Memory Efficient Minimum Substring Partitioning

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

ABSTRACT

Massively parallel DNA sequencing technologies are revolutionizing genomics research, with billions of short reads generated at low costs. Multiple de novo assembly algorithms have been developed to assemble these reads for reconstructing the whole genomes. Unfortunately, the large memory footprint of the existing algorithms makes it challenging to get the assembly done for higher eukaryotes like mammals.

In this paper, we investigate the problem of constructing de Bruijn graph, a core task in several leading assembly algorithms such as Velvet and SOAPdenovo. For large genomes, it often takes several hundreds of gigabytes memory to build a de Bruijn graph. Therefore we propose Minimum Substring Partitioning (MSP), a disk-based approach, 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.

1. INTRODUCTION

High-quality genome sequencing is foundational to many critical biological and medical problems. Recently, massively parallel DNA sequencing technologies [17], including Illumina [2] and SOLiD [1], have been improving significantly, and the cost is continuously lowering down. The price for Human Whole Genome Sequencing at a $30X$ coverage has dropped to $\$4,998$ (www.knome.com). The massive amount of short reads (short sequences with symbols A, C, G, T) generated by these next-generation techniques [17] quickly dominate the scene. How to manage and process the ever-growing sequence data will soon become a database issue.

A key problem in genome sequencing is assembling massive short reads, which are randomly extracted from samples of DNA segments. The number of short reads varies from hundreds of thousands to billions; and the length of each read varies from a few tens of bases to several hundreds. Figure 1 shows the sequence assembly process, where three short sequences are assembled to a longer sequence based on their overlaps.

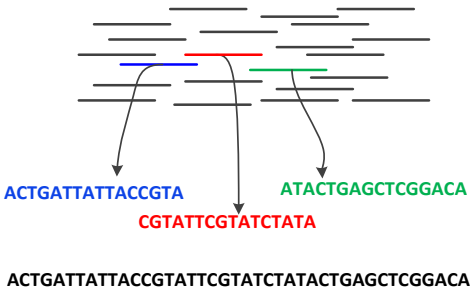


Figure 1: Sequence Assembly

The above process, called De novo assembly, has been extensively studied in the past decade. The existing algorithms can be classified into two categories based on their internal graph structure: the overlap-layout-consensus approach [22, 24], and the de Bruijn graph approach [23, 30, 27, 6, 16]. 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), it is more useful for small genomes. The de Bruijn graph approach breaks short reads to k-mers (substring of length k) and then connect k-mers according to their overlap relations in short reads. It is able to assemble larger quantities (e.g., billions) of short reads with greater coverage.

Despite their popularity, large memory consumption is a major problem for both approaches [19]. For the short read sequences generated from mammalian-sized genome, algorithms such as Euler [23], Velvet [30], AllPaths [6] and SOAPdenovo [16] will not be able to finish assembling successfully within a reasonable amount of memory. 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 only available for short reads datasets with pair end information. Figure 2

Proceedings of the VLDB Endowment, Vol. X, No. Y

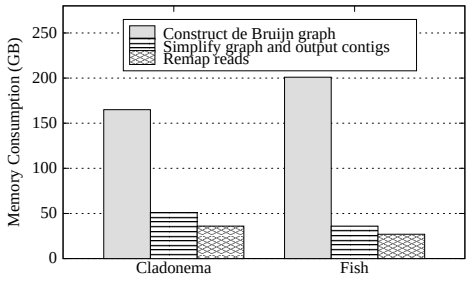


Figure 2: SOAPdenovo: Peak Memory in Each Step

shows a breaddown of memory consumption of these steps, generated by SOAPdenovo [16] on a 126.3 GB Cladonema short read sequences dataset (the read length is 101 and $k = 59$) and a 76.3 GB Lake Malawi cichlid (fish) short read sequences dataset (the read length is 101 and $k = 59$). Here the last two steps (scaffolding and gap closure) are not available to conduct due to the datasets properties. Clearly the most memory consuming part is the de Bruijn graph construction. The similar result was also reported for other datasets. For example, in an assembly experiment conducted on the Human African sequencing data set [16], the de Bruijn graph construction step used 140 GB memory, which is the most memory consuming part. Various techniques have been developed to reducing memory use in each step, e.g., Quake [14] for error correction, Forge [24] and ABySS [27] for distributed processing. In this work, we focus on the de Bruijn graph construction and demonstrate that our disk-based approach can complete the task using less than 10 gigabytes memory, without runtime slowdown.

In de Bruijn graph construction, we need to identify duplicate k-mers that belong to the same vertex. This task is different from removal of duplicates. An intuitive approach is to build a hash table. We can encode each symbol, A, C, G, and T using 2 bits. In a 76.3 GB Lake Malawi cichlid (fish) short read sequences dataset (the read length is 101), if $k = 59$, 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.

In order to reduce the memory footprint, one can adapt a standard disk-based partition-merge approach. 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, $S_1, S_2, \ldots, S_l$, for each subset S_i, generate a hash table H_i of their k-mers in main memory, and output a sorted copy H_i to disk, and finally merge $H_1, H_2, \ldots, H_l$; (2) partition all k-mers from S into disjoint subsets $S_1, S_2, \ldots, S_l$ 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 finally merge them. Both methods do not require very large amount of memory. However, they involve very high I/O overhead. The first solution, requiring multiple disk scans and sorts, is hopeless. The second one will generate a huge number of kmers in the first step. For the 126.3 GB Cladonema short read sequences dataset, with k = 59, the k-mer disk file is nearly 2,850GB and the time used to finish duplicate mapping is around 31 hours.

