

Fig. 7. Frequency distribution of 10-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

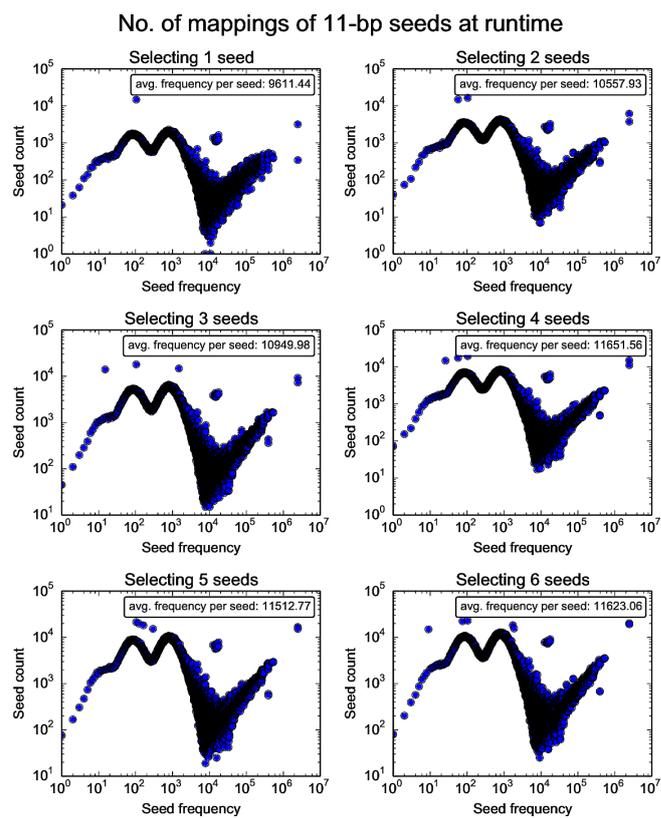


Fig. 8. Frequency distribution of 11-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

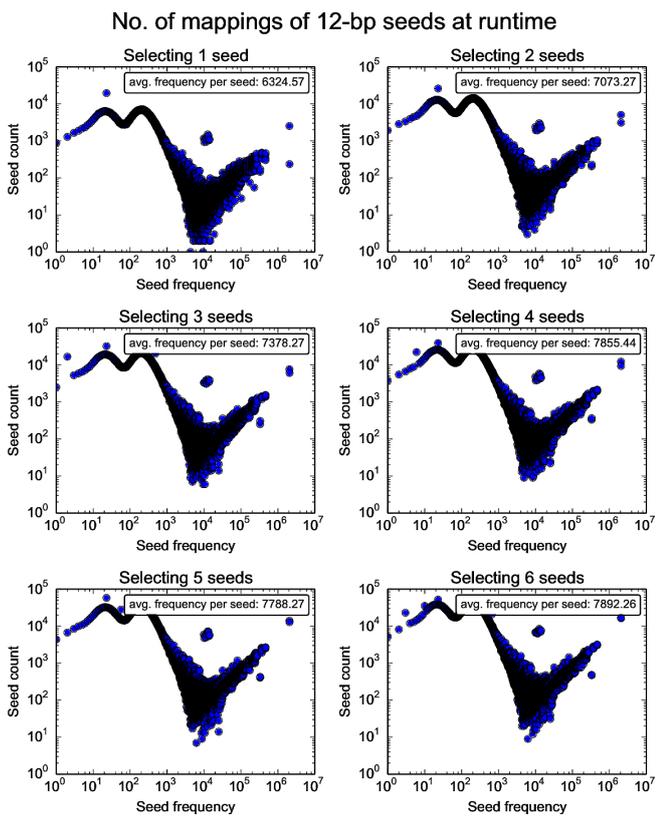


Fig. 9. Frequency distribution of 12-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

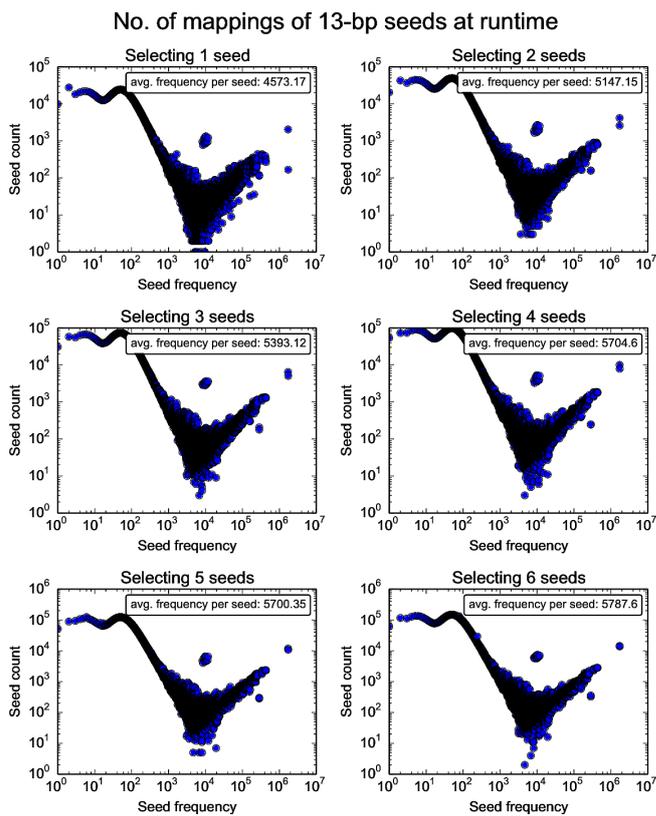


Fig. 10. Frequency distribution of 13-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

