Memory Efficient Minimum Substring Partitioning

Yang Li, Pegah Kamousi, Fangqiu Han, Shengqi Yang, Xifeng Yan, Subhash Suri
University of California, Santa Barbara
{yangli, pegah, fhan, sqyang, xyan, suri}@cs.ucsb.edu

ABSTRACT

Massively parallel DNA sequencing technologies are revolutionizing genomics research. Billions of short reads generated at low costs can be assembled for reconstructing the whole genomes. Unfortunately, the large memory footprint of the existing de novo assembly algorithms makes it challenging to get the assembly done for higher eukaryotes like mammals. In this work, we investigate the memory issue of constructing de Bruijn graph, a core task in leading assembly algorithms, which often consumes several hundreds of gigabytes memory for large genomes. We propose a disk-based partition method, called Minimum Substring Partitioning (MSP), to complete the task using less than 10 gigabytes memory, without runtime slowdown. MSP breaks the short reads into multiple small disjoint partitions so that each partition can be loaded into memory, processed individually and later merged with others to form a de Bruijn graph. By leveraging the overlaps among the k-mers (substring of length k), MSP achieves astonishing compression ratio: The total size of partitions is reduced from $\Theta(kn)$ to $\Theta(n)$, where n is the size of the short read database, and k is the length of a k-mer. Experimental results show that our method can build de Bruijn graphs using a commodity computer for any large-volume sequence dataset.

Source codes and datasets: `grafia.cs.ucsb.edu/msp`

1. INTRODUCTION

High-quality genome sequencing is foundational to many critical biological and medical problems. Recently, massively parallel DNA sequencing technologies [13], such as Illumina [2] and SOLiD [1], have been reducing the cost significantly. The price for Human Whole Genome Sequencing at a $30X$ coverage has dropped to $\$3,750$ (www.knome.com). The massive amount of short reads (short sequences with symbols A, C, G, T) generated by these next-generation techniques [13] quickly dominate the scene. How to manage and process the Big Sequence Data becomes an important issue.

A key problem in genome sequencing is finding the whole genome of various species, which is of great importance in many biological and medical problems. For example, it helps identifying the causes of many hard diseases. But unfortunately, the current low-cost techniques are unable to extract the whole genome directly. Existing sequencing techniques can only produce massive overlapping short reads that are randomly sampled from the genome. Therefore a major challenge in genome research is to assemble those short reads back into the whole genome - a Big Data problem. The number of short reads can easily reach one billion; and the length of each read varies from a few tens of bases to several hundreds. Figure 1 shows a sequence assembly process, where three short sequences are assembled to a longer sequence based on their overlaps.

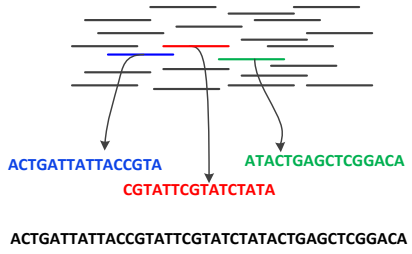


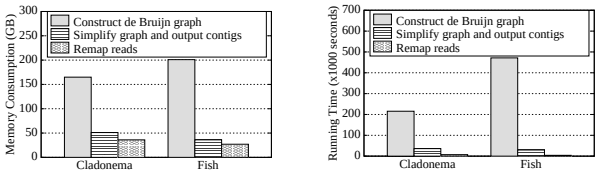
Figure 1: Sequence Assembly

The above process, called De novo assembly, has been extensively studied in the past decade. There are two kinds of approaches: the overlap-layout-consensus approach [17, 19], and the de Bruijn graph approach [18, 26, 23, 5, 12]. The overlap-layout-consensus approach builds an overlap graph between short reads. Due to the sheer size of the overlap graph (each read can overlap with many other reads), this approach is more suitable for small genomes. The de Bruijn graph approach breaks short reads to k-mers (substring of length k) and then connects k-mers according to their overlap relations in short reads. For assembling large quantities of short reads with great coverage produced by state-of-the-art sequencers, the de Bruijn graph approach becomes the most popular method.

Despite their popularity, large memory consumption is a bottleneck for both approaches [15]. For the short read sequences generated from mammalian-sized genome, algorithms such as Euler [18], Velvet [26], AllPaths [5] and SOAPdenovo [12] have to consume hundreds of gigabytes memory. Figure 2 shows a breakdown of memory and runtime consumption in SOAPdenovo [12] on a 258.7 GB Cladonema short read dataset and a 137.5 GB Lake Malawi cichlid

Proceedings of the VLDB Endowment, Vol. 6, No. 3

(fish) short read dataset. A de Bruijn graph based assembly process consists of six steps: error correction (optional), de Bruijn graph construction, contig generation, reads remapping, scaffolding (optional) and gap closure (optional). The last two steps are applicable when pair end information is available. Obviously, the most memory consuming and time intensive part is the de Bruijn graph construction step. Similar results were also reported for other datasets [12]. In this work, we resort to a novel disk-based approach to tackle this bottleneck, using less than 10 gigabytes memory, without runtime slowdown.



(a) *Peak Memory* (b) *Running Time*

Figure 2: SOAPdenovo: Statistics of Computational Complexity at Each Assembly Step

In a de Bruijn graph, each vertex represents a k-mer. In order to build the graph, we have to identify the same k-mers scattered in different short reads. A straightforward solution is to build a hash table. We can encode each symbol, A, C, G, and T using 2 bits. In the aforementioned 137.5 GB fish datasets (the read length is 101), when $k = 59$, there are about 11.8 billion distinct k-mers including reverse complements. Assuming a load factor of 2/3 for the hash table, we could expect the hash table to take nearly 283 GB memory, which is too large.

Alternatively, one can apply a disk-based partition-merge approach, which is popular in databases. Given a set of short reads S, there are two classic scatter-gather methods to identify duplicate k-mers: (1) partition S horizontally into disjoint subsets[1] $S_1, S_2, \ldots, S_t$, for each subset S_i, generate a hash table H_i of their k-mers in main memory, output a sorted copy H_i to disk, and then merge $H_1, H_2, \ldots, H_t$; (2) partition all k-mers from S into disjoint subsets $S_1, S_2, \ldots, S_t$ based on their last few symbols, for each subset S_i, create a hash table H_i, build a k-mer mapping and output H_i to disk, and then combine them. Both methods do not require a large amount of memory; but they are slow. The first solution, requiring multiple disk scans and sorts, is hopeless. The second one has to generate a huge number of k-mers in the first step. For the 258.7 GB Cladonema dataset, with k = 59, the disk file of k-mers is close to 3TB and the time used to finish duplicate mapping is around 30 hours.

In this paper, we re-examine the second scatter-gather approach and find a drawback existing in its k-mer partitioning strategy. Many k-mers generated from the same short read, though having large overlaps inside, are distributed to different partitions, which caused huge overhead. Inspired by this discovery, we introduce a new concept, called *minimum substring partitioning* (MSP). MSP breaks short reads to pieces larger than k-mers; each piece contains k-mers sharing a common minimum substring with fixed length p, $p \leq k$. The effect is equivalent to compressing consecutive k-mers using the original sequences. We demonstrate that this compression approach does not introduce significant computational overhead, but could lead to 10-15 times smaller partitions, thus improving performance dramatically. It is observed that the size of MSP partitions is only slightly larger than the original sequences. Based on a random string model, we analytically derive the expected size of minimum substring based partitions, which is reduced from $\Theta(kn)$ to $\Theta(n)$, where n is the size of the short read database, and k is the length of a k-mer. Furthermore, we prove that the size of the largest partitions decreases exponentially with respect to p, indicating that it is very memory-efficient. When $p = 12$, the memory consumption is less than 10 GB for all the real datasets we tested.

Our main contribution is the development of an innovative disk-based partitioning strategy for solving a critical graph construction problem in genome sequence assembly. Our solution is disk-based, using a small amount of memory without runtime performance loss. To the best of our knowledge, our study is the first work that introduces minimum substring partitioning, studies its properties, and successfully applies it to de novo sequence assembly, a critical problem in genome analysis. Experimental results show that our method can build de Bruijn graphs using a commodity computer for any large-volume sequence dataset.

