Succinct de Bruijn Graphs

Alexander Bowe[1], Taku Onodera[2], Kunihiko Sadakane[1], and Tetsuo Shibuya[2]

[1] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku,
Tokyo 101-8430, Japan
{alex,sada}@nii.ac.jp
[2] Human Genome Center, Institute of Medical Science,
University of Tokyo 4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan
{tk-ono,tshibuya}@hgc.jp

Abstract. We propose a new succinct de Bruijn graph representation. If the de Bruijn graph of k-mers in a DNA sequence of length N has m edges, it can be represented in $4m + o(m)$ bits. This is much smaller than existing ones. The numbers of outgoing and incoming edges of a node are computed in constant time, and the outgoing and incoming edge with given label are found in constant time and $\mathcal{O}(k)$ time, respectively. The data structure is constructed in $\mathcal{O}(Nk \log m / \log\log m)$ time using no additional space.

1 Introduction

Within the last two decades, assembling a genome from enormous amount of reads from various DNA sequencers has been one of the most challenging and important computational problems in molecular biology. Though the problem is proved to be NP-hard [14], many algorithms have been proposed for the problem (see the surveys [10,13,19]). Most of these algorithms follow a so-called Overlap-Layout-Consensus strategy, where an algorithm first finds overlaps between reads, next layouts these reads, and finally finds the consensus genome. These algorithms can be categorized into two types, due to the graph used in the overlap phase.

Most old-time assembly algorithms (especially for the long Sanger reads) first construct a graph called the *overlap graph* after finding the overlapping pairs of reads, where each node represents a read and edges are constructed between nodes *iff* the corresponding two reads have an overlap of enough length [1,9,15]. But this strategy is difficult to apply against the huge data from more recent epoch-making next-generation sequencers (NGSs). The NGS machines can sequence vast amount of genome data. It makes it computationally very hard to compare all the pairs of reads. Moreover, most NGSs cannot read long DNA fragments (*e.g.*, at most 200bp in the case of Illumina HiSeq2000), and their read lengths are not long enough to detect overlaps with enough lengths between reads. To conquer these problems, many recent assembler algorithms utilize a graph called the *de Bruijn graph* in the overlap phase [11,12,18,21,22,24], instead of the overlap graph.

B. Raphael and J. Tang (Eds.): WABI 2012, LNBI 7534, pp. 225–235, 2012.

A de Bruijn graph is a graph where each node represents a k-mer (a substring of length k) that exists in the reads, and an edge exists *iff* there is an exact overlap of length $k-1$ between the corresponding k-mers. The de Bruijn graph can be constructed more efficiently than the overlap graph in many cases, but the overlap phase is still the bottleneck of most assembly algorithms based on the de Bruijn graph. This is because storing the de Bruijn graph requires huge amount of memory. Thus we focus on reducing the memory required for the de Bruijn graph in this paper.

There have been proposed only two data structures for reducing the size of memory for the de Bruijn graph. The succinct data structure proposed by Conway and Bromage [5] is a data structure that straightforwardly represents the de Bruijn graph by a bit vector. Its representation should be smaller than a naive ordinary implementation of the de Bruijn graph, but it still requires $O(m \cdot k)$ memory, where k is the k-mer length and m is the number of edges in the de Bruijn graph, which means it would be very large when k is large. The other data structure is by Ye et al. [23], which stores only a subset of nodes of the de Bruijn graph to save memory, but it is not actually the de Bruijn graph.

In this paper, we propose a new succinct representation of a de Bruijn graph which only requires $m(2+\log\sigma)$ bit to store[1], where σ is the alphabet size (*i.e.*, $\sigma = 4$ in the case of DNA). The size of this representation is not affected by the value of k and is much smaller than either of the two previous methods. Moreover we will present the algorithm to construct the data structure on-line. Our main result is summarized as follows:

Theorem 1. *The k-dimensional de Bruijn graph of M string of total length N on an alphabet of size σ can be stored in $m(2+\log\sigma) + \mathcal{O}((\sigma+M)\log m) + o(m\log\sigma)$ bits where m is the number of edges in the graph. The numbers of outgoing and incoming edges of a node are computed in $\mathcal{O}(\log\sigma/\log\log m)$ time, and the outgoing and incoming edge with given label are found in $\mathcal{O}(\log\sigma/\log\log m)$ time and $\mathcal{O}(k\log^2\sigma/\log\log m)$ time, respectively. The node for a given k-mer is found in $\mathcal{O}(k\log\sigma/\log\log m)$ time. If $\sigma = \mathrm{polylog}(m)$, the time complexities become $\mathcal{O}(1)$, $\mathcal{O}(1)$, $\mathcal{O}(k\log\sigma)$, and $\mathcal{O}(k)$ time, respectively.*

Theorem 2. *The k-dimensional de Bruijn graph of a string of length N can be constructed in $\mathcal{O}\left(Nk\cdot\frac{\log m}{\log\log m}(1+\frac{\log\sigma}{\log\log m})\right)$ time using no additional space. This representation can be converted to the static one in $\mathcal{O}\left(\frac{m\log m}{\log\log m}(1+\frac{\log\sigma}{\log\log m})\right)$ time.*

For DNA sequences ($\sigma = 4$), the succinct de Bruijn graph can be constructed in $\mathcal{O}(Nk\log m/\log\log m)$ time and its space becomes $4m + o(m)$ bits. This is much smaller than existing ones. For example, the succinct representation of Conway and Bromage [5] uses 40.8GB for storing a de Bruijn graph with $m =$ 12,292,819,311 edges and $k = 27$ (28.5 bits per edge). On the other hand, if we use an efficient implementation of *rank*/*select* data structures [17] for our representation, the estimated size is less than 5 bits per edge. Therefore the above graph is stored in less than 8GB.

[1] The base of logarithm is 2.

2 Preliminaries

2.1 de Bruijn Graphs

In the original definition [2], the k-dimensional de Bruijn graph of σ symbols is a directed graph representing overlaps between strings of symbols defined as follows. The graph has σ^k nodes, consisting of all length-k strings of the symbols. A node is denoted by $(u_1, \ldots, u_k)$ where $u_1, \ldots, u_k$ are symbols. For any pair of nodes $u = (u_1, \ldots, u_k)$ and $v = (v_1, \ldots, v_k)$ such that $u_2 = v_1, u_3 = v_2, \ldots, u_k = v_{k-1}$, the graph has a directed edge from u to v labeled with v_k. In this paper we call it the complete k-dimensional de Bruijn graph of σ symbols.

The de Bruijn graphs considered in this paper are subgraphs of the complete de Bruijn graph. We define the k-dimensional de Bruijn graph of a string T as follows. The nodes of the graph correspond to all length-k substrings of T. If the string is of length N, the graph has at most $N - k + 1$ nodes. The edges of the graph are defined in the same way as the complete de Bruijn graph. For convenience, we add k characters \$ at the head of the string, and a \$ at the end.

We can also store a set of M strings $T_1, \ldots, T_M$ as follows. We append a terminator $\$_i$ to the tail of each string T_i, and concatenate all the strings. Then we add k characters $\$_0$ at the head. Figure 1 shows an example.

2.2 Basic Succinct Data Structures

Let $T = T[1]T[2] \cdots T[N]$ be a string of length N on alphabet $\mathcal{A}$, that is, $T[i] \in \mathcal{A}$ for any $i = 1, \ldots, N$. Let $\sigma = |\mathcal{A}|$ denote the alphabet size. We can store T in $N\lceil \log_2 \sigma \rceil$ bits. The space does not depend on the word size of CPU. We can retrieve any character $T[i]$ in constant time using bit operations on words.

The most basic succinct data structure is the one for computing *rank*, *select*, and *access* values on strings, which are defined as follows. The value $access(T, i)$ returns $T[i]$ for $1 \leq i \leq N$. The value $rank_c(T, i)$ where $c \in \mathcal{A}$ and $1 \leq i \leq N$ is the number of c's in $T[1] \cdots T[i]$. For any T and c we define $rank_c(T, 0) = 0$. The value $select_c(T, j)$ where $c \in \mathcal{A}$ and $1 \leq j \leq rank_c(T, N)$ is the position of j-th c in T. For any T and c we define $select_c(T, 0) = 0$ and for any $j > rank_c(T, N)$ $select_c(T, j) = N + 1$. Let $t_r(N, \sigma)$, $t_s(N, \sigma)$, and $t_a(N, \sigma)$ denote the time complexity for computing *rank*, *select*, and *access*, respectively, on a string of length N and alphabet size σ. For brevity, we assume that for any $N_1 \leq N_2$, $t_r(N_1, \sigma) \leq t_r(N_2, \sigma)$ and for any $\sigma_1 \leq \sigma_2$, $t_r(N, \sigma_1) \leq t_r(N, \sigma_2)$. Let $t_b(N, \Sigma)$ denote the maximum of $t_r(N, \sigma), t_s(N, \sigma), t_a(N, \sigma)$.