No. of mappings of 14-bp seeds at runtime

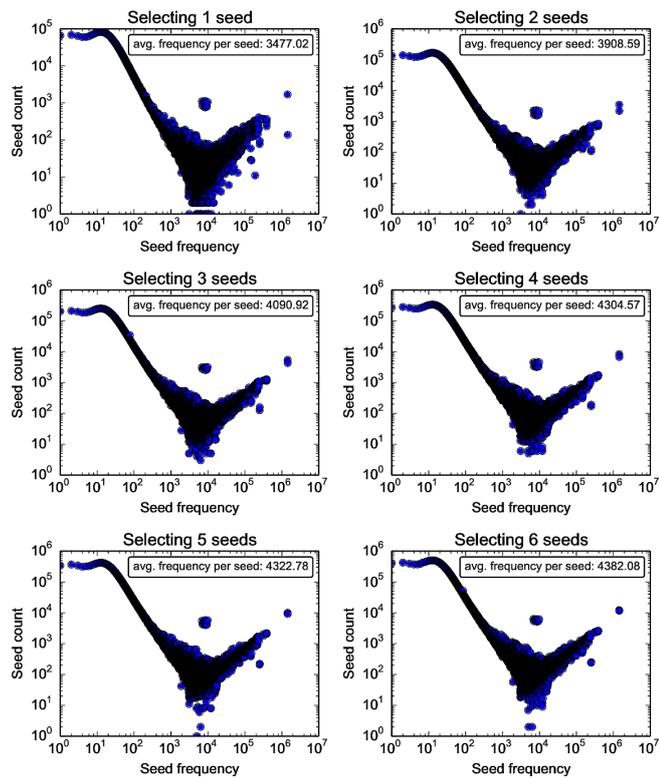


Fig. 11. Frequency distribution of 14-bp seeds at runtime by selecting seed consecutively under different number of required seeds.

## 1 SUPPLEMENTARY MATERIALS

### 1.1 Runtime frequency distributions of seeds under variant lengths

This section presents seed frequency distributions at runtime with regard to different seed lengths. The results are obtained by naively selecting different fixed-length seeds consecutively in the process of mapping 4,031,354 101-bp reads from a real read set, ERR240726, from the 1000 Genome Project.

Figure 7 to Figure 11 from page 1-3 show seed frequency distributions of fixed-length seeds from 10-bp to 14-bp. From these figures, we have three observations: (1) the average seed frequencies of longer seeds are smaller, (2) frequent seeds beyond the seed frequency of  $10^4$  are more frequently selected at runtime and (3) compared to Figure 1, the average frequencies of selected seeds are much larger than the average frequencies of seeds with equal length in the seed database.

As shown in all five figures above, after the seed frequency of  $10^4$ , the seed count increases with greater seed frequencies, which implies that frequent seeds are often selected from reads, regardless of the seed length.

### 1.2 Proof of optimal divider cascading

This section presents the detailed proof of the optimal divider cascading phenomenon.

The optimal divider cascading phenomenon can be explained with two *lemmas*:

**LEMMA 1.** *For any two prefixes from the same iteration in OSS, one prefix must include the other. Among the two prefixes, the minimum seed frequency of the outer prefix must not be greater than the minimum seed frequency of the inner prefix.*

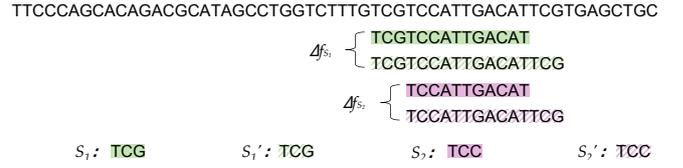
The proof of Lemma 1 is provided below:

**PROOF.** Since both are prefixes of the same read, one must include another, as shown in Figure 4.

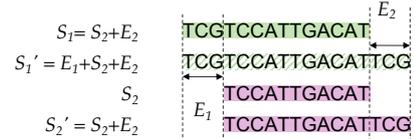
We prove the second part of the lemma by contradiction. Assume the outer prefix has a greater *optimal frequency* (total seed frequency of the optimal seeds) than the inner prefix. Because the inner prefix is included by the outer prefix, the optimal seeds of the inner prefix are also valid seeds for the outer prefix. Yet, the total frequency of this particular set of seeds is smaller than the optimal frequency of the outer prefix, which leads to a contradiction.  $\square$

**LEMMA 2.** *When extending two seeds of different lengths that end at the same position in the read by equal numbers of base-pairs, as one seed includes the other as shown in Figure 12, the frequency reduction ( $\Delta f$ ) of extending the outer seed ( $S_2 \rightarrow S'_2$ ) must not be greater than the frequency reduction of extending the inner seed ( $S_1 \rightarrow S'_1$ ).*

Lemma 2 can be proven with the monotonic non-increasing property of seed frequency with regard to a greater seed length. For example, in Figure 13, there are two seeds taken from the same read,  $S_1$  and  $S_2$ , with  $S_1$  including  $S_2$  and both end at the same position in the read. Now, we simultaneously extend both  $S_1$  and  $S_2$  longer in the read (by taking more base-pairs) by 3-bp, into  $S'_1$  and  $S'_2$  respectively. With  $\Delta f$  denoting the change of seed frequencies before and after extension, we can claim that  $\Delta f_{S_1} \leq \Delta f_{S_2}$ .



