} = \sum_{i=0}^{k} (\sigma - 1)^i \binom{m}{i}$ candidates. The search space can be reduced as follows. Let $B(a)$ be a function that returns a binary representation of $\log \sigma$ bits for any character $a \in \Sigma$. We assume that $B(X)$ applied to a string $X \in \Sigma^m$ is the concatenation $B(X[1]) \cdot B(X[2]) \cdot \cdots \cdot B(X[m])$ of length $m \log \sigma$. Consider the hash function $F : \Sigma^m \to \Sigma^h$ defined as

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

$$F(P) = B^{-1}\left(\left(\bigoplus_{i=1}^{\lceil m/h\rceil-1} B(P[(i-1)h+1..ih])\right) \oplus B(P[m-h+1..m])\right),$$

where $x \oplus y$ denotes the standard bit-wise xor operator between binary strings.

It can be proved that the Hamming ball $H_{k'}(F(P))$ of distance $k' \leq 2k$ around the binary string $F(P)$ will be such that $F(H_k(P)) \subseteq H_{k'}(F(P)))$, namely, for any $X \in \Sigma^m$ such that $D_{\mathsf{H}}(P,X) \leq k$, it also holds that $D_{\mathsf{H}}(F(P),F(X)) \leq k'$. This property guarantees that, instead of searching in T for all positions i such that $T[i..i+m-1] \in H_k(P)$, we can search for all positions i such that $F(T[i..i+m-1]) \in H_{k'}(F(P))$, and, for each such i, check whether $D_{\mathsf{H}}(T[i..i+m-1],P) \leq k$. The number of candidates is now reduced to $\texttt{cand}' = \sum_{i=0}^{k'}(\sigma-1)^i\binom{h}{i}$. Setting $h = \Theta(\log_\sigma n)$ ensures that the expected number of random matches of the candidates in T (false positives) is $O(1)$. If σ and k are constants, then $h = O(\log n)$ and $\texttt{cand}' = O(\log^c n)$, for some constant c that depends only on k and σ. A naive way to implement the index consists of storing in a hash table all the $n-m+1$ pairs $(F(T[i..i+m]),i)$ for $i \in [1..n-m+1]$. This would, however, use space $O(n\log n)$ bits. There exists an elegant way to reduce the space to $O(n\log\sigma)$ bits. The idea is to store a string T' of length $n-m+h$, where

$$T'[i] = B^{-1}\left(\left(\bigoplus_{j=0}^{\lceil m/h\rceil-2} B(T[i+jh])\right) \oplus B(T[i+m-h])\right)$$

for all $i \in [1..n-m+h]$. It can then be easily seen that $T'[i..i+h-1] = F[T[i..i+m-1]]$. Thus, searching in T for all positions i such that $F(T[i..i+m-1]) \in H_{k'}(F(P))$ amounts to searching for all exact occurrences of $X \in H_{k'}(F(P))$ as a substring in the text T'. See Figure 10.3. One can store T' as a succinct suffix array with sampling factor $r = (\log n)^{1+\epsilon}/\log\sigma$, for some constant $\epsilon > 0$, and obtain a data structure that uses $n\log\sigma(1+o(1))$ bits of space and allows locate queries in $O(\log^{1+\epsilon} n)$ time. The original text is kept in plain form, occupying $n\log\sigma$ bits of space. Each of the $\texttt{cand}'$ candidate hash values from $H_{k'}(F(P))$ can be searched in $O(m\log\sigma)$ time, their corresponding matching positions in T' (if any) extracted in $O(\log^{1+\epsilon} n)$ time per position, and finally their corresponding candidate occurrences in T verified in $O(m)$ time per candidate.

Overall, a k-mismatches search is executed in expected time $O(\texttt{cand}' \cdot \log n + \texttt{occ}\cdot(\log^{1+\epsilon} n + m)) = O(\log^{c+1} n + \texttt{occ}\cdot(\log^{1+\epsilon} n + m))$, where c is some constant that depends only on k and σ.