For convenience, we define $pred_c(T, i) = select_c(T, rank_c(T, i))$ which is the position of the first occurrence of c when we scan T from the position i to the head, and $succ_c(T, i) = select_c(T, rank_c(T, i - 1) + 1)$ which is the position of the first occurrence of c when we scan T from the position i to the end. If $T[i]$ is the first (last) occurrence of c, *pred* (*succ*) returns 0 ($N + 1$).

There exist many succinct data structures for *rank* and *select* on strings. Among them, we use the one by Ferragina et al. [8] for the static case (the case the string does not change). A string of T length n on an alphabet of size σ can

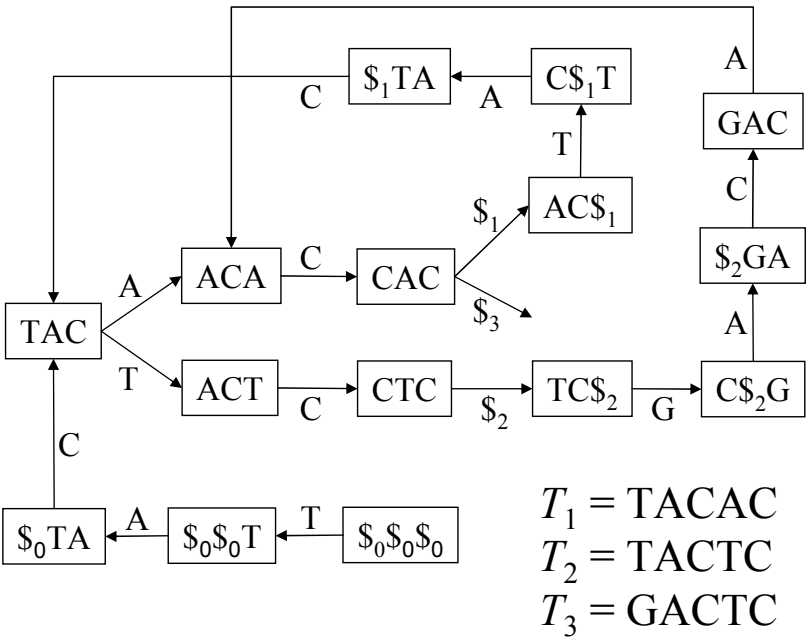

Fig. 1. The 3-dimensional de Bruijn graph of strings 'TACAC', 'TACTC', and 'GACTC'

be stored in $nH_0(T) + \mathcal{O}(\sigma \log n) + o(n \log \sigma)$ bits so that *rank*, *select* and *access* queries take $\mathcal{O}(\log \sigma / \log \log n)$ time, where $H_0(T)$ denotes the order-0 entropy of the string. Note that if the alphabet size σ is polylog(n), the queries are done in constant time. For a binary alphabet case, we can use a simpler data structure that has the same time and space complexities [20].

For the dynamic case where the string is modified by inserting or deleting a character, we use the one by Navarro and Sadakane [16] which stores the string in $nH_0(T) + \mathcal{O}(\sigma \log n) + o(n \log \sigma)$ bits so that *rank*, *select* and *access* queries and insertion and deletion of a character take $\mathcal{O}(\frac{\log n}{\log \log n}(1 + \frac{\log \sigma}{\log \log n}))$ time. For polylog-sized alphabets, the operations are done in optimal $\mathcal{O}(\log n / \log \log n)$ time. The time complexities for insert and delete are denoted by $t_u(n, \sigma)$.

2.3 The XBW Data Structure

The XBW-transform [6] is a method for compressing and indexing labeled trees. It is an extension of the Burrows-Wheeler transform [3] used for compressing and indexing strings. Given a rooted tree with n nodes where each node has a label in the set of size σ, the XBW-transform converts the tree into a representation of $2n + n \log \sigma$ bits. The size of the representation matches the information-theoretic lower bound. We can support tree navigational operations by adding small-size auxiliary indexes.