2. PRELIMINARIES

Definition 1 (Short Read, K-Mer). *A short read is a string over alphabet Σ. A k-mer is a string whose length is k. Given a short read s, $s[i, j]$ denotes the substring of s between the i_{th} and j_{th} (both inclusive) elements. s can be broken into $m-k+1$ k-mers, written as $s[1, k]$, $s[2, k+1], \ldots$, $s[m-k+1, m]$. K-mers $s[i, k+i-1]$, $s[i+1, k+i]$ are called adjacent in s.*

For a short read s, we can view k-mers generated in a way that a window with width k slides through s. Two k-mers, α and β, are adjacent from α to β if and only if the last $k-1$ substring of α is the first $k-1$ substring of β. Let S be a short read set $S = \{s_i\}$. A k-mer extracted from s_i, $s_i[j, j+k-1]$, is written as $s_{i,j}$.

Definition 2 (De Bruijn Graph). *Given a short read set $S = \{s_i\}$, a de Bruijn graph $G = \{V, E\}$ is constructed by creating a vertex for every distinct k-mer in S and connecting two vertices with a directed edge if their corresponding k-mers are adjacent in at least one short read.*

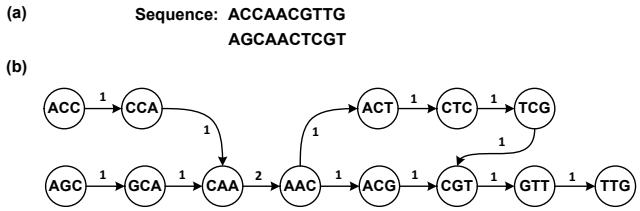


Figure 3: A de Bruijn Graph Example: k=3

Figure 3 shows a de Bruijn graph generated from two short reads with k being 3. The edge weight shows the number of times the two adjacent k-mers appear in short reads. For sake of simplicity, we do not depict the k-mers generated by the reverse complements of short reads (see details in Section 4).

[1]In this paper, we name each subset as a "partition".

2.1 K-mer Mapping

Given a short read dataset, in order to build a de Bruijn graph, one has to map all the duplicate k-mers derived from different short reads into the same vertex. If vertices are assigned with integer id's, e.g., starting at 1, this is equivalent to mapping duplicate k-mers to the same id. This process is called *K-mer Mapping*. Once the mapping is built, by scanning the short reads, we can create the edge set for the de Bruijn graph naturally. Therefore, the task of building a de Bruijn graph is narrowed down to k-mer mapping and edge sequence generation.

2.2 Scatter/Gather

One solution to the memory bottleneck issue is to chunk data to several partitions and process them separately [24, 25]. In this section, we discuss two scatter/gather approaches derived from duplicate detection techniques and then show their space complexity. The first solution is called Horizontal Partition (H-Partition).

1. Divide short read dataset S to disjoint partitions with equal size, S_1, S_2, ..., S_t, such that each partition can be loaded into memory.

2. For each partition S_i, insert k-mers into a hash table H_i. Based on the insertion order, assign an increasing integer id, starting at 1, to each distinct k-mer. Let M_i be the k-mer mapping function in S_i. M_i is local.

3. For each partition S_i, output all k-mers $s_{i,j}$ (in this case, we need to output the k-mer itself and its index, (i, j)) together with the assigned id, in increasing order of (i, j). Let P_i be the output sequences.

4. Merge $\{P_i\}$ to generate a global mapping function M such that it satisfies the following constraint. For any k-mer γ extracted from partition S_j, let S_i be the partition with the smallest i that contains γ, then $M(\gamma) = M_i(\gamma)$.

The output size of Step 3 is $\Theta(kn)$, where n is the size of the short read database, and k is the k-mer's length. Step 4 in H-Partition is costly. It needs a sort/merge process to identify the duplicate k-mers in different partitions.

The main issue of H-Partition arises from the fact that the multiple occurrences of the same k-mer are not located in the same partition. To overcome this issue, one common strategy is to do bucket partitioning. Let H be a hash function of k-mer. We can generate t partitions by distributing k-mer $s_{i,j}$ to the $H(s_{i,j}) \mod t$ partition. We can also use k-mers' last several symbols to scatter them into different partitions. This classic approach is called Bucket Partition (B-Partition).

1. Extract all k-mers from S and put them to disjoint partitions, $S_1, S_2, \ldots, S_t$, according to $H(s_{i,j}) \mod t$.

2. For each partition S_i, insert k-mers into a hash table H_i and assign an increasing integer id, starting at $\Sigma_{j=1}^{i-1}|S_j|$, to each distinct k-mer based on the insertion order, where $|S_j|$ is the number of distinct k-mers in partition S_j. Let M be this k-mer mapping function. It is clear that M is a global mapping function: each distinct k-mer in S will have one unique id.

3. For each partition S_i, output all k-mers $s_{i,j}$ (in this case, we only need to output the index (i, j), not the k-mer string) together with the assigned id, in increasing order of (i, j). Let P_i be the output sequences.

4. Merge $\{P_i\}$ in increasing order of (i, j).

While Step 4 in B-Partition is much faster than that in H-Partition, the total size of all the partitions is the same $\Theta(kn)$, which could easily reach multiple terabytes for a large genome. In the following discussion, we introduce a new partitioning concept, minimum substring partitioning (MSP), that reduces the partition size to $\Theta(n)$.

3. MINIMUM SUBSTRING PARTITIONING

Bucket partitioning has high overhead since adjacent k-mers are likely distributed to different partitions, unless $H(s_{i,j}) \mod t = H(s_{i,j+1}) \mod t$. Karp and Rabin [10] proposed a rolling hash function with the property that the hash value of consecutive k-mers can be calculated quickly. However, it is unknown whether there exists such a hash function that with high probability, two adjacent k-mers could be mapped to the same partition. In this study, we resort to another approach to bypass this problem.

DEFINITION 3 (MINIMUM SUBSTRING[20]). *Given a string s, a length-p substring r of s is called the minimum p-substring (or pivot substring) of s, if $\forall s'$, s' is a length-p substring of s, s.t., $r \leq s'$ ($\leq$ defined by lexicographical order). s is said to be covered by r. The minimum p-substring of s is written as $min_p(s)$.*

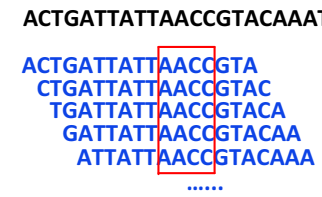


Figure 4: Minimum Substring Partitioning

Since two adjacent k-mers overlap with length $k-1$ substring, the chance for them to have the same minimum p-substring ($p < k$) could be very high. Figure 4 illustrates that the first 5 k-mers have the same minimum 4-substring, $AACC$. In this case, instead of generating these 5 k-mers separately, one can just compress them using the original short read, to $ACTGATTATTAACCGTACAAA$, and output it to the partition corresponding to the minimum 4-substring $AACC$. Formally speaking, given a short read $s = s_1 s_2 \ldots s_m$, if the adjacent j k-mers from $s[i, i+k-1]$ to $s[i+j-1, i+j+k-2]$ share the same minimum p-substring r, then one can just output substring $s_i s_{i+1} \ldots s_{i+j+k-2}$ to partition $H(r) \mod t$ without breaking it to j k-mers. If j is large, this compression strategy will dramatically reduce the partition size and runtime.

DEFINITION 4 (MINIMUM SUBSTRING PARTITIONING). *Given a string $s = s_1 s_2 \ldots s_m$, $p \leq k \leq m$, minimum substring partitioning breaks s to substrings with maximum length $\{s[i, j] | i + k - 1 \leq j, 1 \leq i, j \leq m\}$, s.t., all k-mers in $s[i, j]$ share the same minimum p-substring. $s[i, j]$ is also called super k-mer.*

According to minimum substring partitioning, larger p will likely break a sequence to several segments with different minimum p-substrings, thus increasing the total partition size. On the other hand, a smaller p will produce larger partitions that might not fit in the main memory.

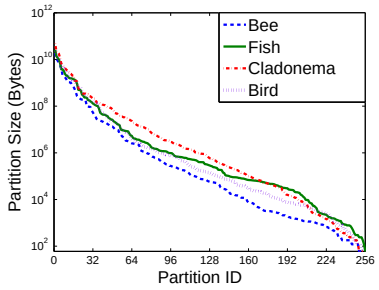


Figure 5: Partition Size Distribution

Figure 5 shows the distribution of partition size with $p = 4$ on the bee, fish, cladonema and bird datasets (ref. to Table 1 for details). The partitions are sorted according to their sizes. There are several large dominating partitions. The value of p determines the total size of partitions and the expected size of the largest partitions. Two techniques are developed to counter the skew distribution. First, when p is not too small (e.g. 10), the size of largest partitions becomes similar. Second, the Wrapped Partition technique introduced later in Section 5.1 can reduce skewness too.