**Fig. 12.** This figure shows two seeds  $S_1$  and  $S_2$ , which are taken from the same read and end at the same position, with  $S_1$  including  $S_2$ . Both seeds are extended by 3-bp into  $S'_1$  and  $S'_2$  respectively.



**Fig. 13.** Two seeds,  $S_1$  and  $S_2$  are taken from the same read and end at the same position in the read. Both  $S_1$  and  $S_2$  are extended by 3-bp into  $S'_1$  and  $S'_2$  respectively. Considering  $S_1$  as a left-extension of  $S_2$  by  $E_1$  and  $S'_1$  as a right-extension of  $S_1$  by  $E_2$ , then we have  $S_1 = E_1 + S_2 + E_2$ .

To prove this inequality, it is essential to understand how is  $\Delta f$  calculated. As Figure 13 also shows, among the two seeds  $S_1$  and  $S_2$ ,  $S_1$  can be considered as a “left-extension” of  $S_2$ . Therefore,  $S_1$  can be represented as  $S_1 = E_1 + S_2$ , where  $E_1$  denotes the left extension of  $S_1$  and the “+” sign denotes a concatenation of strings. Similarly,  $S'_1$  can be represented as a “right-extension” of  $S_1$ , which can be also written as  $S'_1 = E_1 + S_2 + E_2$ , where  $E_2$  is the right  $m$ -bp extension of  $S_1$ . By the same token, we also have  $S'_2 = S_2 + E_2$ . If  $\text{freq}(S)$  denotes the frequency of a seed  $S$ , then  $\Delta f_{S_1} = \text{freq}(S_1) - \text{freq}(S'_1) = \text{freq}(E_1 + S_2) - \text{freq}(E_1 + S_2 + E_2)$ .

Below, we provide the proof of Lemma 2:

**PROOF.** If  $\text{set } \overline{\mathbb{E}_2}$  denotes all DNA sequences that are equal in length with  $E_2$  but excludes  $E_2$  itself, which can be written as  $\overline{\mathbb{E}_2} = \{s \mid (s \in \text{DNA sequence}) \wedge (|s| = |E_2|) \wedge (s \neq E_2)\}$ , then the reduced frequency of  $S_1$  and  $S_2$  can also be written as:

$$\Delta f_{S_1} = \sum_{s \in \overline{\mathbb{E}_2}} \text{freq}(E_1 + S_2 + s)$$

$$\Delta f_{S_2} = \sum_{s \in \overline{\mathbb{E}_2}} \text{freq}(S_2 + s)$$

The right hand side of both equations denote the sum of frequencies of all seeds that share the same beginning sequence  $E_1 + S_2$  (or just  $S_2$ ) other than the sequence  $E_1 + S_2 + E_2$  itself (or  $S_2 + E_2$  for  $S'_2$ ), which is indeed  $\text{freq}(E_1 + S_2) - \text{freq}(E_1 + S_2 + E_2)$  (or  $\text{freq}(S_2) - \text{freq}(S_2 + E_2)$  for  $S_2$ ).

From both equations, we can see that both  $\Delta f_{S_1}$  and  $\Delta f_{S_2}$  iterates through the same set of strings,  $\overline{\mathbb{E}_2}$ . For each string  $i$  in set  $\overline{\mathbb{E}_2}$ , we have  $\text{freq}(E_1 + S_2 + i) \leq \text{freq}(S_2 + i)$ , as the extended longer seed can only be less or equally frequent as the original and shorter seed. Therefore, we have  $\Delta f_{S_1} \leq \Delta f_{S_2}$ .  $\square$

From Lemma 2, we can deduce Corollary 2.1:

**COROLLARY 2.1.** *When extending two substrings of different lengths that ends at the same position in the read by equal number of*

seeds, as one substring includes the other, the frequency reduction of the **optimal seed** (the optimal single seed) of extending the longer substring, is strictly not greater than the frequency reduction of the optimal seed of extending the shorter substring.

We prove Corollary 2.1 by cases:

PROOF. Considering the four substrings from Figure 13,  $S_1$ ,  $S_2$ ,  $S'_1$  and  $S'_2$ . Among the four substrings, we have the following system of equations that describe these substring relationships:

$$\begin{cases} S_1 = E_1 + S_2; \\ S'_1 = E_1 + S_2 + E_2; \\ S'_2 = S_2 + E_2 \end{cases}$$

There are three possible cases of where the optimal seed is selected in  $S'_1$ : (1) from the region of  $S_2 + E_2$ , (2) from the region  $E_1 + S_2$  and the optimal seed overlaps with  $E_1$  and (3) from the region of  $E_1 + S_2 + E_2$  and the seed overlaps with both  $E_1$  and  $E_2$ . Below we prove that the Corollary is correct in each case.

Case 1: The optimal seed is selected exclusively from  $S_2 + E_2$ .

This suggests that the optimal seed in  $S'_1$  is also the optimal seed in  $S'_2$ . Based on Lemma 1, we know the optimal frequency of  $S_1$  is not greater than  $S_2$ .

Combining the two deductions above, we can conclude that extending  $S_2$  to  $S'_2$  provides a frequency reduction of the optimal seed that is greater than or equal to extending  $S_1$  to  $S'_1$ .

Case 2: The optimal seed is selected from the region  $E_1 + S_2$  and it overlaps with  $E_1$ .