10.3.2 Case analysis pruning with the bidirectional BWT index

A search strategy closely related to pattern partitioning is to consider separately all possible distributions of the k mismatches to the $k+1$ pieces, and to perform backtracking for the whole pattern for each case, either from the forward or from the reverse BWT

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

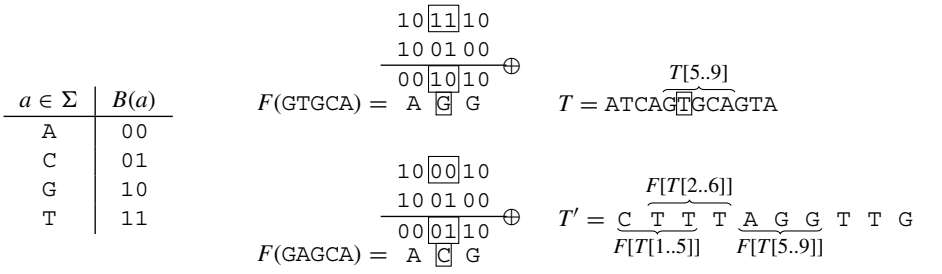

Figure 10.3 Illustrating the idea from Insight 10.3. We have chosen $h = 3$. The pattern $P = \texttt{GAGCA}$ matches the substring $T[5..9]$ with Hamming distance 1. We have $D_{\mathsf{H}}(F(P), F(T[5..9])) = 1$.

index, depending on which one is likely to prune the search space better. Let us call this strategy *case analysis pruning*.

To see how case analysis pruning works, let us first consider the simplest case $k = 1$. The 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 BWT index, since β must appear exactly, and branching is needed only after reading the last $|\beta|$ symbols of P. In case (b), it is affordable to search for $\underline{P} = \underline{\beta\alpha}$ using backward backtracking on the reverse BWT index, since $\underline{\alpha}$ must appear exactly, and branching is needed only after reading the first $|\alpha|$ symbols of P. For obvious reasons, $|\alpha| \approx |\beta|$ is a good choice for an efficient pruning.

Let us then consider $k = 2$ to see the limitations of the approach. The different ways of distributing 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 BWT index and in cases (c) and (f) backtracking on the forward BWT index. For cases (b) and (e) neither choice is good or bad a priori. More generally, 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 the case $k = 1$.

This bottleneck can be alleviated using *bidirectional case analysis pruning*. For example, for the case 101 above, it is possible to search, say, with the forward BWT index, two first blocks backwards, an operation denoted $\overleftarrow{10}1$, and then to continue the search from the reverse BWT index with the third block, an operation denoted $10\overrightarrow{1}$. For this, we need the synchronization provided by the bidirectional BWT: once an interval $[i..j]$ has been found to match a prefix of the pattern with the fixed distribution of errors using `idx.extendLeft`$(c, [i..j], [i'..j'])$, an operation analogous to the backtracking in Algorithm 10.2, one can continue the search from the original start position with the so-far-obtained interval $[i', j']$ using `idx.extendRight`$(c, [i..j], [i'..j'])$ instead. Exercise 10.14 asks you to give the pseudocode of this algorithm.

10.4 Suffix filtering for approximate overlaps

In the absence of a reference sequence to align the reads to, one could try to align the reads to themselves in order to find overlaps, which could later be used for de novo

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

fragment assembly. We studied the computation of exact overlaps in Section 8.4.4, but, due to measurement errors, computing approximate overlaps is also of interest.

Let use first describe an approach based on a *suffix filtering* designed for indexed approximate pattern matching. Let the pattern P be partitioned into pieces $P = P^1P^2 \cdots P^{k+1}$. Then consider the set of suffixes $\mathcal{S} = \{P^1P^2 \cdots P^{k+1}, P^2P^3 \cdots P^{k+1}, \ldots, P^{k+1}\}$. Each $S \in \mathcal{S}$ is searched for in T, allowing zero errors before reaching the end of the first piece in S, one error before reaching the end of the second piece of S, and so on. Obviously this search can be done using backtracking on the reverse BWT index. This filter produces candidate occurrences that need to be verified. It produces fewer candidates than the pattern partition filter from Section 10.1, but it is nevertheless a lossless filter: see Exercise 10.15.

