BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 4 2011, pages 479–486
doi:10.1093/bioinformatics/btq697

Sequence analysis

Advance Access publication January 17, 2011

Succinct data structures for assembling large genomes

Thomas C. Conway* and Andrew J. Bromage

NICTA Victoria Research Laboratory, Department of Computer Science and Engineering, The University of Melbourne, Parkville, Australia

Associate Editor: Martin Bishop

ABSTRACT

Motivation: Second-generation sequencing technology makes it feasible for many researches to obtain enough sequence reads to attempt the *de novo* assembly of higher eukaryotes (including mammals). *De novo* assembly not only provides a tool for understanding wide scale biological variation, but within human biomedicine, it offers a direct way of observing both large-scale structural variation and fine-scale sequence variation. Unfortunately, improvements in the computational feasibility for *de novo* assembly have not matched the improvements in the gathering of sequence data. This is for two reasons: the inherent computational complexity of the problem and the in-practice memory requirements of tools.

Results: In this article, we use entropy compressed or *succinct* data structures to create a practical representation of the de Bruijn assembly graph, which requires at least a factor of 10 less storage than the kinds of structures used by deployed methods. Moreover, because our representation is entropy compressed, in the presence of sequencing errors it has better scaling behaviour asymptotically than conventional approaches. We present results of a proof-of-concept assembly of a human genome performed on a modest commodity server.

Availability: Binaries of programs for constructing and traversing the de Bruijn assembly graph are available from http://www.genomics.csse.unimelb.edu.au/succinctAssembly.

Contact: tom.conway@nicta.com.au

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 12, 2010; revised on December 13, 2010; accepted on December 14, 2010

1 INTRODUCTION

A central problem in sequence bioinformatics is that of assembling genomes from a collection of overlapping short fragments thereof. These fragments are usually the result of sequencing—the determination by an instrument of a sampling of subsequences present in a sample of DNA. The number, length and accuracy of these sequences varies significantly between the specific technologies, as does the degree of deviation from uniform sampling, and all these are constantly changing as new technologies are developed and refined (Fox *et al.*, 2009). Nonetheless, it is typically the case that we have anywhere from hundreds of thousands of sequences several hundred bases in length to hundreds of millions of sequences a few tens of bases in length with error rates between 0.1% and 10%, depending on the technology.

*To whom correspondence should be addressed.

The two main techniques used for reconstructing the underlying sequence from the short fragments are based on overlap-layout consensus models and de Bruijn graph models. The former was principally used with older sequencing technologies that tend to yield fewer longer reads, and the latter has become increasingly popular with second-generation sequencing technologies, which yield many more shorter sequence fragments. Irrespective of the technique, it has been shown [e.g. by Medvedev *et al.* (2007)] that the problem of sequence assembly is computationally hard, and as the correct solution is not rigorously defined, all practical assembly techniques are necessarily heuristic in nature. It is not our purpose here to discuss the various assembly techniques—we restrict our attention to certain aspects of de Bruijn graph assembly—we refer the reader to Miller *et al.* (2010) for a fairly comprehensive review of assemblers and assembly techniques.

Space consumption is a pressing practical problem for assembly with de Bruijn graph-based algorithms and we present a representation for the de Bruijn assembly graph that is extremely compact. The representations we present use entropy compressed or *succinct* data structures. These are representations, typically of sets or sequences of integers that use an amount of space bounded closely by the theoretical minimum suggested by the zero-order entropy of the set or sequence. These representations combine their space efficiency with efficient access. In some cases, query operations can be performed in constant time, and in most cases they are at worst logarithmic.

Succinct data structures are a basic building block; Jacobson (1989) shows more complex discrete data structures such as trees and graphs that can be built using them. Some of the tasks for which they have used include Web graphs (Claude and Navarro, 2007), XPath indexing (Arroyuelo *et al.*, 2010), partial sums (Hon *et al.*, 2003) and short read alignment (Kimura *et al.*, 2009).

2 BACKGROUND

Let Σ be an alphabet, and $|\Sigma|$ be the number of symbols in that alphabet. In the case of genome assembly, the alphabet Σ is $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$. The length of a string s of symbols drawn from Σ is written $|s|$. The notation $s[i,j)$ is used for the substring of s starting at position i (counting from 0) to, but not including j.

The directed de Bruijn graph of degree k is defined as

$$G_* = \langle V_*, E_* \rangle$$
$$V_* = \left\{ s : s \in \Sigma^k \right\}$$
$$E_* = \left\{ \langle n_f, n_t \rangle : n_f, n_t \in V_*; n_f[1,k) = n_t[0,k-1) \right\}$$

That is, the nodes of the de Bruijn graph V_* correspond to all the k length strings over Σ and an edge exists between each pair of nodes for which the last $k-1$ symbols of the first are the same as the first $k-1$ of the second.

The k length string labelling a node is usually referred to as a k-gram in the computer science literature and a k-mer in the bioinformatics literature. The labels of the edges, as noted in Good (1946), are $k+1$mers. For clarity, we use $\rho=k+1$, and refer to edges as ρ-mers.