Since the optimal seed does not overlap with  $E_2$ , the optimal seed in both  $S_1$  and  $S'_1$  must be the same. Therefore extending  $S_1$  to  $S'_1$  provides 0 frequency reduction of the optimal seed. As Lemma 1 suggests, the optimal seed frequency of  $S_2$  must not be greater than the optimal seed frequency of  $S'_2$ . As the result, the Corollary holds in this case.

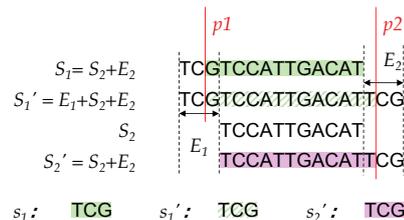
Case 3: The optimal seed is selected across  $E_1 + S_2 + E_2$  and it overlaps with both  $E_1$  and  $E_2$ .

Assuming that the optimal seed,  $s'_1$ , in  $S'_1$  starts at position  $p_1$  and ends at position  $p_2$ . Now assume a seed,  $s_1$ , which starts at  $p_1$  but ends where  $S_1$  ends, as shown in Figure 14. Also assume a seed,  $s'_2$ , which starts at where  $S'_2$  starts and ends at  $p_2$ . From Lemma 2, we know that the reduction of seed frequency of extending  $s_1$  to  $s'_1$  is no greater than the seed frequency reduction of extending  $S_2$  to  $s'_2$ . We also know that the optimal seed frequency of  $S_1$  is no greater than the seed frequency of  $s_1$  and the optimal seed frequency of  $S'_2$  is no greater than the seed frequency of  $s'_2$ . As a result, the frequency reduction of the optimal seed by extending  $S_1$  to  $S'_1$ , is strictly no greater than the frequency reduction of the optimal seed by extending  $S_2$  to  $S'_2$ .  $\square$

Using Lemma 1, Lemma 2 and Corollary 2.1, we are ready to prove that the optimal divider cascading phenomenon is always true.

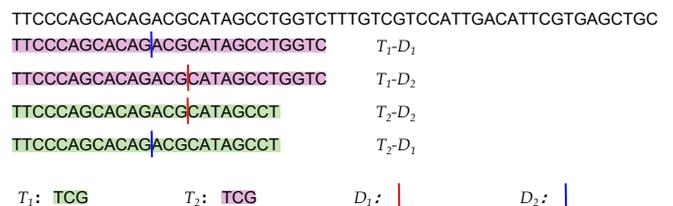
**THEOREM 1.** For two prefixes from the same iteration in OSS, as one prefix includes the other, the first optimal divider of the outer prefix must not be at the same or a prior position than the first optimal divider of the inner prefix.

We can prove Theorem 1 by contradiction. The proof is provided below:



**Fig. 14.** Among the four substrings,  $S_1$ ,  $S_2$ ,  $S'_1$  and  $S'_2$  from Figure 13, assume  $s'_1$  is the optimal seed of substring  $S'_1$ . Also assume two new seeds,  $s_1$  and  $s'_2$ . Between the two seeds,  $s_1$  starts at where  $s'_1$  starts but ends at where  $S_1$  ends while  $s'_2$  starts at where  $S'_2$  starts and ends at where  $s'_1$  ends.

PROOF. Assume  $T_1$  and  $T_2$  are two prefixes from the same iteration in “Optimal Seed Solver”, with  $T_1$  including  $T_2$ . Also assume  $T_1$ 's first optimal divider,  $D_1$ , is closer to the beginning of the read than  $T_2$ 's first optimal divider,  $D_2$ , as shown in Figure 15 ( $D_1 < D_2$ ).



**Fig. 15.** Two prefix  $T_1$  and  $T_2$  taken from the same iteration. We assume  $T_1$ 's first optimal divider is at  $D_1$  and  $T_2$ 's first optimal divider is at  $D_2$ .

Suppose we apply both divisions  $D_1$  and  $D_2$  to both prefixes  $T_1$  and  $T_2$ , which renders four divisions:  $T_1-D_1$ ,  $T_1-D_2$ ,  $T_2-D_1$  and  $T_2-D_2$ , as Figure 15 shows. We can prove that  $T_2-D_2$  is a **strictly less frequent** solution than  $T_2-D_1$ . Since  $D_2$  is the first optimal divider of  $T_2$  and  $D_1 < D_2$ , the minimum frequency of dividing  $T_2$  at  $D_1$  must be greater than dividing  $T_2$  at  $D_2$ . Let  $\text{freq}(T, D)$  denotes the optimal frequency of dividing prefix  $T$  at position  $D$ , then based on our assumptions and Lemma 2, we have the following relationships:

$$\begin{cases} \text{freq}(T_1, D_1) \leq \text{freq}(T_1, D_2) \\ \text{freq}(T_2, D_2) < \text{freq}(T_2, D_1) \\ \text{freq}(T_1, D_1) \geq \text{freq}(T_2, D_1) \\ \text{freq}(T_1, D_2) \geq \text{freq}(T_2, D_2) \end{cases}$$

Based on Corollary 2.1, we know that the frequency reduction of extending  $T_2-D_1$  to  $T_1-D_1$  is strictly not greater than the frequency reduction of extending  $T_2-D_2$  to  $T_1-D_2$ . From Figure 15, we can observe that only the second parts of both  $T_2-D_1$  and  $T_2-D_2$  are extended into  $T_1-D_1$  and  $T_1-D_2$  respectively. Between  $T_2-D_1$  and  $T_2-D_2$ , we can see that  $D_1$  produces a longer second part than  $D_2$ . Based on the Corollary 2.1, the frequency reduction of extending  $T_2-D_2$  to  $T_1-D_2$  is no less than the frequency reduction of extending  $T_2-D_1$  to  $T_1-D_1$ . Given that  $\text{freq}(T_2, D_2) < \text{freq}(T_2, D_1)$  from above, we prove that  $\text{freq}(T_1, D_2) < \text{freq}(T_1, D_1)$ , which contradicts our assumption that  $\text{freq}(T_1, D_2) \geq \text{freq}(T_2, D_2)$ .