In this paper, we re-examine the second scatter-gather approach and find a unique property existing in k-mer partitioning. Many k-mers are not independent from each other. Indeed, for those generated from the same short read, they have huge redundancy inside. Based on 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 p-substring ($p \leq k$). The effect is equivalent to compressing consecutive k-mers using the original sequences. We demonstrate that this compression approach does not introduce significant computing overhead, but could lead to partitions 10-15 times smaller, thus reducing I/O cost dramatically. It is observed that the size of MSP partitions is only slightly larger than the original sequences (25%-50% larger), indicating superior performance. In comparison to traditional partitioning method, the sizes of minimum substring partitions differ dramatically. We analytically derive the capacity of minimum substrings based on a random string model, which could shed light on this property. The unbalanced partition size is not a big issue since multiple small partitions can be wrapped together to form a big one.

Our main contribution is an innovative partition strategy for short sequences that completely solves the memory bottleneck in the de Bruijn graph-based assembly pipeline. Our solution is disk-based, using a very small amount of memory without significant performance trade-off. To the best of our knowledge, our study is the first work that introduces the novel concept of 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 $n-k+1$ k-mers, written as $s[1, k]$, $s[2, k+1], \ldots, s[n-k+1, n]$. Two k-mers in s, $s[i, k+i-1]$, $s[i+1, k+i]$ 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.

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.*

Since DNA sequences only contain four symbols A, C, G, T, each vertex in a de Bruijn graph built from DNA short reads can have at most four direct predecessor vertices and four direct successor vertices. In a de Bruijn graph, we only consider edges for adjacent k-mers which occur in the same short read. It is possible that the same edge may appear in multiple short reads. 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 edge appears in short

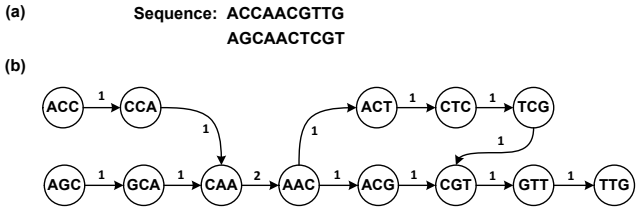


Figure 3: A de Bruijn Graph Example: k=3

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).

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*. Let S be a short read set $S = \{s_i\}$. A k-mer extracted from s_i, $s_i[j, j+k-1]$, can be written as $s_{i,j}$. Let M be the mapping function. For two k-mers extracted from s_{i_1} and s_{i_2}, s_{i_1,j_1} and s_{i_2,j_2}, if $s_{i_1,j_1} = s_{i_2,j_2}$, $M(s_{i_1,j_1}) = M(s_{i_2,j_2})$.

Once the mapping is built, by scanning the short reads, we can create the edge set for the de Bruijn graph naturally. The follow-up error-correction and assembly algorithms can make use of the graph. The mapping from a k-length string to an integer will also simplify the graph representation and reduce storage space. Therefore, the task of building a de Bruijn graph is narrowed down to k-mer mapping, which is very related to duplicate detection that aims to find all distinct k-mers. These two tasks are similar in the sense that both need to detect whether two k-mers are the same or not. However, they are different because k-mer mapping needs to merge the same k-mers from different reads to the same vertex. Nevertheless, the duplicate detection techniques can shed light on the k-mer mapping problem.

2.2 Scatter/Gather

One solution to solve memory bottleneck is to partition short reads/k-mers into several partitions so that each partition can fit in main memory and be processed separately [28, 29]. In this section, we will discuss two scatter/gather approaches derived from duplicate detection techniques and show their limitations. The first solution is called Horizontal Partition (H-Partition).

1. Divide S to disjoint partitions with equal size, S_1, S_2, ..., S_l, 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 α extracted from partition S_j, let S_i be the partition with the smallest i that contains α, then $M(\alpha) = M_i(\alpha)$.

Step 4 in H-Partition is very costly. It needs a sort/merge process to identify the duplicate k-mers in different partitions. Sorting on k-mer strings can be done at the end of the third step, when P_i is written to disk. After P_i's are merged, the mapping between (i, j) and the assigned id needs to be sorted on (i, j) (i.e, (i_1, j_1) entry will appear before (i_2, j_2) if $i_1 < i_2$ or $i_1 = i_2$, $j_1 < j_2$) so that one can build an edge file by scanning the map once. Assume the size of the short reads is N_s bytes and the total size of the k-mers extracted from the short reads is N_k bytes. In H-Partition, Step 1 and 2 need N_s bytes read (we can skip N_s bytes write by defining partition boundaries on the short read file directly). Step 3 needs N_k bytes write. Step 4 needs N_k bytes read and write + $2N_k$ bytes read and write. Since N_k is usually more than one order of magnitude larger than N_s, the total I/O cost for H-Partition is around $3N_k$ bytes read + $4N_k$ bytes write.

The major 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, one common strategy is to do bucket partitioning. Let H be a hash function of k-mer. Assume we want to generate l partitions. One can put a k-mer $s_{i,j}$ to the $H(s_{i,j}) \mod l$ partition. A simplified implementation of bucket partition is to use k-mers' last several symbols to separate 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, $S_1, S_2, \ldots, S_l$, according to $H(s_{i,j}) \mod l$.