We note that among the special properties of the de Bruijn graph is the fact that a given node can have at most $|\Sigma|$ successor nodes: formed by taking the last k bases of the node and extending them with each of the symbols in the alphabet. That is, we can define the successors of a node n:

$$succ_*(n) = \{n[1,k)\cdot b : b \in \Sigma\} \quad (1)$$

$$pred_*(n) = \{b\cdot n[0,k-1) : b \in \Sigma\} \quad (2)$$

To use the de Bruijn graph for assembly, we can build a subset of the graph by finding the nodes and edges that are supported by the information in the sequence reads. The edges are also annotated with a count of the number of times that a ρ-mer is observed in the sequence data. The counts are used for two purposes. The first is to distinguish edges that arise from sequencing errors (which will have very low counts) from those that arise from the underlying genome (which will have higher counts). The second is to estimate the number of copies of that edge in the underlying genome.

Given a set of reads S, we can define a *de Bruijn assembly graph*, defining the nodes V_S in terms of the edges E_S rather than the other way round, as we did above. To define the nodes, we create two (overlapping) sets: the set of nodes F_S *from* which an edge proceeds, and the set of nodes T_S *to* which an edge proceeds.

$$E_S = \{s_i[j,j+\rho) : 0 \le j < |s_i| - k; \forall s_i \in S\} \quad (3)$$

$$F_S = \{e[1,\rho+1) : e \in E_S\}$$

$$T_S = \{e[0,\rho) : e \in E_S\}$$

$$V_S = F_S \cup T_S \quad (4)$$

$$G_S = \langle V_S, E_S \rangle \quad (5)$$

From the DNA alphabet and Equation (1), a given node in the assembly graph can have at most four successor nodes, and by Equation (2), a given node can also have at most four predecessor nodes.

2.1 Reverse complements

An important distinction between ideal strings and the DNA sequences that are used in genome assembly is that the latter can be read in two directions: forwards and in the reverse direction with the individual DNA letters exchanged with their Watson–Crick complements (`A` ↔ `T` and `C` ↔ `G`). In most sequencing scenarios, fragments of DNA are randomly sequenced in either direction, something that must be taken into account during assembly. First, sequence reads are processed twice—once reading them forwards, and then reading them in the reverse complement direction. Then, in most cases, nodes corresponding to reverse complement sequences are merged, and the edges are made bi-directed to match up the sequences correctly [see, for example Medvedev *et al.* (2007)]. For our current discussion, we will not combine them, but will store them separately. This makes the graph symmetric; a forward traversal corresponds to a backwards traversal on the reverse complement path, and vice versa.

Figure 1 shows a de Bruijn assembly graph for a short string.

2.2 From de Bruijn assembly graphs to genomes

The de Bruijn graph is both Eulerian and Hamiltonian, a property that Idury and Waterman (1995) showed was useful for genome assembly. In principle, at least, the assembled sequence corresponds to an Eulerian tour of the de Bruijn assembly graph. The details of how this may be done in practice are beyond the scope of our current discussion, but the approaches include those described in Jackson *et al.* (2009); Pevzner *et al.* (2001); Simpson *et al.* (2009); Zerbino and Birney (2008). Our current discussion is focussed on how we might represent the de Bruijn assembly graph in a practical program for performing large genome assembly.

A simple approach to representing the de Bruijn assembly graph is to represent the nodes as ordinary records (i.e. using a `struct` in C or C++), and the edges as pointers between them. If we assume a node contains the k length substring (or k-mer) represented as a 64 bit integer (assuming $k \le 32$), 32 bit edge counts and pointers to four possible successor nodes, and there are no memory allocator overheads, then the graph will require 56 bytes per node. In the *Drosophila melanogaster* genome, with $k=25$, there are about 245 million nodes (including reverse complements), so we would expect the graph to take nearly 13 GB. For the human genome with $k=25$, there are about 4.8 billion nodes (again, including reverse complements), so the graph would require over 250 GB. These data structures are large, but more is needed, because there is no way in what is described to locate a given node, so for instance a simple hash table (generously assuming a load factor of 1) might require an extra 16 bytes (hash value + pointer) per node or over 70 GB for the human genome. These figures are extremely conservative, since they ignore the effect of sequencing errors.

We can get an estimate of the proportion of edges in the graph that are due to errors with a simple analysis. Most sequencing errors give rise to unique k-mers, and hence many edges that occur only once. Ignoring insertion and deletion errors, for a given k (or ρ), a single error can cause up to ρ spurious edges, which, if we assume a random distribution of errors, are overwhelmingly likely to be unique. Thus, the number of spurious edges is proportional to the volume of sequence data, whereas the number of true edges is proportional to the genome size, and will converge on that number as the volume of sequence data increases. For example, consider the case of an organism with a 1 Mbp genome, which we sequence with sequence reads of 100 bp in length. If we assume that on average a read contains 1 error, then with $\rho=26$, we will typically have 74 true edges and 26 spurious edges. Assuming the reads are uniformly distributed, once the number of reads exceeds about 14 000, almost all the 1 million true edges will be present, and there will be about 364 000 spurious edges. Beyond this, as the number of reads increases, the number of true edges will remain the same, but the number of spurious edges will continue to increase linearly. By