Therefore, the first optimal divider of  $T_1$  must not be at a prior position than the first optimal divider of  $T_2$ .  $\square$

### 1.3 Proof of optimal solution forwarding

From our experiment, we observe that many prefixes within the same iteration share the same optimal divider with the previous prefixes (please look at the example in the next section). Among them, most also share the same 2nd-part frequency. For such prefixes, we propose the theorem below:

**THEOREM 2.** *A prefix that shares the same 2nd-part frequency with the previous prefix while being divided by the previous prefix's first optimal divider must have the same first optimal divider as well as the same optimal total seed frequency.*

**PROOF.** We prove the above theorem by contradiction. Assume there exists another optimal divider,  $div_{new}$ , which is prior to the inherited optimal divider from the previous prefix,  $div_{prev}$ . Also assume  $div_{new}$  provides a solution that has either smaller or equal total seed frequency. Since the previous prefix is 1-bp longer than the current prefix,  $div_{new}$  could also be applied to the previous prefix, which generates a first part that is the same as the current prefix's first part, and a second part that is 1-bp longer than the current prefix's second part (both under  $div_{new}$ ). Given that a substring always provides less or equally frequent optimal seed(s) than any of its included shorter substrings (the proof of this fact is similar to Lemma 1), we know that the second part of the previous prefix provides less or equally frequent optimal seed than the second part of the current prefix. This suggests that, under  $div_{new}$ , the previous prefix generates a total seed frequency that is smaller than or equal to the optimal total seed frequency provided by  $div_{prev}$ . As the result,  $div_{prev}$  must not be the first optimal divider of the previous prefix, which leads to a contradiction.  $\square$

### 1.4 Example

This section presents an example of OSS in action. In this example, we are mapping a 100-bp read to the human reference genome under the error threshold of 3, as shown in Figure 16. Based on the pigeonhole principle, to tolerate 3 errors, we need a total of 4 seeds. According to the pseudo-code described in the Method section, there will be 3 iterations of finding partial optimal solutions in all prefixes of the read (1 seed, 2 seeds and 3 seeds respectively), followed by a final optimal divider search of 4 seeds in the entire read.

In the first iteration, OSS searches for optimal 1-seed solution of all prefixes. Since the frequency of a seed monotonically decreases with longer seed lengths, for a prefix, the least frequent seed in it would be itself. In the second iteration, OSS searches for optimal 2-seed solutions for all prefixes. In this iteration, OSS starts with the longest prefix and gradually progresses to shorter prefixes<sup>2</sup>.

Figure 16 shows how to derive the 2-seed optimal divider of a prefix. First, OSS inherits the optimal divider (marked in red) from

the previous prefix (in gray), based on *optimal divider cascading*. Then, OSS divides the current prefix using the same divider and checks if its second part (in pink) has the same frequency as the second part of the previous prefix's division. In this example, the second part of the previous prefix has an optimal frequency of 11 while the second part of the current prefix has an optimal frequency of 19. Based on *optimal solution forwarding*, the two second parts from the two prefixes are not equal, therefore we cannot forward the optimal solution from the previous prefix. Next, OSS starts moving the divider towards the beginning of the prefix and queries the optimal 1-seed frequencies of the two parts (numbers with green and pink backgrounds, respectively) as well as the frequency differences of the first part (green background with numbers highlighted in red) between two moves. When the frequency increase of the first part is greater than the optimal frequency of the second part, according to *early divider termination*, OSS stops moving the divider and selects the divider with the minimum total seed frequency and goes to the next prefix. In this example, the least frequent division is the first division, with the total seed frequency of 30. Hence,  $opt\_data[2][59]$  is filled with 30. For the next prefix, after inheriting the optimal divider from the current prefix, we observe that the optimal frequency of its second part (in brown) is equal to the optimal frequency of the second part of the current prefix. In this case, OSS forwards the optimal divider of the current prefix as the optimal divider of the next prefix; it inherits the optimal frequency and moves on to the next prefix.

This process is repeated until all prefixes are processed in the second iteration. Figure 16 shows the  $opt\_data[2][\ ]$  array after the second iteration is finished. In this array, all prefixes that inherit the optimal solution from the previous prefix are marked with a blue background.

The third iteration is similar to the second iteration, except that the first part of the division now provides two seeds. This information is provided by  $opt\_data[2][\ ]$ . Figure 17 shows an example prefix in the third iteration.

Finally, OSS searches for the optimal divider of the entire read: the divider that divides the read into a 3-seed prefix and an 1-seed suffix that produces the least total seed frequency. This process is the same as searching for optimal prefix dividers in previous iterations, which starts from the rightmost position and gradually moves towards the beginning of the read. This process is also obliged by early divider termination.