In the following discussion, we are going to employ a random string model and examine some properties of MSP. We do not mean that genomes are structured randomly. We prove that the expected total partition size is $\Theta(n)$, far smaller than $\Theta(kn)$ in H-Partition and B-Partition. We will further show the lower and upper bound of the largest partition in MSP, which decreases exponentially with respect to p, indicating that MSP is very memory-efficient.

3.1 Total Partition Size

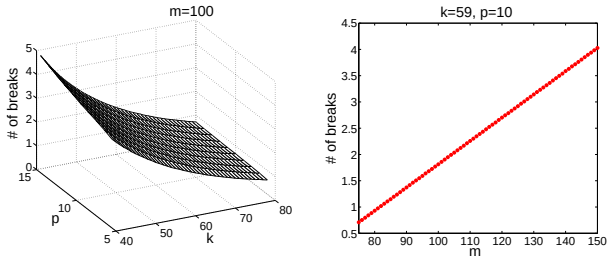
Let l be the average number of breaks that MSP introduces in a given sequence dataset. That is, on average, MSP adds l breaks to a sequence and divides it into multiple substrings $s[i_1, j_1]$, $s[i_2, j_2]$, ..., $s[i_{l+1}, j_{l+1}]$. Let m be the length of individual short reads. Suppose there are n/m short reads, i.e., n is the dataset size. We have the following theorem.

THEOREM 3.1. *The total partition size is* $\Theta(\frac{lk}{m}n + n)$.

PROOF. Each break introduces a substring that overlaps its previous substring with $k-1$ symbols. We have $\frac{n}{m}l$ breaks. Hence, the total partition size is $\Theta(\frac{lk}{m}n + n)$. □

Assume a random string model with four symbols 0, 1, 2, and 3, each having equal probability to occur. We first use a simulation method to demonstrate the average number of breaks for $1M$ short reads with length $m = 100$.

Figure 6(a) shows the expected number of breaks with respect to different p and k values. When p increases, the number of breaks increases. When k increases, the number of breaks decreases. Figure 6(b) shows the expected breaks of short reads with respect to different m values, with $p = 10$ and $k = 59$. It is observed that the average number of breaks



(a) p *and* k (b) m

Figure 6: Average Number of Breaks

increases proportionally with respect to m. We prove this in the following theorem.

THEOREM 3.2. *Let* $l(m, k, p)$ *be the average number of breaks under minimum substring partitioning. In a random string model,* $l(m, k, p) \propto (m - k)$.

PROOF. It is trivial to have $l = 0$ when $m = k$, because the whole string has no break in this situation. Consider the difference between $l(m, k, p)$ and $l(m-1, k, p)$. In an m length string, let $P_1(k, p)$=Pr{the minimum *p-substring* of the last *k-mer* is different from the second last one}. This equals to $P_1(k, p)$=Pr{the first or the last p-substring is the only smallest p-substring}. Since P_1 is only related to the last $k+1$ characters, it is not related to m. Then we have,

$$\begin{aligned} l(m, k, p) &= l(m-1, k, p) + P_1(k, p) \\ &= \cdots = P_1(k, p) \cdot (m - k). \end{aligned}$$

□

Theorem 3.2 told us that l increases proportionally with respect to $m - k$, with a ratio of P_1. Now we examine the bound of $P_1(k, p)$.

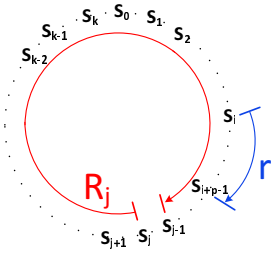


Figure 7: Illustration of Theorem 3.3

THEOREM 3.3. *In a random string model,* $P_1(k, p) \leq \frac{p+1}{k+1}$.

PROOF. Given any string $s = s_0 s_1 \ldots s_k$, we concatenate s_k and s_0 to form a ring as depicted in Figure 7. The ring can generate $k+1$ length-$(k+1)$ strings by starting at different positions: R_j=$s_j s_{j+1} \ldots s_{(j+k) mod (k+1)}$ $j = 0, 1, \ldots, k$. Let $S_{k,p}$ = { length-$(k+1)$ string whose first or last p-substring is the only minimum p-substring in it }. We have $P_1(k, p) = |S_{k,p}|/4^{k+1}$. Now we calculate at most how many R_i strings belong to $S_{k,p}$. Let r be one of the minimum p-substrings among all of the p-substrings in $\{R_i\}$. For any R_i, if r is located inside R_i (neither in the head nor the tail), then R_i does not belong to $S_{k,p}$. In total, there are $k - p$ R_i's

satisfying this condition. So in these $k+1$ R_i strings, at most $p+1$ of them can possibly belong to $S_{k,p}$. This gives us $P_1(k,p) \leq \frac{p+1}{k+1}$. □

COROLLARY 3.4. *In a random string model, the total partition size is $O(pn)$.*

PROOF. According to Theorems 3.1 and 3.3, $\frac{lk}{m}n + n < \frac{(m-k)n}{m}(p+1) + n < (p+1)n + n = O(pn)$. □

Since $p << k$, the total partition size $O(pn)$ is far smaller than $\Theta(kn)$ in traditional partition methods. In practice, p is fixed as a small constant; thus the size becomes $\Theta(n)$. In the following discussion, we present a stronger bound for the total partition size without this assumption.

THEOREM 3.5. *In a random string model, for any integer $a > 0$, $P_1(k+a, p+a) \leq 2 \cdot P_1(k,p) + \frac{p+2}{4^p}$.*

PROOF. Let $S_{k,p} = \{s||s| = k+1$, s' first or last p-substring is the only minimum p-substring in $s\}$, $S^*_{k,p} = \{s||s| = k+1$, s' first or last p-substring is one of the minimum p-substrings in $s\}$. We have $P_1(k,p) = |S_{k,p}|/4^{k+1}$. Let $P_2(k,p) = |S^*_{k,p}|/4^{k+1}$. Given a $k+a+1$ length string t that belongs to $S_{k+a,p+a}$, consider the $k+1$ length string consisting of the first $k+1$ characters of t. Obviously it belongs to $S^*_{k,p}$, so we have $P_1(k+a, p+a) \leq P_2(k,p)$. Hence, we only need to prove $P_2(k,p) - P_1(k,p) \leq P_1(k,p) + \frac{p+2}{4^p}$.

Given a string $s = s_1 s_2 \dots s_{k+1}$ which belongs to set $S^*_{k,p} - S_{k,p}$, we build an injective mapping from $S^*_{k,p} - S_{k,p}$ to $S_{k,p}$. For the situation where the first p-substring of s is one of the minimum p-substrings, let s_r be the first character that is not 0. Then we map $s = s_1 s_2 \dots s_{k+1}$ to $s' = s_1, \dots, s_{r-1}, s_r - 1, s_{r+1}, \dots, s_{k+1}$. It is easy to see that s' belongs to $S_{k,p}$, except two situations: p-substring $s_1 s_2 \dots s_p$ is (1) $00\dots0$ or (2) has only one 1 while all other characters are 0. These two situations have a probability of $\frac{p+1}{4^p}$ (detailed proof omitted due to space limit). Similarly, for the other situation where the last p-substring of s is one of the minimum p-substrings, we map $s = s_1 s_2 \dots s_{k+1}$ to $s'' = s_1, \dots, s_{r-1}, s_r - 1, s_{r+1}, \dots, s_{k+1}$, where s_r is the last character that is not 0. Then s'' belongs to $S_{k,p}$, except for the case that p-substring $s_{k-p+2} s_{k-p+3} \dots s_{k+1}$ is $00\dots0$, whose probability is $\frac{1}{4^p}$. Hence, $|S^* - S| \leq |S| + \frac{p+2}{4^p} \cdot 4^{k+1}$. That is, $P_2(k,p) \leq 2 \cdot P_1(k,p) + \frac{p+2}{4^p}$. □

Assuming $k = m/2$, $k < 100$, $p < k/5$, we have

$$kl = k \cdot l(m,k,p) = k \cdot P_1(k,p) \cdot (m-k) \quad (Theorem\ 3.2)$$
$$< (2k \cdot P_1(k-p+5, 5) + k \cdot \frac{7}{4^5}) \cdot (m-k) \quad (Theorem\ 3.5)$$
$$< (2 \cdot \frac{k}{k-p+6} \cdot 6 + 0.7) \cdot m/2 \quad (Theorem\ 3.3)$$
$$< (12 \cdot \frac{100}{86} + 0.7) \cdot m/2 < 7.4m.$$

Therefore, $\frac{kl}{m}n + n < 8.4n$, which is much better than $\Theta(kn)$.