Let us now modify this approach for detecting approximate overlaps among a set of reads $\mathcal{R} = \{R^1, R^2, \ldots, R^d\}$. Consider the detection of all the suffix–prefix overlaps (R^i, R^j, o^{ij}) such that $D_{\mathsf{H}}(R^i_{|R^i|-o^{ij}+1..|R^i|}, R^j_{1..o^{ij}})/o^{ij} \leq \alpha$, where α is a relative error threshold, and $o^{ij} \geq \tau$ for some minimum overlap threshold τ.

Now, construct a reverse BWT index on $T = R^1\#^1R^2\#^2 \cdots R^d\#^d\#$. Consider some $P = R^i$. First, fix some overlap length o, and partition the prefix $P_{1..o}$ into pieces $P^1P^2 \cdots P^{k+1}$, for $k = \lfloor \alpha o \rfloor$. Suffix filtering could be executed as such, just leaving out candidates that are not followed by any $\#^j$ in T. Repeating this on all possible overlap lengths is, however, extremely ineffective, so we should find a way to partition the whole read so that the suffix filtering works correctly for any overlap length. However, any fixed partitioning will be bad for overlap lengths that cross a piece boundary only slightly. For this, we can use two partitionings that have maximal minimum distance between mutual piece boundaries; such equal partitioning is shown in Figure 10.4. The two suffix filters are executed independently, and overlaps are reported only after reaching the middle point of the current piece: the observation is that any proper overlap will be reported by one of the filters.

It remains to consider the maximum piece length in equal partitioning such that the suffix filters in Figure 10.4 work correctly for all overlap lengths. Such a length h is

$$h = \min_{1 \leq o \leq |P|} \lfloor o/(\lfloor \alpha o \rfloor + 1) \rfloor. \tag{10.4}$$

That is, pieces $P^1, \ldots, P^k$ are of length h and piece P^{k+1} is of length $|P| - kh$. The shifted filter in Figure 10.4 has then piece P^1 of length $h/2$, pieces $P^2, \ldots, P^k$ of length h, and piece P^{k+1} of length $|P| - kh + h/2$.

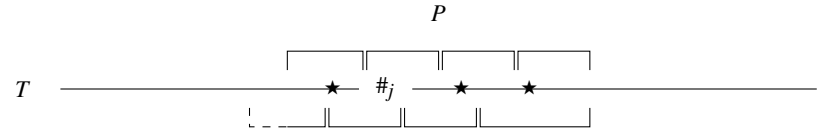

Figure 10.4 Two partitionings enabling suffix filtering to be used for approximate overlap computation. Searching with the second piece in the bottom-most partitioning yields an overlap with the read R^j.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

10.5 Paired-end and mate pair reads

Many sequencing techniques provide *paired-end* and *mate pair* reads. A paired-end read is a suffix–prefix pair of a fragment of DNA, with read length smaller than the fragment length:

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

Mate pair reads are similar to paired-end reads but are produced with a different sequencing technique. The most notable difference from the analysis perspective is that the fragment corresponding to a mate pair can be longer. Also, depending on the sequencing technique, the two ends can come from different strands, and can be oriented in any of the four different relative orders. We may assume that the pairs are normalized to the forward strand and forward reading direction.

Read alignment on these read pairs can be done independently, and often the purpose is to detect larger-scale variants: see Exercise 10.18.

Another purpose is to detect the correct alignment in cases where one of the pairs is inside a repetitive region; among the many occurrences of one of the reads, one can select the one within fragment length distance of the (possibly) unique occurrence of its pair. Often, this selection is done through a cross-product of occurrence lists. However, it can be done more efficiently, as we discuss next.

Algorithm 10.4: Prefix pruning backtracking with the BWT index for solving the k-mismatches problem inside a given range of T

Input: Forward BWT index `idx` of a sequence T, wavelet tree $\mathcal{T}$ of $\mathsf{SA}(T)$, pattern $P = p_1 p_2 \cdots p_m$, error threshold k, prefix pruning function $\kappa(\cdot)$, and query interval $[l..r]$ in T.