Because the XBW is for storing a tree, we cannot use it directly for storing de Bruijn graphs, which is a cyclic graph. This paper proposes a new compact representation of de Bruijn graphs of strings.

3 Succinct de Bruijn Graphs

Let G be a k-dimensional de Bruijn graph of a string T of length N on alphabet $\mathcal{A}$. Let n and m be the numbers of nodes and edges of G, respectively. A succinct representation of G supports the following operations:

- *Outdegree*(v) returns the number of outgoing edges from node v.
- *Outgoing*(v, c) returns the node w pointed to by the outgoing edge of node v with edge label c. If no such node exists, it returns -1.
- *Indegree*(v) returns the number of incoming edges to node v.
- *Incoming*(v, c) returns the node $w = (w_1, \ldots, w_k)$ such that there is an edge from w and v and $w_1 = c$. If no such node exists, it returns -1.
- *Index*(s) returns the index i of the node whose label is the string s of length k.

We define $\mathcal{A}^-$ as any set of size $|\mathcal{A}|$ such that $\mathcal{A}^- \cap \mathcal{A} = \emptyset$. Let c^- denote an element of $\mathcal{A}^-$ corresponding to an element $c \in \mathcal{A}$. We also define a function u as $u(c^-) = c$ for any $c^- \in \mathcal{A}^-$ and $u(c) = c$ for any $c \in \mathcal{A}$. We assume that the function is evaluated in constant time.

3.1 The Succinct Representation

The representation consists of the following components:

- a string $W = W[1]W[2] \cdots W[m]$ where each character is from $\mathcal{A} \cup \mathcal{A}^-$.
- a string *last* of length m on the binary alphabet $\{0, 1\}$.
- an array F of length $\sigma = |\mathcal{A}|$.

An example is shown in Figure 2.

The string W is defined as follows. Each character $W[i]$ represents the label of an edge of G. Each edge $u \to v$ of G is associated with the node label of u. Those edge labels are sorted in the lexicographic order of reversals of associated node labels. Ties are broken by edge labels. Let *Node*$[i]$ denote the node label for $W[i]$. This is not explicitly stored.

The string *last* is defined as *last*$[i] = 1$ if $i = n$ or *Node*$[i]$ is different from *Node*$[i + 1]$, or *last*$[i] = 0$ otherwise. From this definition, all node labels *Node*$[i]$ with *last*$[i] = 1$ are distinct, and those indices i have one-to-one correspondence with the nodes of G. Therefore we use an index i of the strings such that *last*$[i] = 1$ to represent a node v. Let n denote the number of nodes.

The array F stores cumulative frequencies of the last characters of node labels. Namely, for any $c \in \mathcal{A}$, $F[c] = |\{i \mid 1 \leq i \leq m, C(i) < c\}|$ where $C(i)$ denotes the last character of *Node*$[i]$. Because $F[\$_i] = i$ for $i = 0, 1, \ldots, M$, we need not store them.

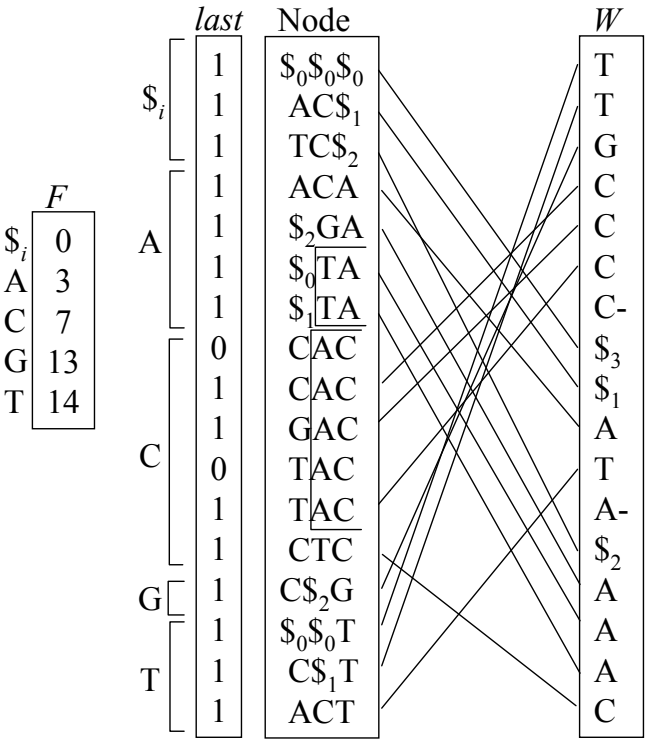

Fig. 2. The succinct representation of the de Bruijn graph in Figure 1. Lines between *Node* and W show *fwd* and *bwd* functions.