3.2 Largest Partition Capacity

Since MSP has to load/hash each partition into main memory, the largest *partition capacity*, defined as the maximum number of distinct k-mers contained by a partition, determines the peak memory. We study its upper bound and lower bound in a random string model.

THEOREM 3.6. *In a random string model, the maximum percentage of distinct k-mers covered by one p-substring is bounded by $\frac{3k}{4^{p+1}}$, when $p \geq 2$.*

PROOF. In a random string model, each symbol has equal opportunity to appear in each position of short reads. The probability of observing any length-m string is equal. As the smallest p-substring defined by lexicographical order, the partition built on the p-substring $00\dots0$ has the largest number of distinct k-mers.

Let $\alpha(k,p)$ denote the percentage of distinct k-mers covered by p-substring $00\dots0$. This percentage is not related to m. For fixed p, there are two situations for a k-mer to have a p-substring $00\dots0$: (1) the first $k-1$ characters have a p-substring $00\dots0$, or (2) the last p-substring is the only $00\dots0$ in this k-mer. Note that in the later situation the first $k-p-1$ characters must not have a p-substring $00\dots0$ and the $(k-p)^{th}$ character must not be 0. This gives us

$$\alpha(k,p) = \alpha(k-1,p) + (1 - \alpha(k-p-1,p))\frac{3}{4} \cdot \frac{1}{4^p}$$

Obviously, $\alpha(p,p) = \frac{1}{4^p}$ and $\alpha(k,p) \geq \alpha(k-1,p)$. Thus

$$\begin{aligned}\alpha(k,p) &= \alpha(k-1,p) + (1 - \alpha(k-p-1,p))\frac{3}{4} \cdot \frac{1}{4^p} \\ &< \alpha(k-1,p) + \frac{3}{4^{p+1}} < \frac{1}{4^p} + (k-p) \cdot \frac{3}{4^{p+1}} \\ &< \frac{3k}{4^{p+1}}, \text{when } p \geq 2.\end{aligned}$$

□

We can further establish a lower bound,

$$\begin{aligned}\alpha(k,p) &= \alpha(k-1,p) + (1 - \alpha(k-p-1,p))\frac{3}{4} \cdot \frac{1}{4^p} \\ &> \alpha(k-1,p) + (1 - \alpha(k,p)) \cdot \frac{3}{4^{p+1}} \\ &> \frac{1}{4^p} + (k-p) \cdot (1 - \alpha(k,p)) \cdot \frac{3}{4^{p+1}}.\end{aligned}$$

From above, if $4 < p < k/5$, we have

$$\frac{2k}{4^{p+1}} < \alpha(k,p) < \frac{3k}{4^{p+1}}.$$

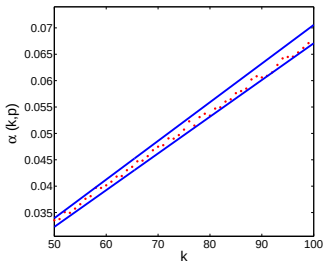


Figure 8: The Bounds of α(k,p)

Figure 8 depicts the bounds for the expected percentage of k-mers covered by the largest partition (corresponding to a p-substring $00\dots0$) with respect to different k values. p is set at 5. The result shows that the bounds we have proved are good: When k changes from 50 to 100, the maximum percentage of distinct k-mers covered by one minimum p-substring (the largest partition) is quite close to the lower and upper bounds we provided.

To calculate the entire distribution of partition capacities (the number of distinct k-mers covered by each p-substring) in a random string model, we develop an efficient quadratic-time algorithm ($O(m^2)$, see the Appendix). Using this algorithm, we do not need to use costly simulation to estimate the partition capacity.

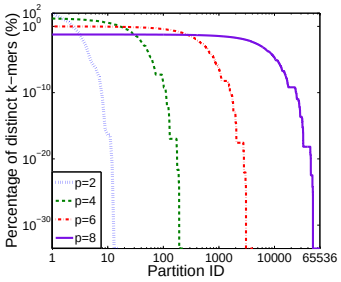


Figure 9: Expected Partition Capacity Distribution

Figure 9 shows the expected distribution of partition capacities with respect to different minimum substring lengths, assuming that 4 bases A, C, G, T appear with equal probability and k-mer length is 59. Here the p-substrings are sorted according to the percentage of k-mers they cover. The figure uses logarithm on both axes. The result shows a property: when p increases, there is a plateau where many p-substrings cover a similar percentage of distinct k-mers. We can conclude that there is no extremely memory consuming partition when p is not very small. Furthermore, the peak memory of MSP can be fully controlled by p.

4. REVERSE COMPLEMENTS

DNA sequences can be read in two directions: forwards and backwards with each symbol changed to its Watson-Crick complements ($A \leftrightarrow T$ and $C \leftrightarrow G$). They are called *reverse complement* and considered equivalent in bioinformatics. Most sequencing techniques extract short reads in either direction. In an assembly processing, each sequence should be read twice, once in the forward direction and then in the reverse complement direction.

Reverse complement is not an issue for bucket partitioning: when a k-mer is read into memory, a reverse complement can be built online. It becomes tricky for minimum substring partitioning since MSP intends to compress consecutive k-mers together if they share the same minimum p-substring. Unfortunately, their reverse complements might not share the same minimum p-substring. This forces us to generate the reverse complement explicitly for each short read, which will double the I/O cost.

DEFINITION 5. *[Minimum Substring with Reverse Complements] Given a string s, a length-p substring t of s is called the minimum p-substring of s, if $\forall s'$, s' is a length-p substring of s or s' reverse complement, s.t., $t \leq s'$ ($\leq$ defined by lexicographical order).*

Definition 5 redefines minimum substring by considering the reverse complement of each k-mer. With this new definition, we need not output reverse complements explicitly, nor change the minimum substring partitioning process. In the following discussion, if not mentioned explicitly, we will ignore this problem.

5. ALGORITHMS

In this section, we describe the detailed algorithm to build a de Bruijn graph. It consists of three steps: Partitioning, Mapping and Merging. Each step is performed by a program that takes an on-disk representation of input and produces a new on-disk representation of output. The input of the first step is the raw short read sequences and the output of the last step is a sequence of id's mapped to the k-mers in short read sequences, in the same order; the duplicate k-mers shall have the same id.

5.1 Partitioning

The first step is to partition short reads using MSP. A straightforward approach is as follows: (1) given a short read s, slide a window of width k through s to generate k-mers, (2) for each k-mer, calculate its minimum p-substring, (3) find super k-mers in s (adjacent k-mers sharing the same minimum p-substring). This method has to calculate the minimum p-substring of every k-mer. Each k-mer needs $(k-p+1)$ p-substring comparisons. Let m be the length of s. In total, this approach needs to perform $(k-p+1)*(m-k+1)=\Theta(mk)$ p-substring comparisons.

The above solution does not leverage the overlaps among adjacent k-mers. When the k-size window slides through s, we can maintain a priority queue on p-substrings in the window. Each time, when we slide the window one symbol to the right, we drop the first p-substring in the previous window from the queue and add the last p-substring of the current window into the queue. Since the number of p-substrings in a window is $k-p+1$ and there are $m-p+1$ p-substrings in s, the number of p-substring comparisons is $O((m-p+1)\log(k-p+1))=O(m\log k)$.