Output: Ranges of $\mathsf{BWT}(T)$ corresponding to k-mismatch occurrences of P in $T_{l..r}$.

```
1  def Branch(d,k,[i..j]):
2      for c ∈ idx.enumerateLeft(i,j) do
3          if p_d ≠ c then
4              k ← k − 1;
5          if k − κ(d − 1) ≥ 0 then
6              [i'..j'] = idx.extendLeft(c,[i..j]);
7              if T.rangeCount(i',j',l,r) > 0 then
8                  if d = 1 then
9                      Print [i'..j'];
10                 else
11                     Branch(d − 1,k,idx.extendLeft(c,[i..j]));
12 Branch(m,k,[1..n]);
```

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

Recall the query $\texttt{rangeCount}(T, i, j, l, r)$ on wavelet trees (Corollary 3.5 on page 27). By building the wavelet tree on the suffix array SA of T instead, one can support the query $\texttt{rangeCount}(\mathsf{SA}, i, j, l, r)$. This can be used to check whether there are any suffix starting positions among the values $\mathsf{SA}[i..j]$ within the text fragment $T[l..r]$. With this information, one can modify any of the search algorithms considered in this chapter to target a genome range: Algorithm 10.4 gives an example.

The shortcoming of this improvement to paired-end alignment is that the wavelet tree of a suffix array requires $n \log n(1 + o(1))$ bits, where so far in this chapter we have used $O(n \log \sigma)$ bits, with a very small constant factor.

10.6 Split alignment of reads

In RNA-sequencing the reads represent fragments of messenger RNA. In higher organisms, such fragments may originate from different combinations of exons of the genome, as shown in Figure 10.5.

Assuming that we have the complete *transcriptome*, that is, all possible RNA transcripts that can exist in nature, the RNA-sequencing reads could be aligned directly to the transcriptome. This leads to a problem identical to the one of aligning DNA reads to a genome. Although databases approximating the transcriptome exist for widely studied organisms, it makes sense to ascertain whether an RNA-sequencing read could directly be aligned to the genome. This is even more relevant for studying various genetic diseases, in which the cell produces novel specific RNA transcripts. This leads us to the *split-read alignment* problem, which we also studied in Section 6.5 under the name *gene alignment with limited introns*. The alignment algorithm given there is too slow for high-throughput purposes, so we will now consider an indexed solution.

It turns out that there is no obvious clean algorithmic solution for split-read alignment. Insight 10.4 discusses an approach based on pattern partitioning and Insight 10.5 discusses an approach based on analysis of the whole read set simultaneously. We will come back to this problem in Section 15.4 with a third approach that is algorithmically clean, but it assumes a preprocessed input, and can thus lose information in that phase.

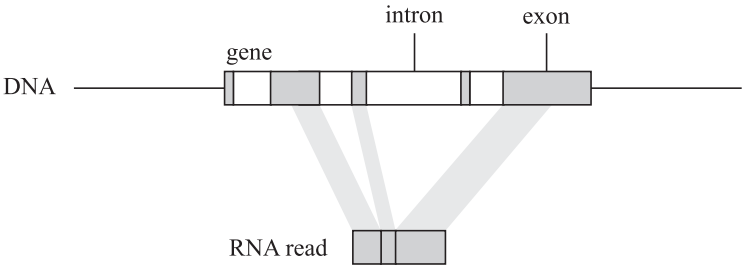

Figure 10.5 An RNA-sequencing read aligned to the genome.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

Insight 10.4 Split-read alignment through pattern partitioning

Consider first the case when a read is aligned to the genome with one split. Then we know that at least half of the read should align to a contiguous region in the genome. With a large enough read length, we should not expect too many alignments for a half-read, so such alignments can be efficiently retrieved using the methods discussed earlier in this chapter. Then, we can extend the alignment of the half-read towards the actual split point, and search for the remaining head or tail. This can be done like in the paired-end alignment, by using Algorithm 10.4 with an estimate on the maximal possible intron length for forming the search region.