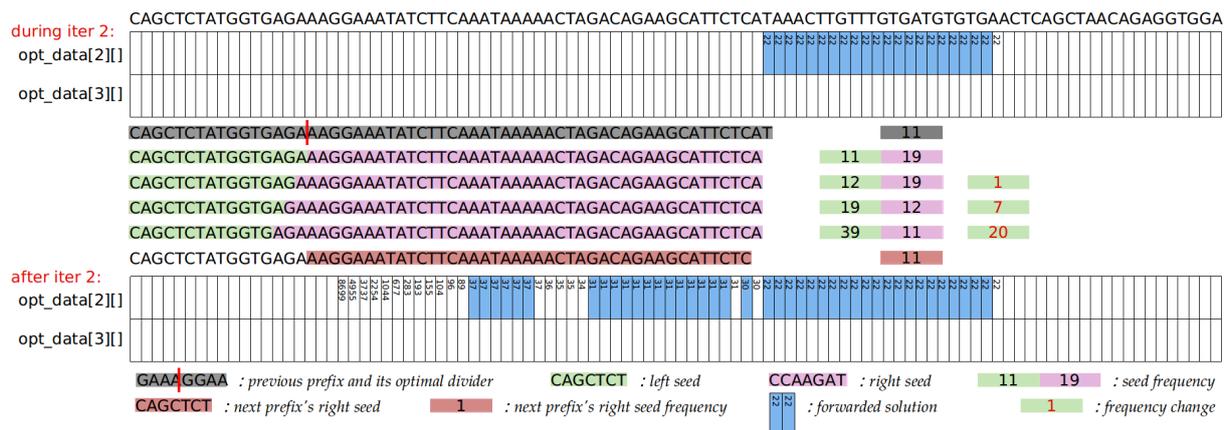
Figure 18 shows the process of the final optimal divider search. This figure also showcases *divider sprinting*. In Figure 18, we can see that from  $opt\_data[2][90]$  to  $opt\_data[2][86]$ , the optimal frequency is always 36; from  $opt\_data[2][85]$  to  $opt\_data[2][74]$ , the optimal frequency is always 42; from  $opt\_data[2][73]$  to  $opt\_data[2][66]$ , the optimal divider is always 43. This suggests that OSS only needs to evaluate the dividers at the beginning and end of each interval (90, 86, 85, 74, 73, 66 and 65) while skipping computations within the intervals. For the read provided in the example, the least total seed frequency is 47.

### 1.5 Backtracking in Optimal Seed Solver

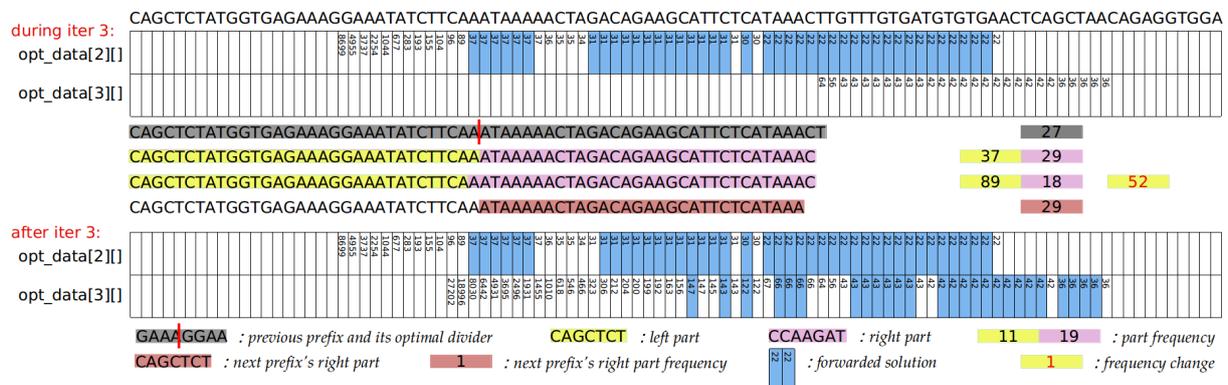
This section presents the pseudo-code of the backtracking process in OSS.

The pseudo-code of the backtracking process is provided in Algorithm 3. The key idea behind the backtracking algorithm is

<sup>2</sup> In our implementation, we do not start with the entire read but only start with the prefix that ends at  $L - (x - i) \times S_{min}$ , where  $x$  is the total number of required seeds and  $i$  is the current iteration. Any longer prefixes will not contribute to the final result. In this example, we have  $S_{min} = 10$ .



**Fig. 16.** This figure shows an example of applying OSS to a 100-bp read. In this figure, OSS is in the second iteration and is searching for the first optimal divider of a prefix. First, OSS tests if the optimal solution of this prefix could be forwarded from the previous prefix (which it cannot). Then, OSS gradually moves the divider towards the beginning of the prefix until the early divider termination is triggered. Finally OSS selects the least frequent division among all divisions as the optimal divider and stores the optimal frequency in  $opt\_data[2][59]$ .



**Fig. 17.** This figure shows the same example as Figure 16 but it illustrates a prefix in the third iteration. Similar to Figure 16, the prefix is also subjected to optimal solution forwarding, optimal divider cascading and early divider termination. The only difference would be that the first part of a division now provides 2 seeds, whose optimal frequency is obtained from  $opt\_data[2][ ]$ .

simple: the element of the  $i^{th}$  row and the  $j^{th}$  column of  $opt\_data$  stores the optimal divider,  $div$ , of the substring  $R[1...j]$ . This  $div$  suggests that by optimally selecting  $i - 1$  seeds from  $R[1...div - 1]$  and one seed from  $R[div...j]$ , we can obtain the least frequent  $i$  seeds from  $R[1...j]$ . From  $div$  we can learn that substring  $R[div...j]$  provides the  $i^{th}$  optimal seeds. Similarly, by repeating this process for the element of  $opt\_data[i - 1][div - 1]$ , we can learn the position and length of the  $(i - 1)^{th}$  optimal seeds. We can repeat this process until we have learnt all optimal dividers of the read.