While the priority queue is theoretically good, the overhead introduced by the queue structure could be high. We thus introduce a simple scan algorithm, as described in Algorithm 1. Algorithm 1 first scans the window from the first symbol to find the minimum p-substring, say min_s, and the start position of min_s, say min_pos. Then it slides the window towards right, one symbol each time, till the end of the short read. After each sliding, it tests whether min_pos is still within the range of the window. If not, it re-scans the window to get the new min_s and min_pos. Otherwise, it tests whether the last p-substring of the current window is smaller than the current min_s. If yes, this last p-substring is set as the new min_s and its start position as the new min_pos. As analyzed in the previous section, adjacent k-mers likely have the same minimum p-substring. Therefore, it needs not to re-scan the window very often. Although the worst case time complexity is $O(mk)$ p-substring comparisons, Theorem 5.1 shows it could be more efficient in practice since the average number of breaks is small.

THEOREM 5.1. *Given an m-length string, assume minimum substring partitioning divides s into $l+1$ substrings. Algorithm 1 needs at most $\Theta(m+lk)$ p-substring comparisons.*

PROOF. Algorithm 1 shows that min_s and min_pos change under two conditions: (1) $i >$ min_pos, or (2) the last p-substring of $s[i, i+k-1] <$ min_s. Under the first condition, it re-scans the k-mer $s[i, i+k-1]$, which introduces $k-p+1$ p-substring comparisons. Under the second condition, it compares the last p-substring of $s[i, i+k-1]$ with the current min_s, which involves 1 p-substring comparison.

Algorithm 1 SimpleScan

```
Input: String s = s_1 s_2 ... s_m, integer k, p.
min_s = the minimum p-substring of s[1, k]
min_pos = the start position of min_s in s
for all i from 2 to m − k + 1 do
  if i > min_pos then
    min_s = the minimum p-substring of s[i, i + k − 1]
    min_pos = the start position of min_s in s
  else
    if the last p-substring of s[i, i+k−1] < min_s then
      min_s = the last p-substring of s[i, i + k − 1]
      min_pos = the start position of min_s in s
    end if
  end if
end for
```

Since the string s is broken into $l+1$ substrings, min_s and min_pos changes for l times. If all these l changes are due to the first condition, the total number of k-mer scans is $l+1$, including the initial scan of the first k-mer. This results in $(k-p+1)*(l+1)$ p-substring comparisons. Within each of these $l+1$ substrings, it needs $n_t - 1$ p-substring comparisons to test the second condition, where n_t is the number of k-mers within the substring $s[i_t, j_t]$. For all $l+1$ substrings, the total number of p-substring comparisons due to this test is $\Sigma_{t=1}^{l+1}(n_t - 1) = m - k - l$. Therefore the total number of p-substring comparisons of Algorithm 1 is bounded by $(k-p+1)*(l+1)+(m-k-l) = m+lk-pl-p+1$, which is $\Theta(m+lk)$. □

Definition 6 (Wrapped Partitions). *Given a string set $\{s_i\}$, a hash function H, the number of partitions t, for any k-mer $s_{i,j}$, minimum substring partition wrapping assigns $s_{i,j}$ to the $H(min_p(s_{i,j}))$ mod t partition.*

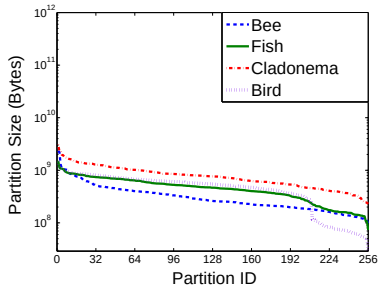


Figure 10: Partition Wrapping

Since each p-substring corresponds to one partition, the total number of partitions in MSP is equal to 4^p. When p increases, the number will increase exponentially. To counter this effect, one can introduce a hash function to wrap the number of partitions to any user-specified partition number. In this case, each partition generated from a p-substring is randomly included in a wrapped partition. The variance of partition sizes will likely decrease. Figure 10 shows the distribution of partition size when $p = 10$ and the number of wrapped partitions is set to 256. The number of partitions is the same as that of $p = 4$ without wrapping. In comparison with Figure 5, the partition size distribution is more uniform.

5.2 Mapping

In this step, each distinct k-mer is mapped to a unique integer id as its vertex id in the de Bruijn graph. For each partition, we create one hash table. Whenever there is a k-mer that does not exist in the table, a new id is assigned to it. The starting id of k-mers in one partition is the maximum k-mer id of the previous partition plus one. Therefore, all the partitions and their corresponding hash tables are disjoint. After one partition is processed, a disk file (called *id file*) is created, the entries, $\langle kmer, id\rangle$ in its hash table are written to that file.

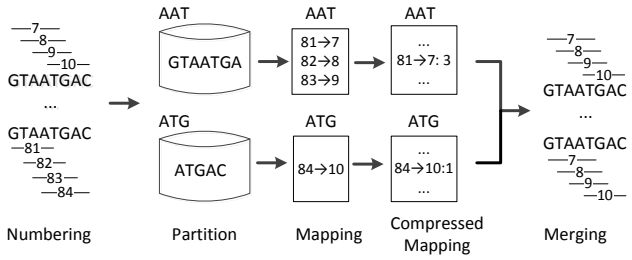


Figure 11: ID Replacement and Merging

In the merging step, we scan the short reads again to build edges for adjacent k-mers. For each pair of adjacent k-mers, we need to locate their ids from their corresponding id files. Considering that the id files are as big as the partition files, it will cause a lot of I/O and seriously slow down the process. In order to solve this problem, we develop an id replacement strategy, which is depicted in Figure 11. During the partitioning step, each k-mer is pre-assigned an integer id increasingly from 1. The same k-mer in different short reads receives different id's. For each partition, whenever we see a k-mer, we first look up the hash table to see if it exists: if yes, instead of writing a mapping record, $\langle kmer, id\rangle$, we write a replacement record, $\langle current\_id, first\_id\rangle$, into the id replacement file, indicating that this k-mer is a duplicate and we have to replace its pre-assigned id, $current\_id$, with the id associated with its first occurrence, $first\_id$.

Figure 11 shows an example of the id replacement process. Assume $k = 5$, $p = 3$, and $GTAATGAC$ occurs in two different short reads. In the beginning, each k-mer in two $GTAATGAC$ is assigned a unique id, e.g, $7 - 10$ and $81 - 84$, respectively. Sequences $GTAATGA$ is sent to the AAT partition, while $ATGAC$ is sent to the ATG partition. During this process, the k-mer 81 is mapped to the k-mer 7, 82 to 8, and 83 to 9, while the k-mer 84 is mapped to the k-mer 10. To compress the replacement file, we write the replacement records as a range instead of multiple individual records. For the example shown in Figure 11, one can just output a range record, $81 \rightarrow 7 : 3$, meaning the 3 consecutive id's starting at 81 will be replaced by 3 consecutive id's starting at 7. Range compression is quite effective since there are many long overlaps in short reads. According to our experiments, this kind of compression reduces the size of id replacement files to that of the original short read file.

5.3 Merging

After obtaining the id replacement files, the last step is merging. In this step, we merge all the replacement files to generate a sequence of id's that map to the original short reads, in the same order. We first open all the replacement

files with each file header pointing to the first id replacement record of the corresponding file. Since all the files are already naturally sorted in increasing order by the first entry (the pre-assigned id's to be replaced) of replacement record, we can find the minimum id to be replaced in the current filer headers. We enumerate id from 1, replace any id if there is a replacement record, move to the next replacement record, and iterate. After this process, we get a sequence of id's corresponding to k-mers in the short read dataset, in the same order. This actually forms a disk-based de Bruijn graph. The last step in Figure 11 shows this process, where two duplicated $GTAATGAC$ sequences receive the same id sequence. This disk-based graph can either be distributed across multiple machines, or scanned, compressed [26] and loaded into memory.

6. EXPERIMENTS

In this section, we present experimental results to illustrate the memory efficiency, effectiveness and important properties of the minimum substring partitioning method on four large real-life datasets: cladonema, bumblebee, fish, and bird. (1) We first analyze the efficiency of our graph construction algorithm in terms of memory and time cost, and compare it with two well-known open-source assembly programs, Velvet [26] and SOAPdenovo [12]. (2) The performance of MSP and two traditional partition/merge algorithms, H-Partition, B-Partition, are compared in terms of partition size and runtime. (3) We change different parameter settings to demonstrate important properties of MSP. All the experiments, if not specifically mentioned, are conducted on a server with 2.40GHz Intel Xeon CPU and 512 GB RAM.