For longer reads containing more split points, we can keep splitting into more and more parts until we find a partitioning into s parts such that one of the pieces is successfully aligned to the genome. This piece, once extended to the left and to the right while maintaining a decent alignment, can be cut off, and the head and tail searched for recursively with the same approach.

The problem of this approach is that the pieces are becoming smaller and smaller, and at some point they are likely to occur in a random sequence. However, this position-restricted search helps somewhat, insofar as a random hit in a smaller region is less likely (see the exercises).

Insight 10.5 Split-read alignment through mutual information

Another approach to split-read alignment is to first consider the reads that map without any split, predict the exon boundaries from these initial alignments, and then align the rest of the reads such that they are allowed to jump from the end of one predicted exon to the start of a nearby predicted exon.

Consider first that an alignment needs to use only one split. Such 1-split alignments can be conducted efficiently, for example, by concatenating all pairs $\alpha\beta$ such that α is extracted from the end of some exon i and β is extracted from the start of some nearby exon j following i. The set of all these concatenated sequences can be indexed using BWT indexes, and now all unmapped reads can be aligned to this set. Choosing $|\alpha|$ and $|\beta|$ greater than the maximum read length and using some bookkeeping, one should be able to find most of the 1-split alignments, assuming that the read coverage is high enough that the exon prediction is accurate enough.

For longer reads containing more split points, we could extend the above approach by considering several exon combinations at once, but this would lead to a combinatorial explosion. There is a more principled approach that may avoid this explosion. Consider the genome as a labeled DAG with an arc from each position i to position $i + 1$, and from each i to all j, if i is an end position of an exon and j is the start position of a nearby exon following i. We can then build the BWT index of Section 9.6 on this labeled DAG and align unmapped reads on it; see Section 10.7 for how to conduct this alignment.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

10.7 Alignment of reads to a pan-genome

So far we have assumed that we have only one reference genome to align all the reads to. For many species, the population-wide variation has been widely explored and there are databases containing variations common in (sub-)populations. One can also consider sets of individual genomes that have been assembled by read alignment or/and (partial) de novo assembly. Read alignment on such *pan-genomic* datasets is of interest, since one can hope to improve alignment accuracy by aligning directly to known variation; in this way, one can reduce the work required for variant calling (see Chapter 14) and improve its accuracy.

10.7.1 Indexing a set of individual genomes

Given a set of individual genomes from the same species, one can naturally build one of the indexes studied in this chapter for each genome, and then apply read alignment in each of them. A better approach for both compression and query efficiency is to concatenate all the genomes and build only one index. The concatenation is highly repetitive, so if the index uses internal compression, a significant amount of space can be saved. Such indexes exist, but we shall explore a more tailored approach that appears to be more efficient in practice.

Assume $T = t_1 t_2 \cdots t_n$ is the concatenation of individual genomes from a species. Apply Lempel–Ziv parsing on T (see Section 12.1) to obtain a partitioning $T = T^1 T^2 \cdots T^p$. Extract a set of *patches* around each phrase boundary to form the set

$$B = \left\{ T_{j-r-k..j+r+k} \mid j \in \{1\} \cup \left\{ 1 + \sum_{1 \le i \le p'} |T^i| \ : \ 1 \le p' \le p \right\} \right\}. \tag{10.5}$$

Concatenating B into a sequence with some special symbols marking the boundaries of consecutive patches, one can build any index for B that allows one to perform read alignment with k errors for reads of length at most r. With some bookkeeping data structures, such alignment positions can be mapped back to the original sequence: see Exercise 10.19.