## 1.6 Lock-step BWT

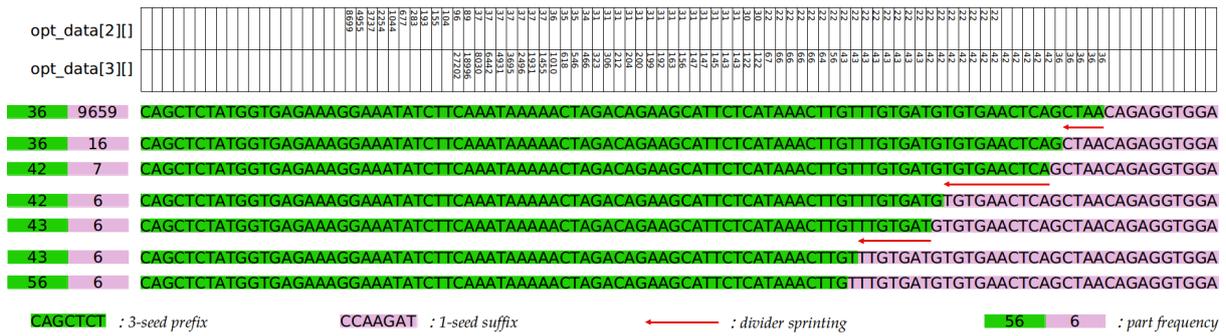
One major drawback of OSS is that it requires prior knowledge of frequencies of all possible seeds of the read. This can be a rather time consuming process since there can be a total of  $\mathcal{O}(L^2)$  seeds.

Näively, given large enough memory space, we can store seeds and their frequencies in a hash table. However, in this manner, the memory space grows exponentially with regard to the length of the seed. At 50 base-pairs, it requires at least  $4^{50}$  computer words to store all seeds in order to support  $\mathcal{O}(1)$  lookup.

### Algorithm 3: Backtracking

**Input:** the final optimal divider of the read,  $opt\_div$   
**Output:** an array of optimal dividers of the read,  $div\_array$   
**Global data structure:** the 2-D data array  $opt\_data[ ][ ]$   
**Pseudocode:**  
 // Push in the last divider  
 $div\_array.push(opt\_div);$   
 $prev\_div = opt\_div;$   
**for**  $iter = x - 1$  **to** 2 **do**  
 |  $div = opt\_data[iter][prev\_div - 1].div;$   
 |  $div\_array.push(opt\_div);$   
 |  $prev\_div = div;$   
**return**  $div\_array;$

Alternatively, we can use more memory efficient data structures and algorithms such as the Burrows-Wheeler Transform, or simply BWT. In BWT, all cyclic suffixes of the reference are sorted



**Fig. 18.** This figure shows the same example as the previous two figures. It shows the final search of the optimal 4-seed divider for the entire read (dividing the read into a 3-seed prefix and an 1-seed suffix). This search is also governed by early divider termination, as with previous iterations. In addition, this figure also shows how divider sprinting operates. Among dividers, when their first part frequencies remain the same, we can omit querying the frequency of the second part and skip to the end of the interval.

lexicographically, with the occurrence counts of letters in the last column of this suffix array stored in a separate `occ` matrix, as Figure 19 shows. The `occ` matrix counts how many times a letter has been seen in the last column of the sorted suffix array at each suffix row. Upon query, based on FM-indexing (Ferragina and Manzini, 2000), BWT traverses the query seed backward one base-pair at a time, starting from the last base-pair. In each traversal, BWT moves a head and a tail pointer in the `occ` matrix, based on the base-pair being traversed and the `occ` counters of the base-pairs in the seed. Specifically, it moves to the region where in the suffix array, all suffixes starts with this base-pair. Then it retrieves the two counts of the base-pairs in the `occ` matrix of the previous head and tail, as illustrated in Figure 19. Finally it applies both counts to the region as the new head and tail. Within the range between the head and the tail pointer, lie cyclic suffixes whose prefix share the same suffix with the seed from the current base-pair to the end.

A main problem with BWT is that it generates numerous memory accesses. For a seed of length  $|s|$ , it needs to move the head and tail pointer  $|s|$  times. Even worse, as each jump can be across very distant ranges in the sorted suffix array, it has a high probability of being a CPU cache miss. Cache misses are costly since an access to the main memory takes much longer time than an access to the CPU cache. Given that there are in total  $\mathcal{O}(L^2)$  seeds in a read, there could be as many as  $\mathcal{O}(L^3)$  memory accesses to obtain frequencies of all seeds for OSS.

In this section, we describe a mechanism, *lock-step BWT*, which reduces the per-read average case cache misses to  $\mathcal{O}(L)$ . Since this mechanism is beyond the scope of finding seeds optimally, in this paper we only provide a high level description of the mechanism.

Lock-step BWT is based on the observation that when one substring is a prefix of another substring, as Figure 20 shows, then for the last base-pair in both BWT traversals, which is the first letter of both substrings, the head and tail pointers of the shorter (inner) substring must include the head and tail pointers of the longer (outer) substring, as shown in Figure 20. This is because all cyclic suffixes in the suffix array between the head and tail pointers of a BWT traversal must share the same substring that is being traversed as their own prefix, as Figure 19 shows. For cyclic suffixes in the suffix array between pointers of the longer substring, they must also be between the pointers of the included shorter substring. However, this statement is not true vice versa.