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).

In B-Partition, Step 1 needs N_s bytes read + N_k bytes write. Step 2 needs N_k bytes read. Step 3 takes N'_k bytes write, where N'_k is the number of k-mers times 3 integer size, i.e., i, j, and the assigned id. Step 4 takes N'_k bytes read + N'_k bytes write. Since N_k is much larger than N_s and N'_k. The I/O cost will be determined by N_k bytes read and N_k bytes write. Overall, B-Partition outperforms H-Partition. Unfortunately, when N_k is very big, B-Partition could be slow. Suppose each short read has length n, then the total number of k-mers within each read would be $n - k + 1$. Therefore, N_k could reach $(n-k+1)*k*|S|$, where $|S|$ is the total number of short reads. For a hundred gigabyte short read dataset, N_k could easily reach multiple terabytes. This huge amount of data not only occupies a lot of disk space, but also causes a heavy I/O overhead.

In the next section, we are going to introduce a new partitioning concept, minimum substring partitioning (MSP), which works much more efficiently than the traditional bucket partitioning method.

3. MINIMUM SUBSTRING PARTITIONING

Bucket partitioning has high I/O overhead because it breaks each short read into many adjacent k-mers and distribute them to different partitions individually. The partition size then increases significantly. According to $H(s_{i,j}) \mod l$, adjacent k-mers will likely be distributed to different partitions unless $H(s_{i,j}) \mod l = H(s_{i,j+1}) \mod l$. Karp and Rabin [13] 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, rather than proposing a new hash function, we resort to another approach to bypass this problem.

DEFINITION 3 (SUBSTRING). *A substring of a string $s = s_1 s_2 \dots s_n$ is a string $t = s_{i+1} s_{i+2} \dots s_{i+m}$, where $0 \leq i$ and $i + m \leq n$.*

DEFINITION 4 (MINIMUM SUBSTRING). *Given a string s, a length-p substring t 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., $t \leq s'$ ($\leq$ defined by lexicographical order). 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 ($p \leq k$) 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 $s = s_1 s_2 \dots s_n$, 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 t, then one can just output substring $s_i s_{i+1} \dots s_{i+j+k-2}$ to partition $H(t) \mod l$ without breaking it to j k-mers. If j is large, this compression strategy will dramatically reduce the partition size. Our experiments on three different short read datasets show that when p is chosen appropriately (e.g., $p << k$), the total partition size is very close to the original short reads size.

DEFINITION 5 (MINIMUM SUBSTRING PARTITIONING). *Given a string $s = s_1 s_2 \dots s_n$, $p, k \in \mathbb{N}$, $p \leq k \leq n$, minimum substring partitioning breaks s to substrings with maximum length $\{s[i,j] | i + k - 1 \leq j, 1 \leq i, j \leq n\}$, s.t., all k-mers in $s[i,j]$ share the same minimum p-substring, and it is not true for $s[i, j+1]$ and $s[i-1, j]$. $s[i,j]$ is also called super k-mer.*

According to Definition 5, we can partition k-mers based on their minimum p-substrings and compress them if adjacent k-mers are distributed to the same partition. Minimum Substring Partitioning also has the following properties:

THEOREM 3.1. *Given a string $s = s_1 s_2 \dots s_n$, $p \leq k \leq n$, if all of k-mers in s have the same minimum p-substring t and $s[n-p+2, n] \geq t'$, where t' is the minimum $p-1$ substring of t, then t' is the minimum $p-1$ substring of s.*

PROOF. (1) If $k = n$, there is only 1 k-mer and the conclusion is obviously true. (2) If $k < n$, assume the conclusion is not true. Then there exists a $p-1$ substring $s[i, i+p-2]$ of s such that $s[i, i+p-2] < t'$. Since $s[n-p+2, n] \geq t'$, we have $s[i, i+p-2] \neq s[n-p+2, n]$. So $s[i, i+p-2]$ is not the last $p-1$ substring of s and $s[i, i+p-1]$ is a legal p substring of s. Let t_l and t_r be the first and last $p-1$ substring of t, respectively. Then $t_l \leq t_r$; otherwise the p substring starting with t_r will be smaller than t, which contradicts that t is the minimum p-substring of all k-mers in s. So $t' = t_l$. Thus from $s[i, i+p-2] < t'$ we have $s[i, i+p-1] < t$. Since all k-mers in s have the same minimum p-substring t, this is contradicted. Therefore the assumption is not true. Hence t' is the minimum $p-1$ substring of s. □

THEOREM 3.2. *Given a string $s = s_1 s_2 \dots s_n$, $p \leq k \leq n$, if all the k-mers in s have the same minimum p-substring t, t only appears once in s, and $s[k-p+1, k] \neq t$, then all the k-mers in s will have the same minimum $p+1$ substring.*

PROOF. Without loss of generality we assume $t = s[i, i+p-1]$. Since $s[k-p+1, k] \neq t$, $s[i, i+p]$ is a legal $p+1$ substring of all the k-mers in s. Now we prove that $s[i, i+p]$ is their minimum $p+1$ substring. Assume this conclusion is not true, which means there exists a $p+1$ substring $s[j, j+p]$ of s such that $s[j, j+p] < s[i, i+p]$. Then we have two situations: either (1) $s[j, j+p-1] < s[i, i+p-1]$ or (2) $s[j, j+p-1] = s[i, i+p-1]$ and $s[j+p] < s[i+p]$. The first situation will not happen since $s[i, i+p-1]$ is the minimum p substring. The second one will not happen either since $s[i, i+p-1]$ only appears once in s. Therefore, the assumption is not true. $s[i, i+p]$ is indeed the minimum $p+1$ substring of all the k-mers in s. Hence all the k-mers in s have the same minimum $p+1$ substring. □