6.1 Data Sets

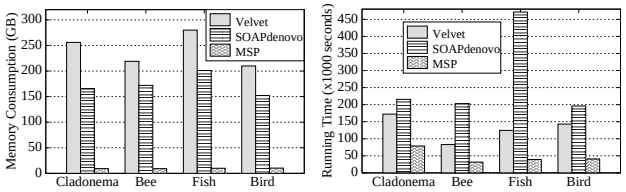
Four real-life short reads dataset are used to test our algorithms. The first one is the sequence data of Cladonema provided by our collaborators. The bee, fish and bird datasets are available via `http://gage.cbcb.umd.edu/data/Bombus_impatiens`, `http://bioshare.bioinformatics.ucdavis.edu/Data/hcbxz0i7kg/fish`, and `http://bioshare.bioinformatics.ucdavis.edu/Data/hcbxz0i7kg/Parrot/BGI_illumina_data`, respectively. Table 1 shows some basic facts.

	Cladonema	Bee	Fish	Bird
Size(GB)	258.7	93.8	137.5	106.8
Avg Read Length(bp)	101	124	101	150
# of Reads(million)	894	303	598	323

Table 1: Datasets: Cladonema, Bombus impatiens(bee), Lake Malawi cichlid(fish), and Budgerigar(bird)

6.2 Efficiency

We first conduct experiments to compare MSP with two real sequence assembly programs on de Bruijn graph construction: Velvet [26], a classic de Bruijn graph based assembler, and SOAPdenovo [12], a highly optimized and leading assembler. For all the experiments, we set the k-mer length to 59 [5]. For MSP, we partition the short reads into 1,000 wrapped partitions with the minimum substring length p being 12. SOAPdenovo is optimized to support multithreading, we use 2 threads here to illustrate its advantage. Both Velvet and MSP use 1 thread. The 8-thread version of SOAPdenovo can roughly achieve the same runtime as MSP. However, its peak memory consumption is still the same as its 2-thread version.



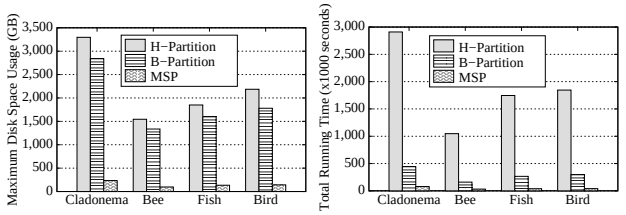
(a) *Peak Memory* (b) *Running Time*

Figure 12: Velvet, SOAPdenovo, and MSP

Figure 12 demonstrates that MSP outperforms Velvet and SOAPdenovo in terms of memory usage and running time. For large datasets, Velvet and SOAPdenovo easily consume more than 150G memory, while our method can complete the task with less than 10G memory, an order of magnitude reduction of memory usage. MSP is also much faster. We also observed that SOAPdenovo is slower than Velvet. This is due to two reasons. First, SOAPdenovo is designed to support multithreads. There are some costs to synchronize among threads. Second, SOAPdenovo has dynamic hash table allocation while Velvet pre-calculates the hash table size and allocates a hash table beforehand. When using many threads, SOAPdenovo will outperform Velvet in runtime.

6.3 Effectiveness

We then conduct experiments to compare MSP with other partition/merge algorithms, H-Partition and B-Partition. For all the three methods, we set the k-mer length to 59 and partition short reads into 1,000 partitions. For MSP, we set the minimum substring length p at 12. For B-Partition, we use the last 4 symbols to partition k-mers. All the three algorithms use the similar amount of memory (around 10 GB). Figures 13(a) and 13(b) show the maximum disk space usage and the total runtime.



(a) *Max Disk Space Usage* (b) *Running Time*

Figure 13: H-Partition, B-Partition, MSP

Figure 13 shows MSP outperforms the two baseline methods: compared with B-Partition, MSP can reduce the maximum disk space usage by 10-15 times and reduce the total execution time by 8-10 times. B-Partition was adopted by out-of-core algorithms such as [11]. It implies that MSP is better than the classic approach that does not leverage the overlaps among data records. H-Partition's overall performance is the worst since it needs multiple disk scans and sorts.

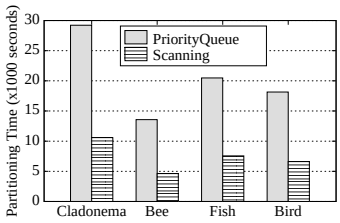
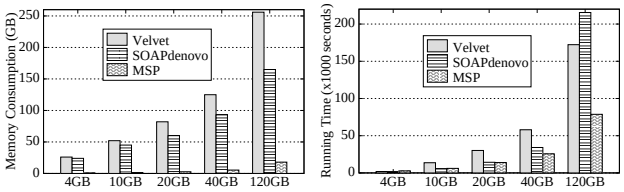


Figure 14: SimpleScan vs. Priority Queue

We then illustrate the advantage of using a scanning method (Algorithm 1) over a priority queue approach in the partitioning step. Here we set k at 59, p at 12 and partition the short reads into 1,000 wrapped partitions. Figure 14 shows that the simple scanning method in Algorithm 1 is around 2 times faster than the priority queue approach. Similar results were observed for other settings of p.

6.4 Scalability

We then conduct experiments to test the scalability of MSP. We vary the data size by randomly sampling the Cladonema dataset. For MSP, we partition the short reads into 1,000 wrapped partitions with p set at 10.



(a) *Peak Memory* (b) *Running Time*

Figure 15: Scalability

Figures 15(a) and 15(b) show that all the three algorithms scale linearly in terms of peak memory consumption and running time. MSP performs the best.

6.5 Properties of MSP

Next we conduct experiments to illustrate the properties of minimum substring partitioning. Figure 16 shows the change of peak memory, partition size, and running time with respect to varying length of minimum substring. Here, we set the k-mer length at 59 and partition short reads into 1,000 wrapped partitions. It shows that the peak memory will decrease significantly when the minimum substring length is increased. The total partition size and the running time will slightly increase. Both increases are negligible, showing that MSP is very effective in reducing memory consumption without affecting the runtime performance. It seems there is a wide range of values p can choose from. For various kinds of datasets we have, p works very well at $10 \sim 16$.

We then fix p at 10, the number of partitions at 1,000, and vary the length of k-mers. Figure 17 shows the change of peak memory, partition size, and running time with respect to k-mer length. It shows that the peak memory increases slowly together with k. It is also observed that increasing k will reduce the total partition size and the running time. There are two effects inside. Given n short reads with length m, the total size of all the k-mers is equal to $k(m-k+1)n$. We have

$$k(m-k+1) = \frac{(m+1)^2}{4} - (\frac{m+1}{2} - k)^2.$$

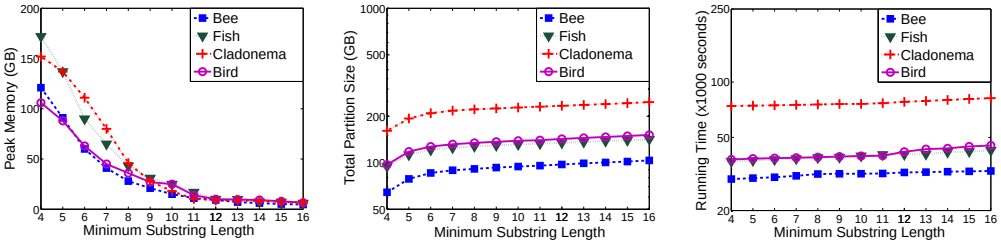
Hence, the size is peaked when $k = (m+1)/2$. The second effect is the compression ratio of MSP for larger k is higher. These two figures demonstrate the second effect dominates, since we do not observe a peak at $k = (m+1)/2$. The result is also in line with the analytical conclusion made for the random string model (see Theorems 3.2 and 3.6).

7. RELATED WORK

High throughput sequencing technologies are generating tremendous amounts of short reads data. Assembling these datasets becomes a critical research topic. With the development of next-generation sequencing techniques, the de Bruijn graph sequence assembly approaches became popular, including Euler[18], Velvet[26], AllPaths[5], SOAPdenovo[12], etc.