**Fig. 19.** This figure shows the outcome of applying BWT to a 17-bp reference. In this figure, the reference is first concatenated with an ending mark, “\$”, and then rotated in a cyclic fashion, which generates a set of cyclic suffixes of the reference. Afterwards, we sort all suffixes lexicographically and use the last column (marked in blue) to generate the `occ` matrix. Upon query, we backtrack the query sequence base-pair by base-pair and moves a set of head and tail pointers according to the occurrence counts in the `occ` matrix. In this figure, we show the traversals of the first three base-pairs with each traversal marked in a unique color (green, yellow and red).

We further observe that for a set of long substrings that have the same inclusive relationship as above, like the one that is shown in Figure 21, the head and tail pointers of nearby prefixes (after sorted by length, prefixes that are close together) are usually nearby, especially for substrings that are not frequent. Even for frequent substrings, after certain lengths, we observe that their head and tail pointers can be grouped into a few nearby ranges. This suggests that one memory access to the `occ` matrix could potentially bring

Longer query: CTTATACCATACCAGGGAG  
Shorter query: CTTATACCATACCAGGGA

	A	C	G	T
...				
CTTATACCATACCAGGGAA..	8176	2313	4761	911
CTTATACCATACCAGGGAA..	8177	2313	4761	911
CTTATACCATACCAGGGAC..	8177	2313	4762	911
CTTATACCATACCAGGGAC..	8177	2313	4762	912
CTTATACCATACCAGGGAC..	8177	2313	4762	913
CTTATACCATACCAGGGAC..	8178	2313	4762	913
CTTATACCATACCAGGGAG..	8179	2313	4762	913
CTTATACCATACCAGGGAG..	8179	2314	4762	913
CTTATACCATACCAGGGAG..	8179	2315	4762	913
CTTATACCATACCAGGGAG..	8179	2315	4762	914
CTTATACCATACCAGGGAG..	8179	2315	4762	915
CTTATACCATACCAGGGAG..	8180	2315	4762	915
CTTATACCATACCAGGGAG..	8181	2315	4762	915
CTTATACCATACCAGGGAT..	8181	2316	4762	915
CTTATACCATACCAGGGAT..	8181	2316	4763	915
CTTATACCATACCAGGGAT..	8182	2316	4762	915
...				

**Fig. 20.** This figure shows the end result of traversing two strings in BWT, where one string is a prefix of the other string. We can see that the pointers of the longer string are included by the pointers of the shorter string. We can also observe that the head pointers and tail pointers of the two strings are nearby, suggesting that the `occ` data of both strings could collide in the same cacheline.

in data for nearby prefixes as well, since they could be stored in the same cacheline. We call this the spacial locality property of BWT traversals.

With above observations, we propose *lock-step BWT*, a mechanism that exploits the spacial locality property of BWT

traversals and reduces the number of cache misses by coordinating BWT traversals of all prefixes in the read, starting from the longest prefix (the read itself). When traversing prefixes, lock-step BWT only traverses one base-pair at a time and at each time traverses the same base-pair for all prefixes before moving on to the next base-pair, as Figure 21 shows. As a result, all prefixes share the same progress in lock-step. This helps grouping prefixes that have `occ` data in the same cacheline together and reduces the number of cache misses. After each single base-pair traversal, lock-step BWT records the size of the gap between the head and tail pointers of each prefix as the frequency of a seed.

To further reduce the number of cache misses, we impose a minimum seed length requirement of  $k$ -bp (e.g. 10-bp) and pre-process the BWT traversals of all  $k$ -bp string (all possible  $k$ -bp permutations). After pre-processing, we store the head and tail pointers in a hash table such that for a prefix query, we can directly skip over the first  $k$ -bp traversals (which are the last  $k$ -bp in the prefix) and start from the  $k + 1$ th base-pair traversal. By skipping the first few hops in a BWT-traversal, we have avoided the most cache-miss-prone regions in a prefix where the head and tail pointers changes drastically and are usually significantly distant from neighboring prefixes (therefore are less likely to be in the same cacheline).

Overall, with lock-step BWT, to obtain frequencies of all seeds, we need at most  $\frac{L^2}{2}$  cache misses in the worst case scenario where no two prefixes share pointers in the same cacheline. On average, we can reduce the number of cache misses to  $\mathcal{O}(L)$ . Given that cache hits are not time-consuming, obtaining frequencies of seeds will not be a major bottleneck of OSS.

Further exploration and quantitative comparisons of lock-step BWT against other related works will be studied in our future work.

prefixes:	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811229	811231
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811229	811232
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811228	811235
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811227	811235
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811225	811237
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811222	811239
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811217	811240
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811216	811241
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811216	811241
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811215	811242
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811213	811244
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811209	811250
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811204	811261
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811194	811273
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811187	811291
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811181	811303
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811171	811321
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811161	811350
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811149	811388
	GGACTCTATGGTGAGAAAGGAAATATCTTCAAATAAAAACTAGACAGAAGCATTCTCATAAACTTGT	811133	811446
	Current bp in BWT traversal: T	head	tail

**Fig. 21.** This figure illustrates how lock-step BWT operates. In lock-step BWT, all prefixes are processed together, one base-pair at a time. The red base-pairs (T) shows the current progress. Many head and tail pointers in this figure are close by to each other, suggesting that their data in the `occ` matrix could fold into the same cachelines.