Theorem 3.1 shows that except one special case (as illustrated in Figure 4, the first 4 k-mers share the same minimum 3-substring, but the 5_{th} k-mer's minimum 3-substring is different from the previous 4 k-mers' minimum 3-substring), if all the k-mers in a string have the same minimum p substring, then they will also have the same minimum $p-1$ substring. The vice-versa is not true as indicated in Theorem 3.2. It means that larger p will likely break a sequence to several segments with different minimum p-substrings, thus increasing the total partition size. However, it also indicates that smaller p will introduce larger partitions that might not be able to fit in the main memory.

Figure 5 shows the distribution of partition size with $p = 4$ on the bee, fish, cladonema and bird datasets. The partitions are sorted according to their sizes. We can see that very few k-mers could have large p-substrings (measured by

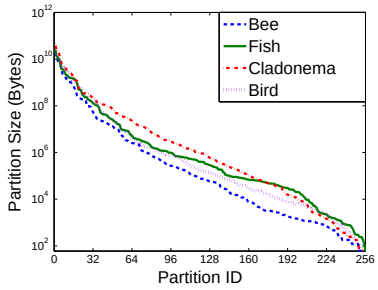


Figure 5: Partition Size Distribution

lexicographical order) as their minimum p-substring. It is also clear from the figure that there are several large dominating partitions. In the extreme case of $p = 1$, the size of the largest partition is almost as same as that of the short read dataset if we assume the symbols are distributed uniformly and randomly. In that case, we lose the point of doing partitioning. On the other hand, a larger p will make partitions smaller at the cost of decreasing the probability that adjacent k-mers sharing the same minimum p-substring, thus increasing the total partition size. In the extreme case of $p = k$, the result of minimum substring partitioning will be very close to that of bucket partitioning. Therefore one needs to make a tradeoff between the largest partition's size and the I/O overhead determined by p. Fortunately, as experimented in Section 7, there is a quite wide range of values that p can choose without affecting the performance of MSP.

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

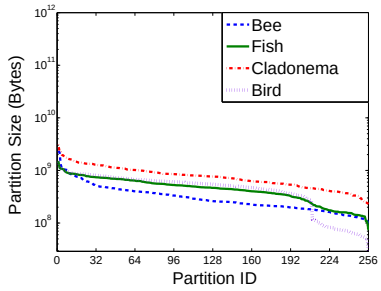


Figure 6: Partition Wrapping

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 mapped to a wrapped partition. The variance of partition sizes will likely decrease. Figure 6 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. However, the size distribution of partitions is more uniform.

Given a string s, $p \leq k \leq |s|$, minimum substring partitioning breaks s into multiple substrings $s[i_1, j_1]$, $s[i_2, j_2]$, ..., $s[i_l, j_l]$. Assume the average number of breaks in each short read is l for a given sequence dataset. We have the following theorem.

THEOREM 3.3. *The total partition size is $\frac{lk}{m}n + n$.*

PROOF. Omitted. □

As verified by real datasets and a random string model in Section 6, it is observed that lk is often close to m, indicating that the total partition size is $\Theta(n)$, far smaller than $\Theta(kn)$ in H-Partition and B-Partition.

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 $C \leftrightarrow G$). For each DNA sequence, its corresponding read in the other direction is called *reverse complement* and they are considered equivalent in bioinformatics. Most sequencing techniques extract short reads from DNA sequences in either direction. So 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. Here we resort to another solution to avoid this problem.

DEFINITION 7. *[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 7 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 discussions, 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 with the adoption of the minimum substring partitioning. 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 corresponding to the k-mers in short read sequences. Each distinct k-mer id represents a node in the de Bruijn graph, with each pair of adjacent k-mer id's in the sequences representing an edge in the graph.

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 t, (3) generate substrings of s such that k-mers in each substring have the same minimum p-substring. This method has to find the minimum p-substring for each k-mer, which introduces $(k-p+1)$ p-substring comparisons. Let n be the length of s. Since $k >> p$ and $n >> k$, this approach needs to perform $(k-p+1)*(n-k+1)$=$O(nk)$ p-substring comparisons, which is not efficient.

The above solution does not utilize the inherent overlaps among k-mers. When we slide the k-size window 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 $n-p+1$ p-substrings in s, the number of p-substring comparisons is $O((n-p+1)\log(k-p+1))$=$O(n\log k)$.

While the priority queue is theoretically good, the overhead introduced by the queue structure could be high given that k and n are quite small in real-life applications. We thus introduce a simplified algorithm, as described in Algorithm 1. When k-size window slides through s and a minimum substring changes at position j of s, we output a string segment of s that contains the previous minimum substring (we omit this part of code in Algorithm 1).