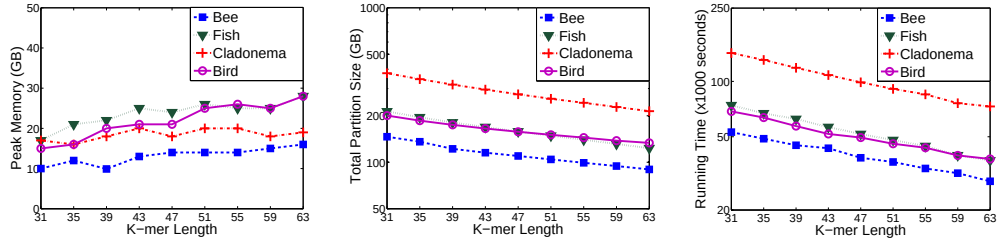
All these de Bruijn graph based algorithms have to solve a critical problem in the process of constructing de Bruijn graph, which merges duplicate k-mers into the same vertex. When the number of short reads comes to the level of billions, the de Bruijn graph can easily consume hundreds of gigabytes of memory. Several algorithms have been proposed to solve the memory overwhelming problem of graph-based assemblers. Simpson and Durbin [22] adopted FM-index [9] to achieve compression in building the string graph [16], which is an alternative graph formulation used in sequence assembly (string graph is much more expensive to construct than de Bruijn graph, so it is not as popular as de Bruijn graph). However, the step of building the suffix array and FM-index is very time-consuming and memory-intensive. Orthogonally, Conway and Bromage [6] used succinct bitmap data structure to compress the representation of de Bruijn graph. But the overall space requirement will still increase as the graph becomes "bigger" (more nodes and edges). Distributed assembly algorithms were also proposed, e.g., ABySS[23] and Contrail[21]. They partition k-mers in a distributed manner to avoid memory bottleneck. Unfortunately, using a hash function to distribute k-mers evenly across a cluster cannot ensure adjacent k-mers being mapped to the same machine. It results in intense cross-machine communications since adjacent k-mers form edges in the graph. The proposed minimum substring partitioning technique solves this problem: it not only generates small partitions, but also retains adjacent k-mers in the same partition.

The de Bruijn graph construction problem is related to duplicate detection. The traditional duplicate detection algorithms perform a merge sort to find duplicates, e.g., Bitton and DeWitt [4]. Teuhola and Wegner [25] proposed an $O(1)$ extra space, linear time algorithm to detect and delete duplicates from a dataset. Teuhola [24] introduced an external duplicate deletion algorithm that makes an extensive use of hashing. It was reported that hash-based approaches are much faster than sort/merge in most cases. Bucket sort [7] is adoptable to these techniques, which works by partitioning an array into a number of buckets. Each bucket is then sorted individually. By replacing sort with hashing, it can solve the duplicate detection problem too. Dupli-



(a) *Peak Memory* (b) *Total Partition Size* (c) *Running Time*

Figure 16: Varying Minimum Substring Length p



(a) *Peak Memory* (b) *Total Partition Size* (c) *Running Time*

Figure 17: Varying K-mer Length k

cate detection has also been examined in different contexts, e.g., stream [14] and text [3]. A survey for general duplicate record detection solutions was given by Elmagarmid, Ipeirotis and Verykios [8].

The problem setting of de Bruijn graph construction is different from duplicate detection in sense that elements in short reads are highly overlapped and a de Bruijn graph needs to find which element is a duplicate to which. The proposed minimum substring partitioning technique can utilize the overlaps to reduce the partition size dramatically. Meanwhile, the three steps, partitioning, mapping, and merging for disk-based de Bruijn graph construction can efficiently connect duplicate k-mers scattered in different short reads into the same vertex.

The concept of minimum substring was introduced in [20] for memory-efficient sequence comparison. Our work develops minimum substring based partitioning and its use in sequence assembly. We also theoretically analyze several important properties of minimum substring partitioning.

8. CONCLUSIONS

We introduced a new partitioning concept - minimum substring partitioning (MSP), which is appropriate and efficient to solve the duplicate k-mer merging problem in the assembly of massive short read sequences. It makes use of the inherent overlaps among k-mers to generate compact partitions. This partitioning technique was successfully applied to de Bruijn graph construction with very small memory footprint. We discussed the relations between the partition size and the minimum substring length and analytically derived the capacity of minimum substrings based on a random string model. Our MSP-based de Bruijn graph construction algorithm was evaluated on real DNA short read sequences. Experimental results showed that it can not only successfully complete the tasks on very large datasets within a small amount of memory, but also achieve better performance than existing state-of-the-art algorithms.

9. ACKNOWLEDGMENTS

We acknowledge support from NSF IIS-0954125, NSF CCF-1161495, and the Center for Scientific Computing at the CNSI and MRL: an NSF MRSEC (DMR-1121053) and NSF CNS-0960316. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

10. REFERENCES

[1] http://www.appliedbiosystems.com.
[2] http://www.illumina.com.
[3] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.
[4] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. *ACM Trans. Database Syst.*, 8:255–265, 1983.
[5] J. Butler, I. MacCallum, M. Kleber, I. Shlyakhter, M. Belmonte, E. Lander, C. Nusbaum, and D. Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.
[6] T. Conway and A. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (2nd ed.)*. MIT Press, 2001.

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19:1–16, 2007.

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

[10] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[11] V. Kundeti, S. R. S, H. Dinh, M. Vaughn, and V. Thapar. Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. *BMC Bioinformaticse*, 11:560, 2010.

[12] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[13] E. Mardis. Next-generation dna sequencing methods. *Annu. Rev. Genomics Hum. Genet.*, 9:387–402, 2008.

[14] A. Metwally, D. Agrawal, and A. E. Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

[15] J. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[16] E. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

[17] E. Myers, G. Sutton, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, S. Kravitz, C. Mobarry, K. Reinert, K. Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

[18] P. Pevzner, H. Tang, and M. Waterman. An eulerian path approach to DNA fragment assembly. In *Proceedings of the National Academy of Sciences*, pages 9748–9753, 2001.

[19] D. Platt and D. Evers. Forge: A parallel genome assembler combining sanger and next generation sequence data. 2010. http://combiol.org/forge/.

[20] M. Roberts, W. Hayes, B. Hunt, S. Mount, and J. Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[21] M. Schatz, D. Sommer, D. Kelley, and M. Pop. Contrail: Assembly of large genomes using cloud computing. 2010. http://contrail-bio.sf.net/.

[22] J. Simpson and R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[23] J. Simpson, K. Wong, S. Jackman, J. Schein, S. Jones, and İ. Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[24] J. Teuhola. External duplicate deletion with large main memories. 1993.

[25] J. Teuhola and L. Wegner. Minimal space, average linear time duplicate deletion. *Communications of the ACM*, 34(3):62–73, 1991.

[26] D. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

11. APPENDIX

We design an efficient polynomial-time algorithm for computing the probability that a given p-substring is the minimum p-substring of a random n-length string S. The complication arises because of the huge overlaps among the p-substrings of S: each p-substring shares $p-1$ symbols with its predecessor, so these subproblems are not *independent*. We design a non-trivial *dynamic programming* algorithm that circumvents this complication, and leads to an $O(n^2)$ algorithm. Because the underlying problem is quite general, we find it best to describe the problem and its solution using the following abstract setting.

Let $S = s_1 s_2 \dots s_n$ be a random string (the DNA sequence), where each letter s_i is an independent random variable taking values from the set $\Sigma = \{0, 1, 2, 3\}$ with probabilities p_0, p_1, p_2, p_3, respectively. That is, s_i assumes value j with probability p_j, for $j = 0, 1, 2, 3$, and these probabilities sum to 1, namely, $\sum_{j=1}^{4} p_j = 1$. We will use the notation S_i for the prefix substring of S of length i, namely, $s_1 s_2 \dots s_i$, and $S(j)$ for its suffix substring of length j, namely, $s_{n-j+1} \dots s_n$. The notation $S_i(j)$ will be used for the j symbol long suffix of the prefix substring S_i ($j \leq i$), namely, $s_{i-j+1} \dots s_i$ (see Figure 18). We will adopt the convention that substrings of length zero are empty; in particular, $S_i(0)$ and $S_0(j)$ are empty strings. Any two substrings of equal length can be compared using the lexicographical order, and we will use the standard notation $<, \leq, =, \geq, >$ to denote their relative order.

In order to distinguish the target string W from the DNA sequence, we will call the former a *word*. In particular, given an m-word W (to distinguish the abstract problem from the real problem, here we use m instead of p), also on the alphabet $\Sigma = \{0, 1, 2, 3\}$, we wish to compute the probability that no m-substring of S is smaller than or equal to W. More specifically, what is the probability that $S_i(m) > W$, for all $i = m, m+1, \dots, n$. As we will argue later, if we know this probability for W and the m-word immediately preceding W in the lexicographical ordering, then by calculating their difference we can get the probability that W *itself* is the minimum m-substring, which is what we ultimately need.