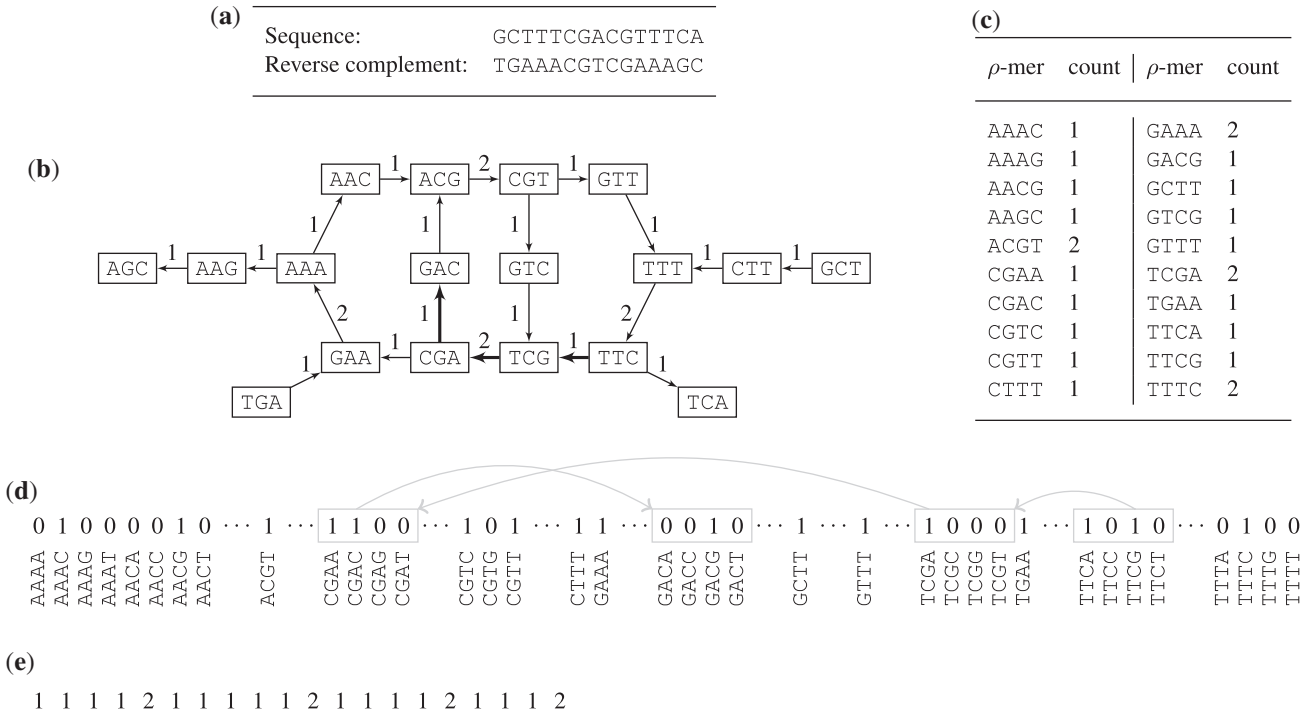

Fig. 1. A de Bruijn assembly graph and its representation. (**a**) Source sequence. (**b**) The corresponding assembly graph. The edges are labelled with counts. The path marked with bold arrows is TTCGAC. (**c**) Extracted ρ-mers with counts. (**d**) The sparse bitmap representation. The gray boxes exemplify groups of edges that proceed from a single node. The arrows show the sequence TTCGAC. (**e**) Dense array of counts.

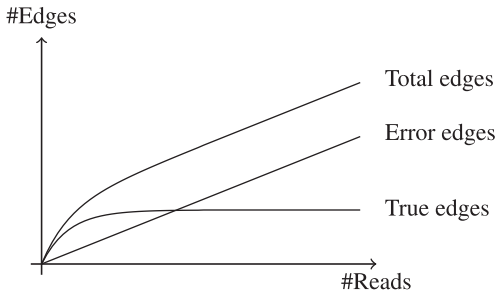

Fig. 2. A sketch showing the relationship between the number of sequence reads and the number of edges in the graph. Because the underlying genome is fixed in size, as the number of sequence reads increases the number of edges in the graph due to the underlying genome that will plateau when every part of the genome is covered. Conversely, since errors tend to be random and more or less unique, their number scales linearly with the number of sequence reads. Once enough sequence reads are present to have enough coverage to clearly distinguish true edges (which come from the underlying genome), they will usually be outnumbered by spurious edges (which arise from errors) by a substantial factor.

the time the *coverage* (the expected count on all the true edges) reaches 40 (which we have observed in several data sets), we would expect to see about 14 million spurious edges. That is, the spurious edges would outnumber the true edges by a factor of 14.

Figure 2 illustrates this problem.

Much of this space is devoted to storing pointers, so the question naturally arises: are these pointers necessary, or can they be avoided? Existing assemblers such as *velvet* (Zerbino and Birney, 2008) combine nodes corresponding to forward and reverse complements, and merge nodes on unbranched paths, and although these techniques significantly reduce the amount of memory required, they nonetheless represent an *ad hoc* approach to the problem of reducing the memory required to represent the de Bruijn assembly graph.