Algorithm 1 Minimum Substring Computing

```
Input: String s = s_1 s_2 ... s_n, 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 n − 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
```

In Algorithm 1, initially when the window starts at position 1, we scan the window to find the minimum p-substring, say min_s, and the start position of min_s, say min_pos. Then we slide the window towards right, one symbol each time, till the right bound of the window reaches the end of the short read. After each sliding, we test whether the min_pos is still within the range of the window. If not, we have to re-scan the window to get new min_s and min_pos. Otherwise, we test whether the last p-substring of the current window is smaller than current min_s. If yes, we set this last p-substring as new min_s and its start position as new min_pos. If not, we just keep the old min_s to calculate the partition boundary. As described in last section, the neighboring k-mers will likely contain the same minimum p-substring. Therefore, we do not have to re-scan the window very often. The worst case time complexity is $O(nk)$ p-substring comparisons. Theorem 5.1 shows this algorithm is more efficient in reality when s is broken to a few pieces. This is very true in minimum substring partitioning of real short reads. For example, in the Lake Malawi cichlid (fish) dataset, with $p=6$ and $k=59$ we get the average number of breakdowns for $|s|=101$ is $l=2.52$.

THEOREM 5.1. *Given a string $s=s_1s_2\ldots s_n$, $p,k\in\mathbb{N}$, $p\le k\le n$, minimum substring partitioning breaks s into substrings $s[i_1,j_1]$, $s[i_2,j_2]$, ..., $s[i_l,j_l]$, $i_1<i_2<\ldots<i_l$. Algorithm 1 needs at most $n+(l-1)k-pl+1$ p-substring comparisons.*

PROOF. Algorithm 1 shows min_s and min_pos will 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, we have to re-scan the k-mer $s[i,i+k-1]$, which introduces (k-p+1) p-substring comparisons. Under the second condition, we only have to compare the last p-substring of $s[i,i+k-1]$ with the current min_s, which only involves 1 p-substring comparison. Since the string s is broken into l substrings, min_s and min_pos have changed for $(l-1)$ times. If all these $(l-1)$ changes are due to the first condition, we have to perform the largest number of p-substring comparisons. Therefore the total number of k-mer scans is l, including the initial scan of the first k-mer. These l k-mer scans will introduce $(k-p+1)*l$ p-substring comparisons. Within each of these l substrings, we need (n_m-1) p-substring comparisons to test the second condition, where n_m is the number of k-mers within the substring $s[i_m,j_m]$. For all l substrings, the total number of p-substring comparisons due to this test is $\Sigma_{m=1}^{l}(n_m-1)=(n-k+1-l)$. Therefore the total number of p-substring comparisons of Algorithm 1 is at most $(k-p+1)*l+(n-k+1-l)=n+(l-1)k-pl+1$. □

5.2 Mapping

In this step, we assign each distinct k-mer a unique integer id as its corresponding vertex id in the de Bruijn graph. A straightforward solution is to process each partition one by one. For each partition, break the super k-mers into k-mers and insert these k-mers into a hash table. Since adjacent k-mers are only different by the first and last symbol, rolling hash functions [13, 7] or direct bit operations (A, C, G, T can be encoded using 2 bits) can be applied here to improve the performance. Whenever we see a new k-mer, we first look up the hash table to see if it is already in the hash table: if yes, we do nothing; otherwise, we put this k-mer along with an assigned id into the hash table. The starting id of k-mers in one partition is the maximum k-mer id of the previous partition plus one. After processing one partition, write the entries in hash table to a disk id file and release the memory occupied by that hash table. This approach works well for the mapping step. However it will cause a serious problem for the merging step.

In the merging step, we have to scan the short reads. For each k-mer, we have to find the id file recording its id and load that file into memory. This will cause a lot of I/O operations and seriously slow down the process. In order to avoid this, we develop an id replacement strategy instead of using direct id mapping. During the partitioning step, each k-mer is assigned an integer id. The id's are assigned increasingly from 1. When we write each super k-mer into partitions, we also write its first k-mer's id along with it so that the id's of the follow-up k-mers within this

super k-mer can be inferred from the first k-mer's id. In the mapping step, for each partition, we create a hash table in the memory and create a disk id replacement file as the k-mer id replacement table. Whenever we see a new k-mer, we first look up the hash table to see if it is already in the hash table: if yes, we write a replacement record (indicating the current k-mer is a duplicate and we have to replace its pre-assigned id with the id associated with its first occurrence) into the id replacement file; otherwise, we put this k-mer along with its pre-assigned id into the hash table.

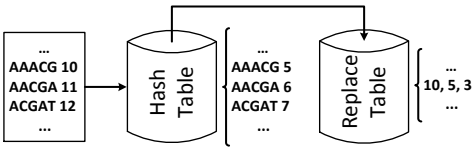


Figure 7: ID Replacement and Range Compression

Figure 7 shows an example of the id replacement process, where the 10_{th} k-mer can be replaced by the 5_{th}, the 11_{th} by the 6_{th}, the 12_{th} by the 7_{th}. Through several experimental tests, we find that the id replacement table can be very long, which increases the I/O workload. To solve this problem, we develop an id range compression heuristic: write the replacement records as a range instead of multiple individual records. If several consecutive replacement records are incremental, we can write the range of them into the table. For the example shown in Figure 7, instead of outputting three records, one can just output one range record, $10, 5, 3$, meaning the 3 consecutive id's starting at 10 will be replaced by 3 consecutive id's starting at 5. Range compression is quite effective since high throughput sequencing technologies often cover a DNA sequence many times, resulting in many highly overlapping short reads. Based on our test, this kind of compression will generate id replacement tables whose total size is in the same level with the raw short reads.

