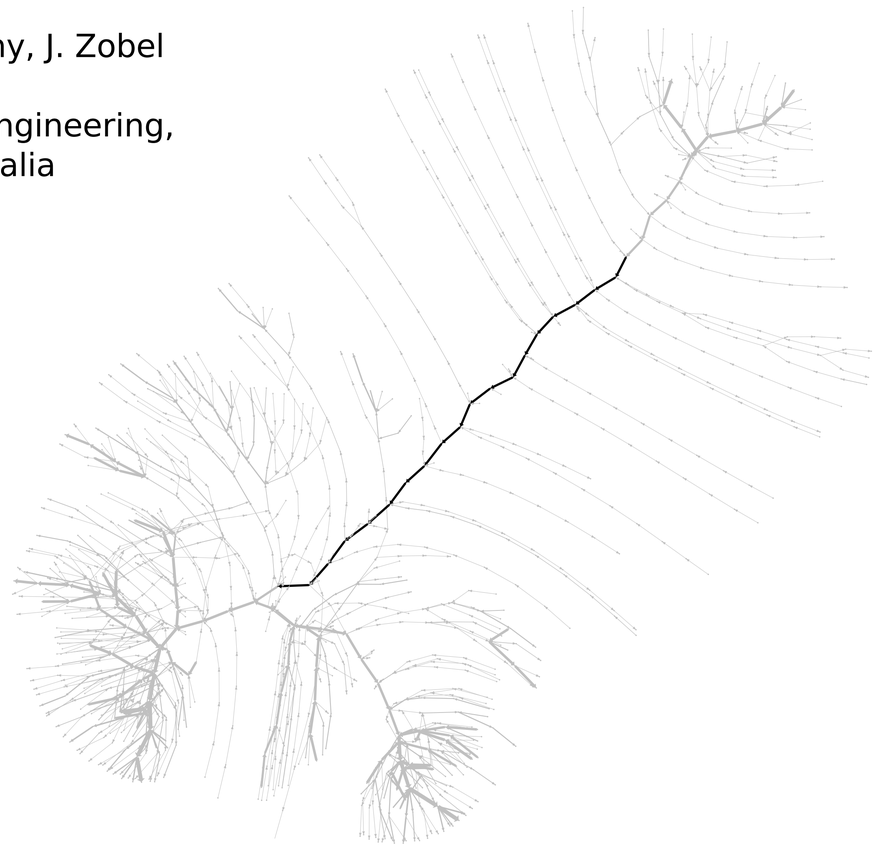
Resource Efficient de novo Assembly of Eukaryotic Genomes

B. Beresford-Smith, A. Bromage, T. Conway, J. Wazny, J. Zobel
NICTA Victoria Research Laboratory
& Department of Computer Science and Software Engineering,
The University of Melbourne, Parkville, Australia

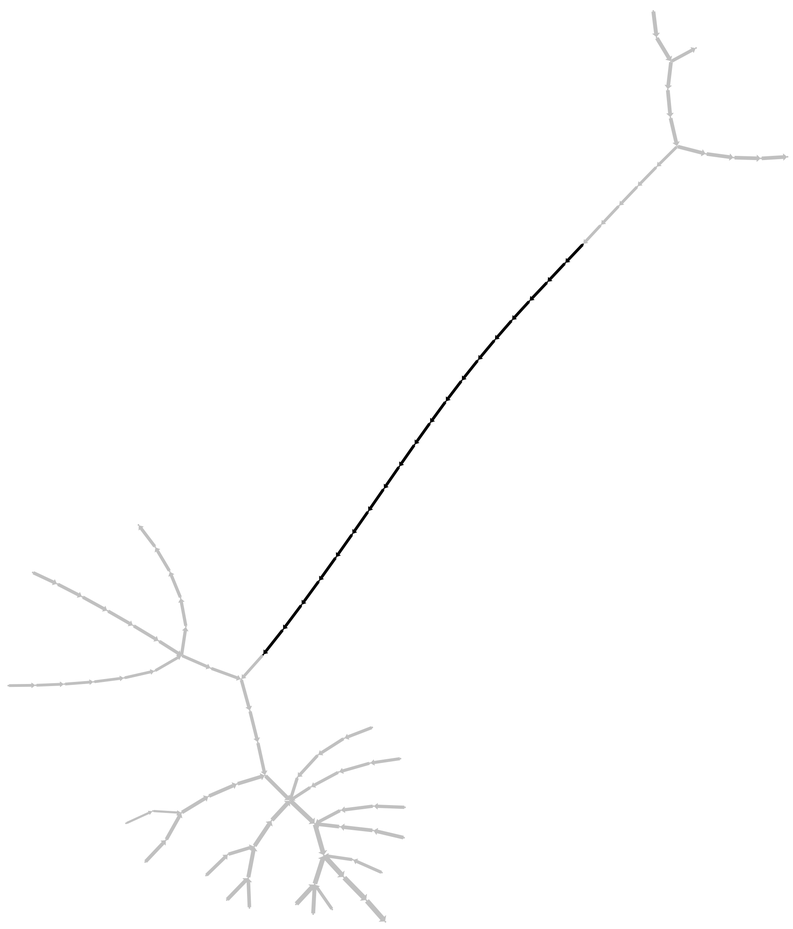


Assembly of genomes from short sequence reads is a core problem in bioinformatics. Assembly algorithms based on de Bruijn graphs have been very successful, particularly with sequence data that is characterised by large numbers (millions or more) of short (30–500 nucleotides) reads. However, a serious practical impediment to the assembly of larger genomes with current de Bruijn graph implementations is the amount of memory required to represent the graph and the sequence reads, with accurate assembly of genomes larger than a few million base-pairs being infeasible on commodity servers.