The array F is represented in $\mathcal{O}(\sigma \log m)$ bits. If F does not change, we can store it as it is using a simple array and $F[c]$ is computed in constant time. In a dynamic case that a new node or edge is inserted to the de Bruijn graph, we have to update F accordingly. By using a balanced binary tree, F can be maintained in $\mathcal{O}(\log \sigma)$ time.

We also use the inverse of F, that is, given i, we need to know the last character c of $Node[i]$. In a static case, this can be computed in constant time using a *rank*/*select* data structure of $\mathcal{O}(\sigma \log m)+\mathcal{O}(m \log\log m/\log m)$ bits [20]. In a dynamic case, it is done in $\mathcal{O}(\log \sigma)$ time using a balanced binary search tree. It can be improved to $\mathcal{O}(\frac{\log m}{\log\log m}(1+\frac{\log \sigma}{\log\log m}))$ time using [16]. This data structure uses $\mathcal{O}(\sigma \log n)+\mathcal{O}(m \log\log m/\log m)$ bits. Let t_f denote the largest time complexity of those operations.

A character $W[i]$ is from either $\mathcal{A}$ or $\mathcal{A}^-$. If $W[i]$ is from $\mathcal{A}^-$, it means that there exists $j < i$ such that $W[j] = u(W[i])$ and $Node[j]$ and $Node[i]$ have the identical suffix of length $k-1$.

We can define a one-to-one mapping between indices i of *last* with $last[i] = 1$ and indices j of W with $W[j] \in \mathcal{A}$. As stated above, the indices i with $last[i] = 1$ have one-to-one correspondence with the nodes of the de Bruijn graph G.

Consider indices j with $W[j] \in \mathcal{A}$. Let $Node'[j]$ denote the concatenation of the length $k-1$ suffix of $Node[j]$ and $W[j]$. For any $Node'[j]$, there exists i such that $Node[i] = Node'[j]$. Because of the definition of W, there are no indices j and j' $(j \neq j')$ such that $Node'[j] = Node'[j']$. Therefore there is a one-to-one mapping. Furthermore, the mapping is represented by *rank* and *select* queries on W. Let i, j be indices such that $last[i] = 1$ and $Node[i] = Node'[j]$. Let $c = C(i)$ be the last character of $Node[i]$ and $r = rank_1(last, i) - rank_1(last, F[c])$. Then it holds $j = select_c(W, r)$. From j, i is computed by $c = W[i]$, $r = rank_c(W, j)$ and $i = select_1(last, rank_1(last, F[c]) + r)$. We define $bwd(i) = j$ and $fwd(j) = i$. The time complexities of $bwd(i)$ and $fwd(j)$ are $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

Our data structure is similar to the XBW data structure [6] in the sense that the *last* array in ours is the same as S_{last} in the XBW. We propose a new encoding scheme for storing labels of a graph.

3.2 The *Outdegree* and *Outgoing* Operations

The $Outdegree(v)$ operation is easy to support. We assume that v is the index of *last* such that $last[v] = 1$ and $Node[v]$ is the label of the node. From the definition of *last*, it is obvious that $Outdegree(v) = v - pred_1(last, v-1)$. The time complexity is $\mathcal{O}(t_b(m, 2))$.

The $Outgoing(v, c)$ operation is done as follows. For any $1 \leq i \leq m$, we define $R(i) = [pred_1(last, i-1) + 1, succ_1(last, i)]$, which is the range of W and *last* that for all $j \in R(i)$, $Node[j]$ are identical. The labels of outgoing edges of node v are stored in $W[j]$ for $j \in R(v)$. Let j be the index such that $u(W[j]) = c$. We can find j by $pred_c(W, v)$ and $pred_{c^-}(W, v)$. Then $x = Outgoing(v, c)$ can be computed by $x = fwd(j)$.