ABySS (Simpson *et al.*, 2009) goes further, and avoids pointers in the first place. It represents the graph as a (distributed) hash table, which acts as a mapping from k-mer to a single byte containing the connectivity information: a bit indicating the existence of each of the four forward and four reverse complement edges. ABySS as described in Simpson *et al.* (2009) does not record the multiplicity of the edges. Thus, if we assume we store the k-mers with 2 bits per base, do not merge non-branching paths and we ignore alignment issues, the space usage of the graph is $|E_s|(\frac{k}{4}+1)\frac{1}{\delta}$, where δ is the load factor of the hash table. It is clear that the load factor is crucial to the effectiveness of this method, and although an in-depth discussion of hash table implementation is beyond our current scope, we note that if space usage is to be minimized, open addressing techniques such as Cuckoo Hashing (Fotakis *et al.*, 2003; Pagh and Rodler, 2001) should be used. If we assume a space efficient hashing technique, we might have $\delta = 0.8$, in which case, in the case of the human genome above, with 4.8 billion k-mers, assuming forward edges are combined with reverse compliments, we would require 43.5 GB. We note that the space usage of this graph representation, like the node-and-pointer one, is linear in the number of edges.

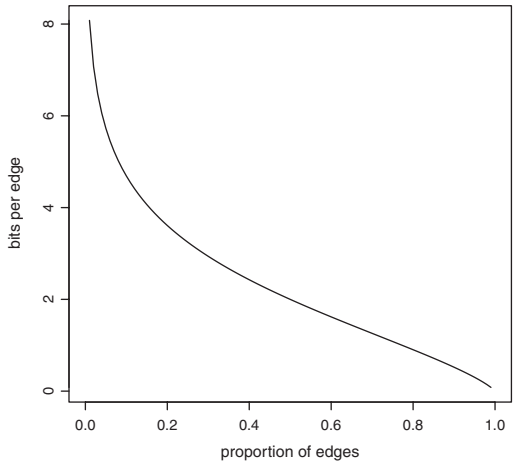

Fig. 3. The number of bits per edge required to represent the de Bruijn graph. The function is over the range $[\frac{1}{4^\rho}, 1]$, and although the vertical scale shown stops at 8 bits per edge, in practice, the number of bits per edge at the left hand limit of $\frac{1}{4^\rho}$ is 2ρ.

3 APPROACH

Our approach to memory-efficient representation of an assembly graph begins by re-framing the question of whether the pointers in a naive graph representation are necessary. Rather we ask what information is necessary, and what is redundant or ephemeral. How many bits are required to represent the de Bruijn assembly graph from an information-theoretic point of view?

The de Bruijn assembly graph is a subset of the de Bruijn graph. Of the $|\Sigma|^\rho$ edges in the de Bruijn graph, the assembly graph contains $|E_S|$. The self-information of a set of edges that make up an assembly graph, and hence the minimum number of bits required to encode the graph, is

$$\#\text{bits} = \log\binom{4^\rho}{|E_S|} \tag{6}$$

(Note, that unless otherwise specified, all logarithms are base 2.)

To appreciate the implications of this lower bound, it is useful to consider not just the total number of bits, but the number of bits per edge. In conventional approaches, the number of bits per edge is approximately constant—doubling the number of edges doubles the space required. In contrast, if we divide the total number of bits required according to Equation (6) by the number of edges, we see that the number of bits per edge monotonically decreases as the number of edges increases. This function, expressed in terms of the proportion of edges present, is shown in Figure 3. There is an empirical validation of this presented in Section 4 of the Supplementary Materials.

For the de Bruijn assembly graph with $k=25$, the human genome (build 37) yields 4 796 397 453 distinct edges, including reverse complements. By the equation above, taking S to be the genome itself:

$$\#\text{bits} = \log\binom{4^{26}}{4,796,397,453} \approx 12\text{GB}$$

We do not need to store the nodes explicitly, since they are readily inferred from the edges:

$$\text{from-node}(e) = e[0, \rho-1)$$
$$\text{to-node}(e) = e[1, \rho)$$

Equation (6) gives a lower bound on the number of bits required to represent the de Bruijn assembly graph. We would like to find a concrete representation that comes close to that theoretical minimum while allowing efficient random access. The notion that the assembly graph is a subset of the de Bruijn graph immediately suggests that we could create a bitmap with a bit for each edge in the de Bruijn graph, and set the bits for the edges that occur in the assembly graph. Such a scheme depends on being able to enumerate the ρ-mers (i.e. the edges). This is done trivially by numbering the bases (we use A=0, C=1, G=2 and T=3), and interpreting the ρ symbols as an integer of length 2ρ bits. Conceptually, then, we can create a bitmap with 4^ρ bits, and place **1**s in the positions corresponding to the edges in the assembly graph. In this representation, the k-mers are represented implicitly. This is an important point since all other assembly approaches that we are aware of have to store the k-mers explicitly.

For example, in Figure 1d, the node labelled with the k-mer TTC has two outgoing edges labelled with the ρ-mers TTCA and TTCG. No explicit representation of the k-mer is required because it is implicit from the existence of the two ρ-mers. Note that nodes with incoming but no outgoing edges cannot be interrogated directly, but because the graph is symmetric over reverse complementation, the reverse complement nodes will have outgoing edges.

Given such a bitmap, we can determine the successor set of a given node from the definition of the de Bruijn assembly graph, by probing the positions corresponding to the four edges that could proceed from the node. For a node corresponding to a k-mer n, the four positions in the bitmap are $4n$, $4n+1$, $4n+2$ and $4n+3$.