5.3 Merging

After obtaining the id replacement tables, the last step is merging. In this step, we merge all the replacement tables to generate one big table. Since all the replacement tables generated in the Mapping step are already naturally sorted in increasing order by the first entry (the pre-assigned id's to be replaced) of each replacement record, the merged table is also sorted in that order. Then we can enumerate id's from 1 to the largest one in the Mapping step and do replacements with respect to the merged table. If the replacement table indicates that the pre-assigned id of the current k-mer should be replaced, we write the replacing id for this k-mer and move to next replacement record; otherwise we just write this k-mer's pre-assigned id for it. After finishing this enumeration and the corresponding replacements, we will get the de Bruijn graph in the form that the short read sequences represented by the corresponding sequences of k-mer id's, with each pair of adjacent k-mer id's in the sequences representing an edge in the graph.

6. PROPERTIES

The minimum substring partitioning technique is a disk-based approach. Different from the traditional partition/merge approaches, it leverages the overlaps among the data to reduce the partition size, thus having minimal impact on running time, while solving the memory bottleneck. It is important to further study the properties of MSP to fully understand this methodology. Here, we assume a random string model with four symbols A, C, G, and T, which may occur with different probabilities. The result can be extended to any number of symbols.

6.1 Capacity of Minimum P-substrings

The capacity is defined as the percentage of k-mers represented by a minimum p-substring. It determines the number of distinct k-mers contained in the partitions derived by MSP, i.e., the size of the corresponding hash table when a partition is loaded into memory. Since MSP has to load each partition into main memory, the capacity determines the peak memory. While a close formula is not available yet, we design an efficient polynomial-time algorithm (See the Appendix) for computing the probability that a given p-substring is the minimum p-substring of a random n-length string S.

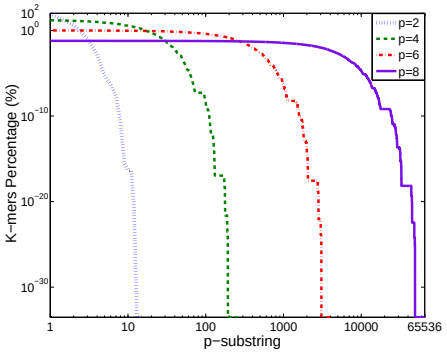


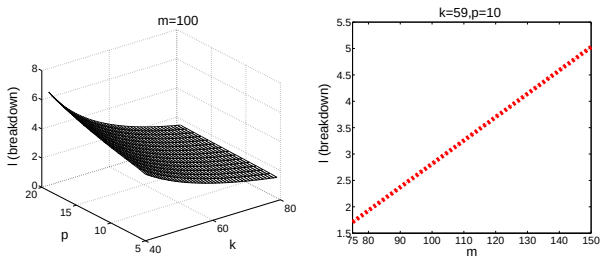
Figure 8: Expected Partition Size Distribution

Figure 8 shows the expected distribution of k-mers with respect to different p values, assuming 4 bases A, C, G, T appear with equal probability and k-mer length being 59. Here the p-substrings are sorted according to the percentage of k-mers they cover. And the figure uses logarithm on both axes. The result clearly shows a property we discussed before: when p increases, there is a plateau where many p-substrings cover a similar percentage of k-mers. We can conclude that there is no memory consuming partition when p is large. Furthermore, the peak memory of MSP can be fully controlled by varying p.

6.2 Average Number of Breaks

The average number of breaks l in a short read determines the total partition size, which is equal to $\frac{nkl}{m} + n$. We use a simulation method for computing the average number of breaks. It generates $1M$ short reads, runs MSP and average the break numbers.

Figure 9(a) shows the expected breaks of short reads with respect to different p and k values, assuming 4 bases A, C, G, T appear with equal probability and the short read length m being 100. When p increases, the number of breaks increases. When k increases, the number of breaks decreases. Figure 9(b) shows the expected breaks of short reads with respect to different m values, where the minimum substring length is set at 10 and k is set at 59. It is observed that



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

Figure 9: Average Number of Breaks

the average number of breaks is very small and it increases proportionally with respect to m. Since kl is close to m, in practice, the total partition size is $2n \sim 3n$. Note that this is much better than kn claimed by [15]

7. EXPERIMENTS

In this section, we present experimental results which illustrate the memory efficiency, effectiveness and inherent properties of the minimum substring partitioning method on four very large real-life short reads datasets: cladonema, bumblebee, fish, and bird. (1) We first analyze the efficiency of our graph construction algorithm by reporting the memory and time cost of building a de Bruijn graph and compare it with two sequence assembly programs, Velvet [30] and SOAPdenovo [16]. (2) The performances of MSP and two traditional partition/merge algorithms, H-Partition, B-Partition, are compared in terms of partition size and runtime. (3) We also change different parameters to illustrate interesting properties of minimum substring partitioning. All the experiments, if not specifically mentioned, are conducted on a server with 2.40GHz Intel Xeon CPU and 512 GB RAM, using a single thread.