The time complexity for $Outgoing(v, c)$ is $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

3.3 The *Indegree* and *Incoming* Operations

Consider to compute $Indegree(v)$. Let $d = C(v)$ and $x = bwd(v)$. Then it holds $d = W[x]$ and the first character of $Node[x]$ is the label of an edge pointing to v. Let $y = succ_d(W, x)$. Then all d^- between $W[x]$ and $W[y]$ correspond to parents of v. The number of such d^- is computed by *rank* on W. The time complexity is $\mathcal{O}(t_f + t_b(m, 2\sigma))$.

To compute $Incoming(v, c)$, we need to obtain the first character of $Node[i]$ such that $x \leq i < y$ and $u(W[i]) = d$. The first character of $Node[i]$ is computed by $C(b^{k-1}(i))$ where b^{k-1} stands for applying $bwd(succ_1(last, i))$ repeatedly $k-1$ times. We perform a binary search to find the index i such that $c = C(bwd^{k-1}(i))$. The time complexity is $\mathcal{O}(k(t_f + t_b(m, 2\sigma)) \log \sigma)$.

3.4 The *Index* operation

Recall that $Index(s)$ returns the index i of the node whose label is the string s of length k. Precisely, it returns i such that $last[i] = 1$ and $Node[i] = s$.

The algorithm for $Index(s)$ is similar to [7]. Let $i_1 < i_2 < \cdots < i_w$ be the indices such that $last[i_j] = 1$ and $Node[i_j]$ and s have the same suffix of length d ($1 \leq d \leq k$). Let i_0 be the smallest index in $R(i_1)$. Then for any i such that $i_0 \leq i \leq i_w$, $Node[i]$ and s have the same suffix of length d and for other indices this does not hold. Therefore $Index(s)$ can be done by computing ranges $[i_0, i_w]$ for $d = 1, 2, \ldots, k$. Let c_d denote the d-th character of s ($1 \leq d \leq k$). For $d = 1$, the range is $[F[c_1] + 1, F[c_1 + 1]]$. Given the range $[\ell_d, r_d]$ for d, we can compute the range $[\ell_{d+1}, r_{d+1}]$ for $d+1$ as follows. The end of the range r_{d+1} is computed by $r_{d+1} = Outgoing(r_d, c_{d+1})$. The beginning of the range ℓ_d is computed by $pred_1(last, Outgoing(succ_1(last, \ell_d), c_{d+1}) + 1$.

The above algorithm can be simplified. Instead of computing ranges $[i_0, i_w]$, we can use $[i_1, i_w]$. For $d = 1$, the range is $[succ_1(last, F[c_1]+1), F[c_1+1]]$. Given the range $[\ell_d, r_d]$ for d, the range for $d+1$ is obtained by $r_{d+1} = Outgoing(r_d, c_{d+1})$ and $\ell_{d+1} = Outgoing(\ell_d, c_{d+1})$. The time complexity is $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$.

3.5 Time and Space Complexities

We implement the above data structure for the static case using known succinct data structures. The array F is stored in $\sigma \log m$ bits. The data structure for computing $C(i)$ uses $\mathcal{O}(\sigma \log m) + \mathcal{O}(m \log\log m / \log m)$ bits. The operation time t_f is constant. The string $last$ is stored in $m + o(m)$ bits so that $rank$, $select$, and $access$ takes constant time [20]. The string W is stored by using [8]. Because the characters of W are from $\mathcal{A} \cup \mathcal{A}^- \cup \{\$_1, \ldots, \$_M\}$, the alphabet size is $2\sigma + M$. We can reduce the alphabet size to $2\sigma + 1$ by unifying the M terminators $\$_1, \ldots, \$_M$ into a character \$. We distinguish two terminators, but encode them using the same code.

The string W is stored in $m \log(2\sigma + M) + \mathcal{O}((\sigma + M) \log m) + o(m \log \sigma) = m + m \log \sigma + \mathcal{O}((\sigma + M) \log m) + o(m \log \sigma)$ bits, and the time complexities t_r, t_s, t_a are $\mathcal{O}(\frac{\log \sigma}{\log\log m})$. Therefore the time complexities for $Outdegree$, $Indegree$, $Outgoing$, $Incoming$, and $Index$ are $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{\log \sigma}{\log\log n})$, $\mathcal{O}(\frac{k \log^2 \sigma}{\log\log n})$, $\mathcal{O}(\frac{k \log \sigma}{\log\log n})$, respectively.