There is a particular formalism, first proposed by Jacobson (1989) for querying sets of integers represented as bitmaps that is useful in this setting. Given a bitmap $\mathbf{b}$ with the positions of the set members set to **1** and the rest of the positions set to **0**, the formalism uses two query operators *rank* and *select* with the following definitions[1]:

$$\text{rank}_{\mathbf{b}}(p) = \sum_{0 \le i < p} b_i$$
$$\text{select}_{\mathbf{b}}(i) = \max\{p < n | \text{rank}_{\mathbf{b}}(p) \le i\}$$

Intuitively, $rank_{\mathbf{b}}(p)$ is the number of ones in the bitmap $\mathbf{b}$ to the left of position p, and $select_{\mathbf{b}}(i)$ is the position of the i-th set bit, where the set bits are numbered starting from zero.

Using the *rank*/*select* formalism, we can compute the set of the successor edges for a node n efficiently given a bitmap representing the set of edges:

$$\text{succ}_{E_S}(n) = \left\{ select_{E_S}(r) \mid r \in \left[\text{rank}_{E_S}(4n), \text{rank}_{E_S}(4n+4)\right) \right\}$$

This forms the basis of a method for efficient traversal of a de Bruijn assembly graph represented as a set of integers or, equivalently, a bitmap.

[1]The literature contains several slightly different definitions that arise from different conventions for subscripting arrays: mathematical literature tends subscript from one; computer science literature from zero. We use the latter.

Next we consider how the edge counts should be represented. For this we draw on the *rank*/*select* formalism again, and note that while the edges are sparse (a point that we will come back to shortly), the *ranks* of the edges are dense, filling the range $[0, |E_S|)$. Therefore, we can store the edge counts in a vector of 32 bit integers.

4 METHODS

The preceding discussion presented a technique for representing a de Bruijn assembly graph as a bitmap using 4^ρ bits. For a typical value of $\rho = 26$ (i.e. $k = 25$), the bitmap would require 512TB. This is clearly infeasible (and larger k would be worse), but the bitmap is extremely sparse. Of the 4.5×10^{15} bits, for the human genome, only 4.8×10^9 are **1**. That is, the fraction of the bits that are set is 10^{-8}, so a representation which exploits the sparsity should be used. Equation 6 gives a precise lower bound on the number of bits that any lossless representation requires, and there has been a large amount of research in the last two decades on the efficient representation of data structures that are close to this theoretical limit.

Let $\mathcal{B}_{\nu,\mu}$ be the set of bitmaps with ν bits, where exactly μ bits are set. Jacobson (1989) defines a *succinct representation* as a way of mapping the elements of $\mathcal{B}_{\nu,\mu}$ into a read-only memory such that the amount of space used to represent a bitmap is close to $(1 + o(1))\log|\mathcal{B}_{\nu,\mu}|$ bits. A *succinct data structure* is a succinct representation that also supports desired query operations efficiently. 'Efficiently' can mean either low asymptotic complexity or practical speed on real hardware. In our case, the query operations that we wish to support are *rank* and *select*.

Although Jacobson (1989) defines succinct data structures as read-only objects, Mäkinen and Navarro (2008), among others, show how *insert* and *delete* can be implemented without sacrificing the succinct nature of the representation, for a suitable definition of 'succinct' which takes into account the dynamic nature of the data structure. We do not consider dynamic succinct data structures in this work because existing proposals tend to be quite complex and subtle to implement, and while they tend to have reasonable time complexity in an asymptotic sense (though they are usually not as fast as static data structures), they often exhibit prohibitively high constant factors.

A summary [abstracted from Okanohara and Sadakane (2006)] of the data structures that we use in our implementation is shown in Table 1. Note that ν the number of possible edges is 4^ρ, and therefore increasing ρ (or by implication k) will increase the space required, even if the number of extant edges does not change.

The **darray** and **sarray** data structures (Okanohara and Sadakane, 2006) are optimized for the case when the bitmap is 'dense' or 'sparse', respectively. If $\mu/\nu \approx 1/2$, $\log\binom{\nu}{\mu} \approx \nu$, so storing the uncompressed bitmap is already succinct; in this case, we call the bitmap 'dense', and implement *rank* and *select* with $o(\nu)$ extra space to speed up those operations. If $\mu/\nu \ll \frac{1}{2}$, then we call the bitmap 'sparse'; in this case, the bitmap can be compressed close to optimal space using Elias–Fano coding (Elias, 1974), which is the basis for **sarray**. A more detailed discussion of these data structures is presented in Section 2 of the Supplementary Materials.

We emphasize that although we have used these particular concrete entropy-bounded sparse bit array structures, our technique (and ultimately our code) is expressed in terms of *rank* and *select*, so it would be a straight forward matter to substitute other concrete representations [e.g. such as the version in Claude and Navarro (2008) of the structure originally due to Raman *et al.* (2007)].

Table 1. Summary of succinct data structures

Method	Size (bits)	Rank complexity	Select complexity
darray	$\nu + o(\nu)$	$O(1)$	$O(\log^4 \mu/\log \nu)$
sarray	$\mu \log \frac{\nu}{\mu} + 1.92\mu + o(\mu)$	$O(\log \frac{\nu}{\mu}) + O(\log^4 \mu/\log \nu)$	$O(\log^4 \mu/\log \nu)$

Returning to the representation of the edge counts, in Section 3, we suggested storing the counts in a vector of 32 bit integers indexed by edge rank. This actually uses much more memory than necessary. As previously noted, prior to error removal, a vast majority of edges in the graph are spurious and will have a very low edge count. Most of the true edges have modest counts also: edges that are unique in the underlying genome will have a count somewhere around the basic coverage (e.g. 15–50). For most edges, 8 bits of storage is sufficient, and for most of the remainder 16 bits is sufficient. Only a handful of edges, in practice, need more than 16 bits. Therefore, using 32 bits for every edge is very wasteful.