7.1 Data Sets

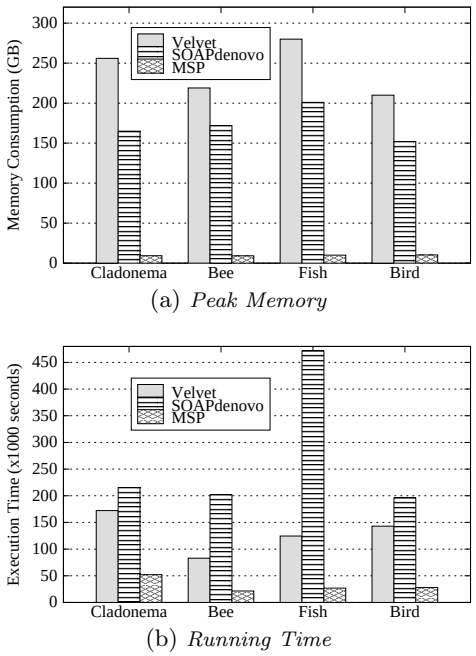
Four real-life short reads datasets are used to test our algorithms. The first one is the sequence data of Cladonema provided by our collaborators. The second one is the sequence data of Bombus impatiens (bee). The third one is the sequence data of Lake Malawi cichlid (fish). And the last one is the sequence data of Budgerigar (bird). 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. Some basic facts about the four datasets are shown in Table 1.

	Cladonema	Bee	Fish	Bird
Size(GB)	126.3	45.8	76.3	58.3
Avg Read Length(bp)	101	124	101	150
# of Reads(million)	894	303	598	323

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

7.2 Efficiency

We first conduct experiments to compare MSP with two real sequence assembly programs: Velvet [30], a classic de Bruijn graph based assembler, and SOAPdenovo [16], a highly optimized and leading assembler. For all the experiments, we set the k-mer length to 59 [6]. For MSP, we partition the short reads into 1000 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, though with only 1 thread it will be 3 times slower. Both Velvet and MSP use 1 thread.



(a) *Peak Memory*

(b) *Running Time*

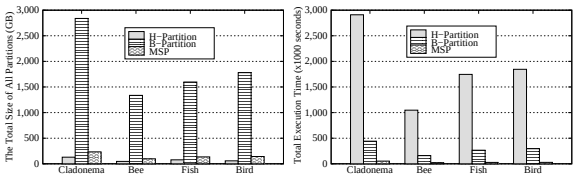
Figure 10: Velvet, SOAPdenovo, MSP

Figure 10 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 100G memory, while our method can complete the task with less than 10G memory, an order of magnitude reduction of memory usage.

7.3 Effectiveness

We then conduct experiments to compare MSP with other two partition/mergign algorithms, H-Partition and B-Partition. For all the three methods, we set the k-mer length to 59 and partition short reads into 1000 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 11(a) and 11(b) show the size of partitions on disk and the total running time of constructing a de Bruijn graph.

Figure 11 shows MSP outperforms the two baseline methods: compared with B-Partition, MSP can reduce the total partition size by 10-15 times and reduce the partition time by 7-9 times. B-Partition was adopted by out-of-core algorithms such as [15]. It implies that MSP is better than the classic approach that does not consider the overlaps among data records. H-Partition has the smallest partition size (slightly smaller than MSP's). Unfortunately, its overall performance is the worst since it needs multiple disk scans and sorts.



(a) *Partition Size* (b) *Total Time*

Figure 11: H-Partition, B-Partition, MSP

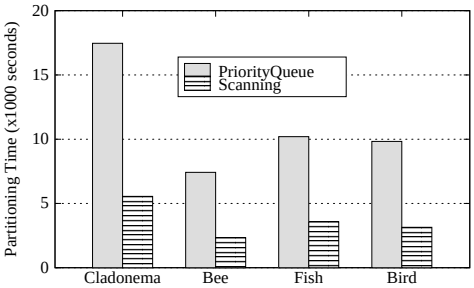


Figure 12: Scanning vs. Priority Queue

We then illustrate the advantage of using a scanning method (Algorithm 1) over a priority queue approach in the minimum substring partitioning step. Here we set the k-mer length as 59 and partition the short reads into 256 wrapped partitions with the minimum substring length p being 6. Figure 12 shows using the simple scanning method in Algorithm 1 is around 2 times faster than the priority queue approach.

7.4 Properties of MSP

Next we conduct experiments to illustrate the properties of minimum substring partitioning. Figure 13 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 1000 wrapped partitions. It shows that increasing minimum substring length will decrease the peak memory significantly. It will also make the total partition size slightly larger and accordingly make the running time slightly longer. However, both increases are negligible, indicating that MSP is not very sensitive to the setting of p and there is a wide range of values we can choose.

We then fix the minimum substring length at 10 and the number of partitions at 1000, and vary the length of k-mers. Figure 14 shows the change of peak memory, partition size, and running time with respect to k-mer length. It shows that the peak memory is within a small range when k changes, indicating that the peak memory is not very sensitive to the value of k. It is also observed that increasing k will make the total partition size decrease and accordingly make the running time decrease. There are two effects inside. Given n short reads with length L, the total size of all the k-mers is equal to $k(L-k+1)n$. We have

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

Hence, the size is peaked when $k = (L+1)/2$, i.e., 51 for cladonema and fish, and 63 for bee. The second effect is the compression ratio of MSP for larger k is higher than smaller k. These two figures demonstrate the second effect dominates, since we do not observe a peak at $k = (L+1)/2$.

7.5 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 1000 wrapped partitions with the minimum substring length p being 10.