For polylog-size alphabets, $Outdegree$, $Indegree$ and $Outgoing$ takes constant time, $Incoming$ takes $\mathcal{O}(k \log \sigma)$ time, and $Index$ takes $\mathcal{O}(k)$ time.

4 On-Line Construction

In this section we propose an on-line construction algorithm of the de Bruijn graph of a string. Here on-line means given the succinct de Bruijn graph G of a string $T = T[1] \cdots T[N]$, we change it to the succinct de Bruijn graph G' of the string $T' = T[1] \cdots T[N+1]$ which is made by appending a character to T.

As stated above, our succinct representation of G assumes that a character \$ is appended to the end of T. Let p be the position of \$ in W. To construct the succinct representation of G', we first change $W[p]$ from \$ to $T[N+1]$ and modify other parts if necessary, then insert \$ to another position of W. The details are as follows.

Let p be the position of \$ in W for the string $T = T[1] \cdots T[N]$. If a new character $c = T[N+1]$ is appended to the end of T, we change $W[p]$ from \$ to $T[N+1]$. We have to maintain the invariant that for all $i \in R(p)$, that is, $Node[i] = Node[p]$, $W[i]$ are distinct. Because before changing $W[p]$ they are distinct, we can check the invariant by finding the character $c = T[N+1]$ or c^- in $W[i]$ such that $i \in R(p)$. This is done by *rank* and *select* on W.

If $T[N+1]$ already exists in the range, let p' be its position. We delete $W[p]$ and $last[p]$ and we insert \$ in W at position $x = fwd(p')$. We also insert 0 in $last[x]$ because $Node[x]$ already exists. We update $p = x$ and the array F accordingly.

If $T[N+1]$ does not exist in the range, we change $W[p] =$ \$ to either $c = T[N+1]$ or c^-. To determine c or c^-, we first find the nearest occurrence of c to $W[p]$, namely, its position is $j = pred_c(W, p-1)$ if it exists ($j > 0$). We compare $Node[j]$ with $Node[p]$. If they have the same suffix of length $k-1$, we change $W[p]$ to c^-, and otherwise change $W[p]$ to c. We compare characters of $Node[j]$ and $Node[p]$ one by one using the *bwd* function. We also compare $Node[j_2]$ with $Node[p]$ where $j_2 = succ_c(W, p+1)$ if it exists ($j_2 \leq m$). If they share the length $k-1$ suffix, we change $c_2 = W[j_2]$ to c_2^-. This takes $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$ time. If the nearest c does not exist ($j = 0$), let $j = F[c]$. The position x to insert \$ is computed by $x = fwd(j)$. We insert 0 to $last[x]$ if $W[p]$ or $W[j_2]$ has a character in $\mathcal{A}^-$, or 1 otherwise. Finally we set $p = x$ and update the array F.

In total, the update operation takes $\mathcal{O}(k(t_f + t_b(m, 2\sigma)))$ time. If we use the dynamic *rank*/*select* data structure of [16] for W and *last*, $t_b = \mathcal{O}(\frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m}))$ time. We also use [16] for computing $C(i)$. Then $t_f = \mathcal{O}(\frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m}))$ and the space is $\mathcal{O}(\sigma \log n) + \mathcal{O}(m \log\log m / \log m)$ bits. Because we repeat this update operation N times for all characters of the input string, the succinct de Bruijn graph can be constructed in $\mathcal{O}\left(Nk \cdot \frac{\log m}{\log\log m}(1 + \frac{\log\sigma}{\log\log m})\right)$ time. For polylog-sized alphabets, it becomes $\mathcal{O}(Nk \cdot \frac{\log m}{\log\log m})$.

It is easy to construct the static data structure from the dynamic one. The strings *last* and W for the static one are generated by applying *access* operations to the dynamic one for $i = 1, \ldots, m$ in $\mathcal{O}(mt_b(m, 2\sigma))$ time. After constructing the static strings, the auxiliary data structures for computing *rank*/*select* are constructed in $\mathcal{O}(m)$ time.