There are many techniques for creating compressed representations of vectors of integers [see Moffat and Turpin (2002)], but in most cases they do not provide efficient random access. Succinct data structures implementing *rank*/*select* yield an effective technique first introduced by Brisaboa *et al.* (2009). We split each count into the three parts alluded to above: the least significant 8 bits, the 'middle' 8 bits and the most significant 16 bits. We store the least significant 8 bits in a dense vector of bytes L. Corresponding to it, we store a succinct bitmap B_L with a **1** marking those entries for which the middle 8 bits or the most significant 16 bits are non-zero. In a dense vector of bytes M (indexed by rank in B_L), we store the middle 8 bits of those entries for which a **1** exists in B_L. Corresponding to M, we store a sparse bitmap B_M with a **1** marking those entries for which the most significant 16 bits are non-zero. Finally, we have a dense vector if 16 bit words H (indexed by rank in B_M) with the most significant bits of those entries marked in B_M. The bitmaps B_L and B_M are represented using **sarray**.

5 RESULTS

We have created a set of programs that construct and manipulate the de Bruijn assembly graph representation we have described. These do not constitute a complete assembler, but represent the kinds of traversal and manipulation of the graph that are required to build an assembler.

The proof-of-concept assembly procedure is as follows, with each step being performed by a separate program that takes an on-disk representation of the data and produces a new on-disk representation of the data:

(1) Extract ρ-mers (forwards and reverse complements) from the sequence reads and sort them into lexographic order. The result of the sort operation is a list from which we can extract ρ-mer/count pairs from which we construct the the sparse array for the graph structure and the succinct representation of the counts. On large datasets, this can be done in parts, and the resulting partial graphs are merged to form the complete graph.

(2) Perform a left-to-right traversal of the list of edges/counts and discard low frequency edges which almost certainly correspond to errors.

(3) Perform iterations of the tip removal algorithm exactly as described in Zerbino and Birney (2008).

(4) Perform depth first traversal to read of non-branching paths within the graph to report as contigs.

The first step demonstrates the feasibility of building the graph representation; the second, that it is possible to do trivial processing efficiently; the third, that graph traversal can be done to produce a modified representation (in this case eliminating paths in the graph that probably correspond to errors); and the fourth that meaningful contigs can be obtained. A more detailed description of these steps including pseudo-code is provided in Section 3 of the Supplementary

Table 2. Graph size in gigabytes, and build time in minutes, for the main phases of the proof-of-concept assembly

	Number of edges	Graph size (GB)	Counts size (GB)	Total size (GB)	Build time
Complete graph	12 292 819 311	40.8 (28.5)	11.5 (8.01)	52.3 (36.5)	2080
After removing low frequency edges	4 799 738 381	15.1 (27.1)	4.5 (8.02)	19.6 (35.1)	46
After removing tips	3 840 690 715	12.2 (27.2)	3.6 (8.03)	15.8 (35.2)	845

The parenthetical numbers are in bits per edge.

Table 3. Assembly statistics comparing the ABySS assembly of the Yoruban individual NA18507, using the Illumina reads reported in Simpson *et al.* (2009) (SRA accession number SRX016231) with a proof-of-concept assembly using the succinct graph representation.

	ABySS		Proof-of-concept	
Cores		168		8
Nodes		21		1
Total RAM (GB)		336		32
Minimum contig size	≥100 bp	≥1 kb	≥100bp	≥1 kb
Number of contigs	4 348 132	549 522	7 693.288	41 292
Median size (bp)	253	1463	165	1146
Mean size (bp)	484	1703	224	1219
Maximum size (bp)	15 911	15 911	22 032	22 032
N50 size (bp)	870	1731	250	1176
Number of contigs >N50	674 953	188 171	1 994 863	17 939
Sum (Gbp)	2.10	0.94	1.72	0.05

The reported time for the ABySS assembly was 15 h, compared with our elapsed time of 50 h. It is not clear from Simpson *et al.* (2009) whether the reported time is aggregate time, or elapsed (wall) time, though the latter seems more likely.

Materials. We believe that this proof-of-concept demonstrates the feasibility of our method, though a complete assembler would need to do significantly more processing on the graph (e.g. bubble removal), should use read-coherence to resolve local ambiguities and should make use of pairing information to resolve repeats.

We have run this proof-of-concept assembly 'pipeline' on the sequence data from a Yoruban individual from Bentley *et al.* (2008), sample number NA18507, with $k=27$. The assembly was performed using a single computer with 8×2 GHz Opteron cores and 32 GB RAM. The size of the graph (edges and counts) at the stages of the pipeline are shown in Table 2. Each step produces a set of files containing the representation of the graph. These files are then brought into memory by the program for the next step using memory-mapped I/O. The complete graph, at the end of the first step, is 52 GB, which is larger than the 32 GB RAM on the machine, but the next step (removing low frequency edges) does a sequential pass over these structures to produce a new, smaller set. So although the process virtual size is considerably larger than main memory, the accesses have extreme locality, so the overall behaviour is efficient.