In order to build some intuition into the problem, let us consider the prefix S_i. Let us call S_i *clean* if it does not contain an m-substring $\leq W$. Suppose we inductively assume S_{i-1} to be clean. Then, it follows that S_i is clean only if $S_i(m) > W$. In other words, to ensure that a prefix substring S_i is clean we need two conditions: (1) the substring S_{i-1} is clean, (2) the *m-suffix* of S_i, $S_i(m)$, is larger than W. In fact, we will need these conditions to be *recursively* enforced, meaning that we will need $S_i(j) > W_j$, for all j.

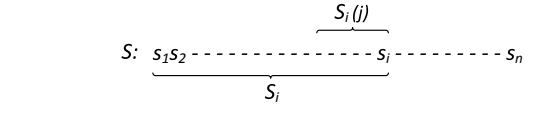


Figure 18: Illustration of S, S_i, and the j-suffix of S_i.

With this motivation, we now define the 2-dimensional table Q, which will form the basis of our dynamic programming algorithm. The table Q has size $(n+1) \times m$, where

the entry $Q[i,j]$ holds the probability that S_i is clean *and* $S_i(j) > W_j$. Thus, $Q[i,0]$ is the probability that S_i is clean, and the final value we wish to compute is $Q[n,0]$, which is the probability that the entire string S is clean, meaning it has no m-substring less than or equal to W. Of course, the probability that W *is the minimum* m-word in S is easily computed as $Q'[n,0] - Q[n,0]$, where Q' is the same dynamic programming table computed for the target m-word W', where W' is the immediate predecessor of W in the lexicographical ordering of m-words.

Algorithm *MinSTB* (Minimum Substring Tail Bounds) describes in pseudo-code how to compute the Q table in row-major order, with the convention that $Q[0,j] = 1$ for all j. Assuming the first i rows of the table have been computed, the algorithm shows how to compute the row $i+1$. The analysis of the algorithm is given in the following theorem.

Algorithm *MinSTB*: Computes the values $Q[i+1,j]$.

Require: $0 \le j \le m,\ j \le i \le n$
  if $i+1 < m$ **then**
    $Q[i+1,0] = 1.$
    $Q[i+1,1] = \sum_{k>w_1}^{3} p_k.$
    $Q[i+1,j] = \sum_{k>w_j}^{3} p_k + Q[i,j-1]\cdot p_{w_j},\quad j>1.$
  else
    $Q[i+1,0] = Q[i,0]\cdot\sum_{k>w_m}^{3} p_k + Q[i,m-1]\cdot p_{w_m}.$
    if $w_m > w_j$ and $j>0$ **then**
      $Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_m}^{3} p_k + Q[i,m-1]\cdot p_{w_m}.$
    end if
    if $w_m < w_j$ and $j>0$ **then**
      $Q[i+1,1] = Q[i,0]\cdot\sum_{k>w_1}^{3} p_k.$
      $Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j}^{3} p_k + Q[i,j-1]\cdot p_{w_j}\ .$
    end if
    if $w_m = w_j$ and $j>0$ **then**
      $Q[i+1,1] = Q[i,0]\cdot\sum_{k>w_1}^{3} p_k.$
      if $W_{m-1}(j-1) > W_{j-1}$ **then**
        $Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j}^{3} p_k + Q[i,m-1]\cdot p_{w_j}.$
      end if
      if $W_{m-1}(j-1) \le W_{j-1}$ **then**
        $Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j}^{3} p_k + Q[i,j-1]\cdot p_{w_j}.$
      end if
    end if
  end if

THEOREM 11.1. *Given a random string S and an m-word W on $\Sigma = \{0,1,2,3\}$, we can compute the probability that S has no m-substring $\le W$ in $O(n^2)$ time.*

PROOF. We prove how Algorithm *MinSTB* correctly computes the $(i+1)$th row of the table Q from the ith row. First consider the case where $i+1 < m$. Then S_{i+1} has no m-substring, and we just need that $S_{i+1}(j) > W_j$. If $j=0$, then $Q[i+1,j] = 1$. If $j=1$, then we simply need that $s_{i+1} > w_1$. Thus

$$Q[i+1,1] = \sum_{k>w_1}^{3} p_k.$$

Finally if $j>1$, then all we need is that either (1) $s_{i+1} > w_j$, or (2) $s_{i+1} = w_j$ and $S_i(j-1) > W_{j-1}$. Therefore:

$$Q[i+1,j] = \sum_{k>w_j}^{3} p_k + Q[i,j-1]\cdot p_{w_j},$$

where $\sum_{k>w_j}^{3} p_k$ is the probability that $s_{i+1} > w_j$, p_{w_j} is the probability that $s_{i+1} = w_j$, and $Q[i,j-1]$ is the probability that S_i is clean and $S_i(j-1) > W_{j-1}$.

Now consider the case where $i+1 > m$. Suppose S_i is clean, then S_{i+1} is not clean if and only if $S_{i+1}(m) \le W$. This will **not** happen if and only if $s_{i+1} > w_m$, or $s_{i+1} = w_m$ but $S_i(m-1) > W_{m-1}$. Therefore

$$Q[i+1,0] = Q[i,0]\cdot\sum_{k>w_m}^{3} p_k + Q[i,m-1]\cdot p_{w_m},$$

where $Q[i,0]$ is the probability that S_i is clean.

The computation of $Q[i+1,j]$ for $j \neq 0$ depends on the value of w_m compared to w_j. This stems from the fact that if $w_m < w_j$, then we only need to compare S_{i+1} against W_j, i.e., if $S_i(j-1) > W_{j-1}$ then necessarily $S_i(m-1) > W_{m-1}$. In fact, there are three cases:

1. $w_m > w_j$. In this case if $s_{i+1} > w_m > w_j$ then S cannot have any m-substring $\le W_m$, or any j-substring $\le W_j$, which ends at index i. So all we need is for S_i to be clean. If $s_{i+1} < w_m$ then S_{i+1} is not clean. If $s_{i+1} = w_m$, then $s_{i+1} > w_j$ and $S_{i+1}(j) > W_j$. In this case we just need that $S_i(m-1) > W_{m-1}$, and we have

   $$Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_m} p_k + Q[i,m-1]\cdot p_{w_m}.$$

   This holds if $j=1$, since $s_{i+1} > w_j$ even if $s_{i+1} = w_m$.

2. $w_m < w_j$. The argument is similar to the previous case: if $s_{i+1} > w_j > w_m$, then all we need is for S_i to be clean. If $s_{i+1} = w_j$ then if $j=1$,

   $$Q[i+1,1] = Q[i,0]\cdot\sum_{k>w_1} p_k,$$

   but if $j>1$ we need that $S_i(j-1) > W_{j-1}$. Therefore

   $$Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j} p_k + Q[i,j-1]\cdot p_{w_j}.$$

3. $w_m = w_j$. If $s_{i+1} > w_m$ then all we need is for S_i to be clean. If $s_{i+1} = w_m = w_j$, then if $j=1$

   $$Q[i+1,1] = Q[i,0]\cdot\sum_{k>w_j} p_k.$$

   If $j>1$ there are two cases:

   - $W_{m-1}(j-1) > W_{j-1}$. Then if S_{i+1} is clean, necessarily $S_{i+1}(j) > W_j$ because $s_{i+1} = w_j$ and $S_{i+1}(m-1) > W_{m-1}$, which implies that $S_{i+1}(j-1) > W_{j-1}$. Therefore we only need S_{i+1} to be clean, which happens only if $S_i(m-1) > W_{m-1}$. In this case:

     $$Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j} p_k + Q[i,m-1]\cdot p_{w_j}.$$

   - $W_{m-1}(j-1) \le W_{j-1}$. The argument is the same as the previous case, except that now we need $S_i(j-1) > W_{j-1}$, which happens with probability $Q[i,j-1]$. Thus

     $$Q[i+1,j] = Q[i,0]\cdot\sum_{k>w_j} p_k + Q[i,j-1]\cdot p_{w_j}.$$

Each entry of the table can be computed in constant time, and therefore the whole table can be computed in $O(n^2)$. □