5 Concluding Remarks

We have proposed a succinct representation of de Bruijn graphs, which can be constructed with efficient time and space complexities, and in an on-line manner. Therefore they are useful for large-scale genome assembly.

The succinct de Bruijn graph can be also used for data compression. The PPM (Prediction by Partial Matching) is a text compression algorithm [4]. In the order-k PPM, a character is compressed using statistical information that it appears after a string of length k based on a given probability distribution. We can easily extend our succinct de Bruijn graph to be used for PPM compression. In addition to the array W, we use another array to store the numbers of times

that each edge is traversed. Then we have enough information for compression. The succinct de Bruijn graph is used for natural language processing because it stores all n-grams in a text.

Our future work will be to improve the time complexity for the on-line construction algorithm, and to implement the proposed data structure and apply it for assembling large genomes and PPM data compression. A sample source code is available at `http://code.google.com/p/csalib/`.

Acknowledgments. KS and TS are supported in part by KAKENHI 23240002.

References

1. Batzoglou, S., Jaffe, D.B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J.P., Lander, E.S.: Arachne: a whole-genome shotgun assembler. Genome Research 12, 177–189 (2002)
2. De Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
3. Burrows, M., Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report (1994)
4. Cleary, J.G., Witten, I.H.: Data Compression Using Adaptive Coding and Partial String Matching. IEEE Trans. on Commun. COM-32(4), 396–402 (1984)
5. Conway, T.C., Bromage, A.J.: Succinct data structures for assembling large genomes. Bioinformatics 27(4), 479–486 (2011)
6. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM 57(1), 4:1–4:33 (2009)
7. Ferragina, P., Manzini, G.: Indexing compressed texts. Journal of the ACM 52(4), 552–581 (2005)
8. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed Representations of Sequences and Full-Text Indexes. ACM Transactions on Algorithms 3(2(20)) (2006)
9. Huang, X., Yang, S.P.: Generating a genome assembly with pcap. Current Protocols in Bioinformatics Unit 11.3 (2005)
10. Kasahara, M., Morishita, S.: Large-Scale Genome Sequence Processing. Imperial College Press (2006)
11. Li, R., Zhu, H., Ruan, J., Qjan, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K.: H Yang, and J. Wang. De novo assembly of human genomes with massively parallel short read sequencing. Genome Research 20, 265–272 (2009)
12. MacCallum, I., Przybylski, D., Gnerre, S., Burton, J., Shlyakhter, I., Gnirke, A., Malek, J., McKernan, K., Ranade, S., Shea, T.P., Williams, L., Young, S., Nusbaum, C., Jaffe, D.B.: Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. Genome Biology 10(R103) (2009)
13. Miller, J.R., Koren, S., Sutton, G.: Assembly algorithms for next-generation sequencing data. Genomics 95, 315–327 (2010)
14. Myers, E.W.: Toward simplifying and accurately formulating fragment assembly. Journal of Comutational Biology 2, 275–290 (1995)

15. Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H.J., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D., Venter, J.C.: A whole-genome assembly of drosophila. Science 287, 2196–2204 (2000)
16. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. Submitted for Journal Publication (2010). A preliminary version appeared In: Proc. ACM-SIAM SODA, pp. 134–149 (2010), http://arxiv.org/abs/0905.0768
17. Okanohara, D., Sadakane, K.: Practical Entropy-Compressed Rank/ Select Dictionary. In: Proc. of Workshop on Algorithm Engineering and Experiments, ALENEX (2007)
18. Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to dna fragment assembly. Proceedings of the National Academy of Sciences 98, 9748–9753 (2001)
19. Pop, M.: Genome assembly reborn: recent computational challenges. Briefings in Bioinformatics 10(4), 354–366 (2009)
20. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms 3(4) (November 2007)
21. Sahli, M., Shibuya, T.: Arapan-s: a fast and highly accurate whole-genome assembly software for viruses and small genomes. BMC Research Notes (in press)
22. Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J.: Abyss: a parallel assembler for short read sequence data. Genome Research 19, 1117–1123 (2009)
23. Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., Yu, D.W.: Exploiting sparseness in de novo genome assembly. BMC Bioinformatics 13(suppl. 6:S1) (2012)
24. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Research 18, 821–829 (2008)