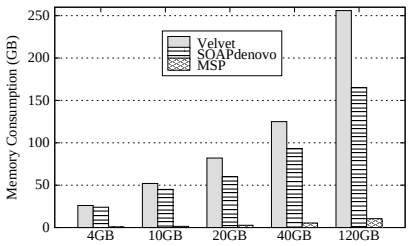


Figure 15: Scalability: Peak Memory

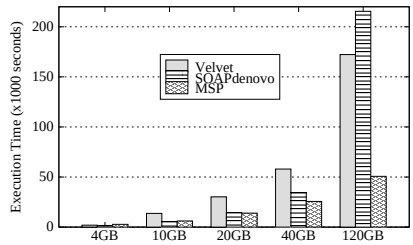


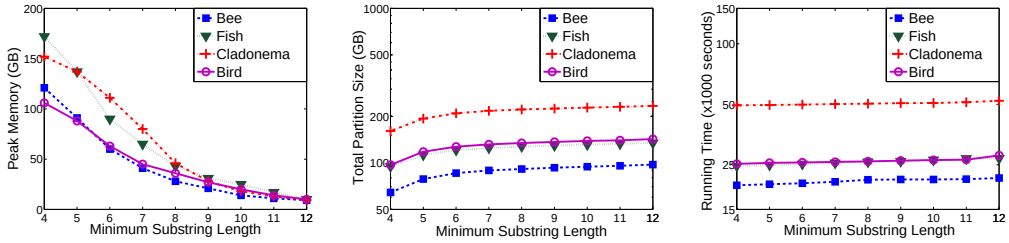
Figure 16: Scalability: Running Time

Figures 15 and 16 show that the memory consumption of MSP doesn't change very much with respect to the increasing size of data. The other two approaches do not have this nice property. All the three algorithms scale linearly in terms of running time.

8. RELATED WORK

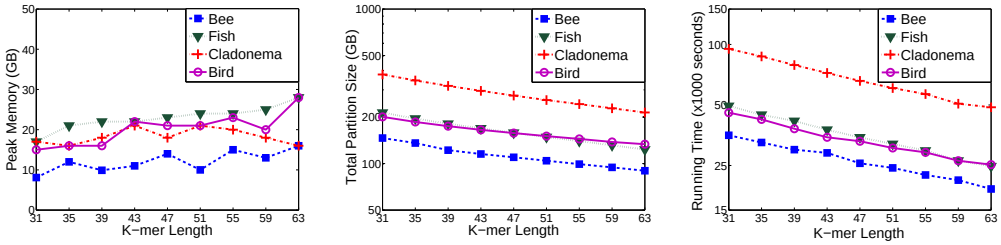
High throughput sequencing technologies are generating tremendous amounts of short reads data. Assembling these datasets becomes a critical research topic. In the area of overlap-layout-consensus sequence assembly, algorithms such as Celera [22], ARACHNE [3], PCAP [12], Phusion [20], etc. have been developed. With the development of next-generation sequencing techniques, the de Bruijn graph sequence assembly approaches became popular, including Euler[23], Velvet[30], AllPaths[6], SOAPdenovo[16], 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 read sequences 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 [26] adopted FM-index [11] to achieve compression in building the string graph [21], which is an alternative graph formulation used in sequence assembly (string graph is much more



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

Figure 13: Varying Minimum Substring Length p



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

Figure 14: Varying K-mer Length k

expensive to construct than de Bruijn graph, so it is not that popular as de Bruijn graph). However, the step of building the suffix array and FM-index in their method is very time-consuming and memory-intensive. Orthogonally, Conway and Bromage [8] 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[27] and Contrail[25]. 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 high I/O overhead and heavy 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 [5]. Teuhola and Wegner [29] proposed an $O(1)$ extra space, linear time algorithm to detect and delete duplicates from a dataset. Teuhola [28] 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 [9] 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. Duplicate detection has also been examined in different contexts, e.g., stream [18] and text [4]. A survey for general duplicate record detection solutions was given by Elmagarmid, Ipeirotis and Verykios [10].

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.

9. 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 very compact partitions. This partitioning technique was successfully applied to de Bruijn graph construction with very small memory footprint. We also discussed the properties 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 show that it can not only successfully finish the tasks on very large datasets within a reasonable amount of memory, but also achieve better performances than existing state-of-the-art algorithms.

10. REFERENCES

[1] http://www.appliedbiosystems.com.

[2] http://www.illumina.com.

[3] S. Batzoglou, D. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. Mesirov, and E. Lander. Arachne: a whole-genome shotgun assembler. *Genome research*, 12(1):177–189, 2002.

[4] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.

[5] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. *ACM Trans. Database Syst.*, 8:255–265, 1983.

[6] 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.

[7] J. D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15:291–320, 1997.

[8] T. Conway and A. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

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

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

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

[12] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. Pcap: a whole-genome assembly program. *Genome research*, 13(9):2164–2170, 2003.

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

[14] D. Kelley, M. Schatz, and S. Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):R116, 2010.

[15] 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.

[16] 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.

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

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

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

[20] J. Mullikin and Z. Ning. The phusion assembler. *Genome research*, 13(1):81–90, 2003.

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

[22] 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.

[23] 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.

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

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

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

[27] 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.

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

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

[30] 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 ($j \le i$), namely, $s_{i-j+1} \dots s_i$ (see Figure 17). 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 $<, \le, =, \ge, >$ 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

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

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 $i = m, m+1, \ldots, n$. 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 $\le W$. 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.

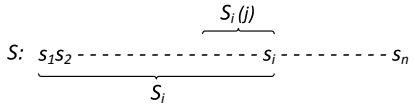


Figure 17: Illustration of S, S_i, and the j-suffix of S_i.

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.

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:

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

- $W_{m-1}(j-1) \leq W_{j-1}$. 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 $O(n^2)$. □