LEMMA 10.1 *A pattern $P = p_1 \cdots p_m$, $m \le r$, has a k-errors occurrence in $T = t_1 \cdots t_n$, if and only if it has a k-errors occurrence in the set B defined in Equation (10.5).*

Proof The direction from B to T is obvious from construction. Consider now an occurrence inside some T^i of the Lempel–Ziv partitioning $T = T^1 T^2 \cdots T^p$ such that the occurrence is not included in the corresponding boundary patches. This means that the occurrence substring is an exact copy of some substring occurring earlier in T. If this exact copy is not included inside the corresponding patches, it also must be an exact copy of some substring occurring earlier in T. Continuing this induction, one notices that there must be an exact copy of the occurrence overlapping a phrase boundary, which means that it is included in a patch in B. □

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

The first occurrence found in the above proof is called the *primary occurrence* and the others are called *secondary occurrences*. Exercise 10.20 asks you to show how the wavelet tree can be used for reporting the secondary occurrences.

*10.7.2 Indexing a reference genome and a set of variations

Another approach to pan-genome indexing is to represent the reference and the known variants as a labeled DAG, as illustrated in Figure 10.6.

Then one can build the BWT index of Section 9.6 on this labeled DAG. The main difference of this approach from the one indexing a set of genomes is that now arbitrary (re)combinations of the underlying variants are allowed. That is, the labeled DAG represents all individuals that can be formed by combining the underlying variants.

In Section 9.6 we learned how to do exact pattern matching using the BWT index of a labeled DAG. Supporting basic backtracking is analogous to the corresponding algorithms on standard BWT indexes, but the various pruning techniques studied earlier are not easy to extend: there appears to be no bidirectional analog of the BWT index on a labeled DAG. Prefix pruning seems to be the only method that applies: Exercise 10.22 asks you how to compute values $\kappa(i)$ in this case. Assuming that these values have been computed, one can apply Algorithm 10.3 verbatim by just switching the index used. This algorithm returns intervals in BWT_G of the labeled DAG G, and one can then obtain the vertices of the DAG that have a path starting inside those intervals. Extracting the full alignments, that is, all the paths that match the pattern, is a nontrivial procedure, which we study next.

Let us assume that we have kept the *backtracking tree* $\mathcal{T}$ whose nodes are intervals in BWT_G, and whose edges are the backward steps taken from the parent intervals towards child intervals. That is, each edge e of $\mathcal{T}$ is labeled with a character $\ell(e)$. We shall explore how to extract the prefix of the lexicographically smallest path matching $P_{1..m}$. That is, we study the extraction of the first entry in the interval $\overrightarrow{u}$, where $\overrightarrow{u}$ is the left-most interval found during backtracking, and thus u is a leaf of $\mathcal{T}$. First, we need to find the vertex v of G whose interval $\overrightarrow{v}$ of BWT_G is the first belonging also to $\overrightarrow{u}$. This is given by

$$\overrightarrow{v} = [\mathtt{select}_1(\mathtt{out}, \mathtt{rank}(\mathtt{out}, \overrightarrow{u}))..\mathtt{select}_1(\mathtt{out}, \mathtt{rank}(\mathtt{out}, \overrightarrow{u}) + 1) - 1], \tag{10.6}$$

where out is the bitvector encoding the out-neighbors in the Burrows–Wheeler index of G of Definition 9.19. Knowing now $\overrightarrow{v}$, we want to move along arc (v, w) in G such that $\ell(w) = \ell((u, p))$, where p is the parent of u in $\mathcal{T}$, and w is the out-neighbor of v

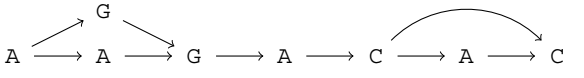

Figure 10.6 A labeled DAG representing a reference and a set of variants. Here the reference is AAGACAC and the set of variants consists of the second A substituted by G and the last A deleted.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

with the first interval $\overleftarrow{w}$ included in $\overleftarrow{p}$. Recall that we considered how to find the interval of the ith out-neighbor of v in Section 9.6: see Equation (9.8). We can now carry out a binary search over all out-neighbor intervals to find the vertex w with the smallest interval included in $\overleftarrow{p}$. Continuing this way up in the backtracking tree $\mathcal{T}$, we can list one of the paths matching P. Exercise 10.23 asks you to extend this algorithm to list all such paths.