Simpson *et al.* (2009) report results of assembling the same data with ABySS. In Table 3, we reproduce the results reported there for the assembly not using the pairing information from the reads, along with the results from our proof-of-concept assembly. Importantly, we have included the scope of the computing resources used in both cases. Unsurprisingly, our 'pipeline', lacking bubble elimination and read-coherent disambiguation of branches, mostly produces only short contigs. Curiously, the longest contig at about 22 kb does not match the reference human genome at all, but is an exact match in to the Epstein–Barr virus, which is an artifact of the preparation of the cell line from which the sequence data were obtained. That this is the longest contig is unsurprising, since viral sequences are not diploid like the human genome, and therefore are less prone to bubbles due to heterozygosity; and viral sequences tend to contain far less repetition than the human genome, and will therefore have much less branching in their de Bruijn graph representation.

6 DISCUSSION

We have claimed that the number of bits per edge should be monotonically decreasing with the number of edges. This is clearly not the case in the results in Table 2: the graph containing all the edges present in the sequence data uses more bits per edge. The analysis in Section 3 gives a lower bound for the number of bits required for the graph. For the 12 billion edges in our complete graph, this suggests that about 22 bits per edge (or 30.7 GB in total) are required. From Table 2, we see that for the complete graph 28.5 bits are required. This translates to about 6.5 bits (or 10 GB) of space used beyond the the theoretical minimum. As discussed in Section 4 of the Supplementary Materials, this is an artifact of our implementation, which could be eliminated, but in absolute terms is very minor. To put it in perspective, 28.5 bits per edge is dramatically less than the 64 bits required for a pointer, and even a hashing-based approach would require at least 35 bits per edge.[2] Other entropy-compressed bit vector representations may bring the space usage of the graph closer to the theoretical minimum.

We have presented a practical and efficient representation of the de Bruijn assembly graph, and demonstrated the kind of the operations that an assembler needs to perform but of course there is much more to doing *de novo* assembly with de Bruijn graph methods than we have presented. A combinatoric number of Eulerian paths exist in the de Bruijn assembly graph, among which true paths must be identified [this is the Eulerian *superpath* problem described by Pevzner *et al.* (2001)]. This is usually done in the first instance by using the sequence reads to disambiguate paths. In the second instance, this is done by using paired sequence reads (e.g. *paired-end* and *mate-pair* sequence reads), in a process usually called *scaffolding*. The algorithms described in the literature can either be implemented directly on our representation or, in most cases, adapted.

[2]An ABySS-like hashing approach combines forward and reverse-complement k-mers, so there are half as many keys. However, 2 bits are required to store each base, so the total size of the hash table, both key and value, is $k+8$ bits per edge. This assumes 100% loading of the hash table, and no overhead in the storing of the k-mers.

One important caveat is that our representation depends on the properties of the de Bruijn graph (i.e. the relationship between nodes and the edges that connect them). While edges may be added or removed, the representation cannot be treated as an arbitrary graph; there cannot, for example, be two nodes that represent the same k-mer along different paths. We do not believe this is a significant obstacle to building a complete assembler based on this representation, and our proof-of-concept implementation supports this belief.

As well as building a practical assembler based on the representation we have presented, there are several opportunities for improving the graph construction. At the moment, the run-time is dominated by sorting, which is done sequentially, and with fairly generic sorting code. We speculate that the sequential sorting speed could be doubled with modest effort, and the whole could be parallelized fairly easily.

6.1 A succinct representation of sequence reads

Among the several components required for a practical assembler mentioned above, the use of reads during assembly is worthy of some further examination. A practical assembler will use the sequence reads to help disambiguate conflations in the de Bruijn graph. Here, we present a simple technique that uses succinct data structures to form a compact representation of the sequence reads, given the de Bruijn assembly graph.

The de Bruijn graph already contains most of the information present in the sequence reads. Each sequence read corresponds to a walk in the de Bruijn assembly graph. The information present in the sequence reads that is not present in the graph is as follows: (i) where in the graph the sequence read starts; (ii) where in the graph it ends (or, equivalently, its length); and (iii) at nodes in the graph where there is more than one outgoing edge, the edge which should be followed.

If we sort the sequence reads into lexicographic order (discarding the original order of the reads), we can efficiently store the initial k-mer of each read and, moreover, construct an efficient index that lets us determine which reads begin with a given k-mer. The lengths of the reads can be stored efficiently by creating a sparse bitmap corresponding to the concatenation of all the sequence reads, with a **1** denoting the start of a sequence read. The *rank* and *select* functions give an efficient means of determining the position in the bitmap of the start and end of a given read.

The sequence of choices or the *walk* that the sequence read follows may be encoded very efficiently in the following way. At each node, we can number the extant outgoing edges $[0, 3]$, and assign a rank to the edge taken by a given sequence read. The ranks may be assigned lexicographically, or in order of edge count (highest to lowest). These ranks require two bits, which we can store in a pair of sparse bitmaps—one for the least significant bit and one for the most significant bit. The positions in these bitmaps correspond to the positions in the bitmap marking the initial positions of sequence reads. In practice, a large majority of nodes have only one outgoing edge, so the rank will be **0**, hence the bitmaps will be sparse. Most of the nodes which have more than one outgoing edge have only two, so in the vast majority of cases, the most significant bit of the rank will be zero, making the bitmap for the most significant bit even more sparse than the one for the least significant bit.