10.8 Literature

Pattern partitioning and more advanced filters for approximate pattern matching are surveyed in Navarro (2001); suffix filtering is a later development by Kärkkäinen & Na (2007).

Finding theoretical solutions to indexed approximate pattern matching was the Holy Grail of pattern matching for a long time, until in Cole *et al.* (2004) an index with optimal running time was found (assuming many parameters as constants). Their index requires superlinear space, making it less useful for genome-scale sequences. This bottleneck was soon overcome in Chan *et al.* (2006); they provide an $O(n \log n)$ bits index with running time $O(m + \mathtt{occ} + \text{polylog}(n))$, where m is the length of the pattern and $\mathtt{occ}$ is the number of occurrences with up to k-errors, with k and the alphabet size σ assumed to be constants. The space can be further decreased to $O(n \log \sigma)$ bits by using a compressed suffix tree as part of the index. However, the result works only for very small values of k and σ, since the $O(\cdot)$ space terms hide 3^k, σ^k, and $\log^{k^2} n$ factors. In read alignment, the sequences are quite accurate and allowing only a few errors usually suffices in the search.

As far as we know, the above theoretical approaches have not been explored in practice; the tools used in read alignment are mostly based on backtracking on BWT indexes or k-mer indexes. Using dynamic programming on top of BWT indexes for read alignment was first proposed in Lam *et al.* (2008). Prefix pruning is from Li & Durbin (2009). The forward version of case analysis pruning is from Langmead *et al.* (2009) and its bidirectional improvement is from Li *et al.* (2009). The idea in Insight 10.3 is from Policriti & Prezza (2014a) and Policriti & Prezza (2014b), which adapt a hash function initially considered in Policriti *et al.* (2012).

The extension of suffix filtering to approximate overlaps was proposed in Välimäki *et al.* (2012). Better theoretical bounds were achieved in Välimäki (2012).

Split-read alignment using mutual information was proposed in Trapnell *et al.* (2009) and its extension to longer reads was experimented with in Sirén *et al.* (2014).

Pan-genome indexing for read alignment was first studied in Mäkinen *et al.* (2009, 2010) and Schneeberger *et al.* (2009); the first two use compressed variants of BWT indexes, and the last uses k-mer indexing. Theoretical advances to support linear-time exact pattern searches on arbitrary-length patterns with relative Lempel–Ziv compressed collections were achieved in Do *et al.* (2012) and Gagie *et al.* (2012); these results are significant since Lempel–Ziv compression is superior to BWT compression on highly repetitive collections, such as a set of individual genomes. Practical alternatives using

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

a hybrid of Lempel–Ziv factorization and BWT indexes were given in Wandelt *et al.* (2013) and Ferrada *et al.* (2014). Our description follows Ferrada *et al.* (2014). An alternative to these is to extract directly the context around variations and build a BWT index on these patches as in Huang *et al*. (2013). These practical approaches work only on limited pattern length. Finally, the labeled DAG approach was proposed in Sirén *et al.* (2011) and Sirén *et al.* (2014). This approach works on any pattern length, but the index itself has exponential size in the worst case; the best-case and average-case bounds are similar to those of Mäkinen *et al.* (2010). Here one should observe that identical functionality with context extraction alone would result in the best-case exponential size index on the length of the read.

In Chapter 14 we study variation calling over pan-genome indexes. We shall learn that, while the labeled DAG approach can exploit recombinations of variants already during read alignment, this lack of functionality in the approach of indexing a set of individuals can be partly alleviated by a better control over the recombination model when predicting the novel variants.

Exercises

10.1 Recall mapping quality, Insight 10.1. Consider how it could be refined to take good alignments into account in addition to only the best. How would you in practice approximate such mapping quality under, for example, a k-errors search?

10.2 Consider splitting a pattern into fewer than $k+1$ pieces. How many errors should be allowed for the pieces in order to still obtain a lossless filter for the k-errors search?

10.3 Give the pseudocode for the algorithm depicted in Figure 10.2 to compute maximum-scoring semi-local alignment along suffix tree paths.

10.4 Modify the above approach to solve the k-errors problem. Do you always need to fill the full dynamic programming matrix on each path? That is, show how to prune branches that cannot contain an occurrence even if the rest of the pattern $P_{j..m}$ exactly matches a downward path.

10.5 Modify the above approach to solve the k-errors problem using Myers' bitparallel algorithm instead of standard dynamic programming.

10.6 Modify all the above assignments to work with backtracking on the BWT index.

10.7 Most sequencing machines give the probability that the measurement was correct for each position inside the read. Let $\mathbb{P}(p_i)$ be such a probability for position i containing p_i. Denote $M[c,i] = \mathbb{P}(p_i)$ if $p_i = c$ and $M[c,i] = (1-\mathbb{P}(p_i))/(\sigma-1)$ if $c \neq p_i$. Then we have a *positional weight matrix (PWM) M* representing the read (observe that this is a profile HMM without insertion and deletion states). We say that the matrix M *occurs* in position j in a genome sequence $T = t_1 t_2 \cdots t_n$ if $\mathbb{P}(M,T,j) = \prod_{i=1}^{m} M[t_{j+i-1}, i] > t$, where t is a predefined threshold. Give the pseudocode for finding all occurrences of M in T using backtracking on the BWT index. Show how to prune branches as soon as they cannot have an occurrence, even if all the remaining positions match $P_{1..j}$ exactly.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

10.8 The goal in read alignment is typically to find the unique match if one exists. Consider how to optimize the backtracking algorithms to find faster a best alignment, rather than all alignments that satisfy a given threshold.

10.9 Some mate pair sequencing techniques work by having an adapter to which the two tails of a long DNA fragment bind, forming a circle. This circle is cut in one random place and then again X nucleotides apart from it, forming one long fragment and another shorter one (assuming that X is much smaller than the circle length). The fragments containing the adapter are fished out from the pool (together with some background noise). Then these adapters containing fragments are sequenced from both ends to form the mate pair. Because the cutting is a random process, 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?

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

10.11 Give the pseudocode for computing the values $\kappa(i)$ for prefix pruning applied on the prefixes $P_{1..i}$ of the pattern P.

10.12 Show that the values $\kappa(i)$ in prefix pruning are correct lower bounds, that is, there cannot be any occurrence missed when using the rule $k' + \kappa(i) > k$ to prune the search space.

10.13 Show that the computation of the values $\kappa(i)$ in prefix pruning is also feasible using the forward BWT index alone by simulating the suffix array binary search.

10.14 Give the pseudocode for the k-mismatches search using case analysis pruning on the bidirectional 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 the fewest errors.

10.15 Show that suffix filtering is a lossless filter for a k-mismatches search.

10.16 Compute the minimal read length m such that the expected number of occurrences of the read in a random sequence of length n when allowing k mismatches is less than $1/2$.

10.17 Compute the minimal read length m such that the expected number of occurrences of the read in a random sequence of length n when allowing k errors is less than $1/2$ (use approximation, the exact formula is difficult).

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press

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

10.19 Consider the pan-genome read alignment scheme described in Section 10.7.1. Develop the bookkeeping data structures to map alignment position from the patched sequence B to the position in T.

10.20 Consider the pan-genome read alignment scheme described in Section 10.7.1. Show how the wavelet tree can be used to retrieve secondary occurrences, each in $O(\log n)$ time.

10.21 Consider the pan-genome read alignment scheme described in Section 10.7.1 and recall Exercise 3.6. Show how to use van Emde Boas trees to retrieve secondary occurrences, each in $O(\log \log n)$ time.

10.22 Show how to compute values $\kappa(i)$ for prefix pruning for prefixes $P_{1..i}$ of the pattern P in the case of the BWT index on a labeled DAG.

10.23 Give a pseudocode for listing all the subpaths matching a pattern using the BWT index of the labeled DAG. Extend the algorithm given in Section 10.7.2 for listing the matching prefix of a path starting at vertex v with smallest lexicographic order.

https://doi.org/10.1017/CBO9781139940023.012 Published online by Cambridge University Press