If one wished to use this encoding to encode sequence reads other than those represented in the de Bruijn assembly graph, then it is no longer the case that every sequence read corresponds to a walk in the graph. In this case, a 'nearest' walk could be found, and the differences between the sequence read and the walk could be recorded. This could be done using a sparse bitmap to record those positions at which the walk and the sequence read diverge, and a corresponding vector (indexed by rank in the said bitmap) of bases could be used to store the actual base in the sequence read. There is an optimization problem to find the 'nearest' walk, but simple heuristics are likely to be sufficient.

This scheme could be generalized for sequencing technologies where we may wish to explicitly encode gaps in the sequence read, for example *strobe reads* (Ritz *et al.*, 2010), by the use of an auxiliary bitmap marking the locations of the gaps. This would be an interesting line for further research.

7 CONCLUSION

We have presented a memory-efficient representation of the de Bruijn assembly graph using *succinct* data structures which allow us to represent the graph in close to the minimum number of bits. We have demonstrated its effectiveness by performing a proof-of-concept assembly of a human genome on a commodity server; further work will build on this to produce a more complete assembler.

ACKNOWLEDGEMENT

We thank the anonymous reviewers whose helpful comments have improved this article considerably. Thanks are also due to Justin Zobel, Arun S. Konagurthu and Bryan Beresford-Smith for many fruitful discussions during the long gestation of this work, and for their feedback on drafts of this article.

Funding:National ICT Australia (NICTA) is funded by the Australian Government's Department of Communications; Information Technology and the Arts; Australian Research Council through Backing Australia's Ability; ICT Centre of Excellence programs.

Conflict of Interest: none declared.

REFERENCES

Arroyuelo,D. *et al.* (2010) Fast in-memory xpath search over compressed text and tree indexes. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, *Long Beach*, California, USA, pp. 417–428.

Bentley,D.R. *et al.* (2008) Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, **456**, 53–59.

Brisaboa,N.R. *et al.* (2009) Directly addressable variable-length codes. In Karlgren,J. *et al.* (eds) *SPIRE*, Vol. 5721 of *Lecture Notes in Computer Science*, Springer, Saariselkä, Finland, pp. 122–130.

Claude,F. and Navarro,G. (2007) A fast and compact web graph representation. In Ziviani,N. and Baeza-Yates,R.A. (eds) *SPIRE*, Vol. 4726 of *Lecture Notes in Computer Science*, Springer, Santiago, Chile, pp. 118–129.

Claude,F. and Navarro,G. (2008) Practical rank/select queries over arbitrary sequences. In Amir,A. (eds) *SPIRE*, Vol. 5280 of *Lecture Notes in Computer Science*, Springer, Melbourne, Australia, pp. 176–187.

Elias,P. (1974) Efficient storage and retrieval by content and address of static files. *J. ACM*, **21**, 246–260.

Fotakis,D. *et al.* (2003) Space efficient hash tables with worst case constant access time. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Springer, Berlin, Germany, pp. 271–282.

Fox,S. *et al.* (2009) Applications of ultra-high-throughput sequencing. *Methods Mol. Biol.*, **553**, 79–108.

Good,I.J. (1946) Normal recurring decimals. *J. London Math. Soc.*, **s1-21**, 167–169.

Hon,W.-K. *et al.* (2003) Succinct data structures for searchable partial sums. In Ibaraki,T. (eds) *ISAAC*, Vol. 2906 of *Lecture Notes in Computer Science*, Springer, Kyoto, Japan, pp. 505–516.

Idury,R.M. and Waterman,M.S. (1995) A new algorithm for dna sequence assembly. *J. Comput. Biol.*, **2**, 291–306.

Jackson,B.G. *et al.* (2009) Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics*, **10 (Suppl. 1)**, S14.

Jacobson,G. (1989) Space-efficient static trees and graphs. In *SFCS '89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA. IEEE Computer Society, pp. 549–554.

Kimura,K. *et al.* (2009) Computation of rank and select functions on hierarchical binary string and its application to genome mapping problems for short-read dna sequences. *J. Comput. Biol.*, **16**, 1601–1613.

Mäkinen,V. and Navarro,G. (2008) Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, **4**, 1–38.

Medvedev,P. *et al.* (2007) Computability of models for sequence assembly. In Giancarlo,R. and Hannenhalli,S. (eds) *WABI*, Vol. 4645 of *Lecture Notes in Computer Science*, Springer, Philadelphia, PA, USA, pp. 289–301.

Miller,J.R. *et al.* (2010) Assembly algorithms for next-generation sequencing data. *Genomics*, **95**, 315–327.

Moffat,A. and Turpin,A. (2002) *Compression and Coding Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA.

Okanohara,D. and Sadakane,K. (2006) Practical entropy-compressed rank/select dictionary. *CoRR*, **abs/cs/0610001**.

Pagh,R. and Rodler,F.F. (2001) Cuckoo hashing. *Lect. Notes Comput. Sci.*, **2161**, 122–144.

Pevzner,P.A. *et al.* (2001) An eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, **98**, 9748–9753.

Raman,R. *et al.* (2007) Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, **3**, 43.

Ritz,A. *et al.* (2010) Structural variation analysis with strobe reads. *Bioinformatics*, **26**, 1291–1298.

Simpson,J.T. *et al.* (2009) Abyss: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

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