We have developed and implemented an assembler using a practical and efficient representation of the de Bruijn graph, recently published [3], which exploits the succinct data structures developed by Okanohara & Sadakane [1] and Raman, Raman, & Rao [5]. With our implementation, Gossamer, we have built the de Bruijn graph for a human genome from 1.6x10^9 Illumina 75bp reads using a single server with only 32GB memory. The assembly required 51 hours.

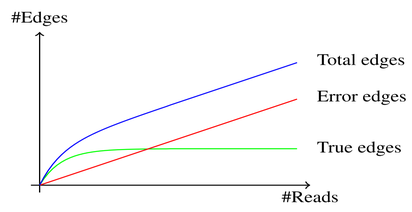
Importantly, our solution requires a sub-linear increase in space requirements as the amount of sequence data (and therefore also sequencing errors) increases. Since the publication [3], we have also developed a technique for threading reads through the assembly graph, requiring space logarithmic in the volume of read data.

We are also making use of the space efficiency of gossamer for the de novo assembly of transcriptomes and metagenomes, where deep sequencing can increase the sensitivity and dynamic range of the analysis.

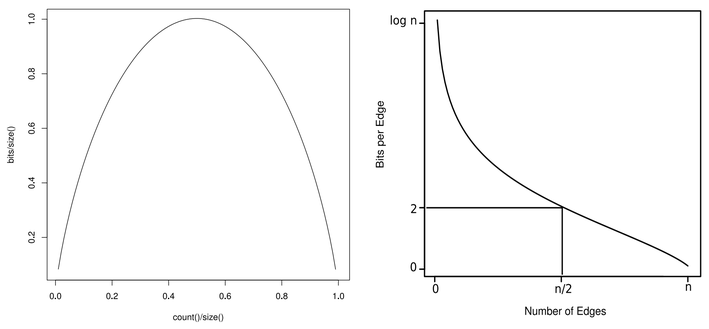


The figures immediately above, and top centre depict a small fragment of the E. Coli genome (highlighted) and the graph around it before (above) and after(below) error removal, with k=27.

The Effect of Errors & Entropy



The figure above is a cartoon showing the relationship between the number of reads and the number of edges in the assembly graph. When the number of reads is small, adding reads adds true edges not previously seen, as well as error edges. Once the number of reads approaches 1x coverage, more reads increase the counts on true edges, but add very few more. However, because error edges tend to be unique, the number of error edges continues to grow linearly with the coverage. This creates a significant problem: by the time coverage is high enough to distinguish true edges from error ones (e.g. 20x), the error edges will typically outnumber true ones by at lan order of magnitude or more. A succinct representation is particularly valuable in this case, because the number of bits per edge depends on the entropy of the distribution of edges. The relationship between the number of edges (as a proportion of the total number of possible edges) and the number of bits per edge is shown in the figure below. When the number of edges is very small, the number of bits per edge is close to log(n). When the number of edges reaches n/2, the number of bits per edge is 2, and when the number of edges approaches n, the number of bits per edge converges on 0. In practice, of course, the number of edges will never be more than a minute fraction of n, however, even for the case of n = 4^25, representing a million edges requires not 50 bits per edge, but 32, and representing a billion edges requires only 22 bits per edge.



A Succinct Graph Representation

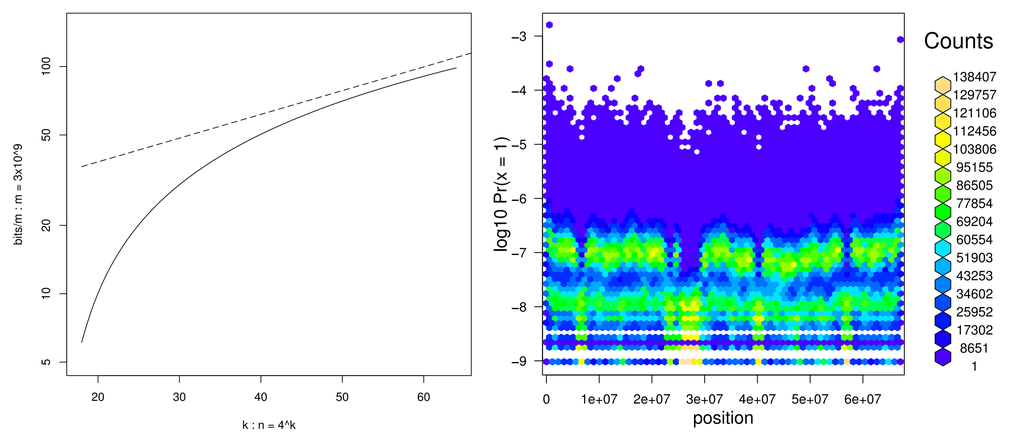
The de Bruijn assembly graph is a subset of the de Bruijn graph, which is defined over all ρ-mers. Given |E| ρ-mers, an obvious representation would be to use |E| 2ρ bit quantities (m log n, in the succinct data structures literature). The information theoretic lower bound on the number of bits required to represent the set of ρ-mers is

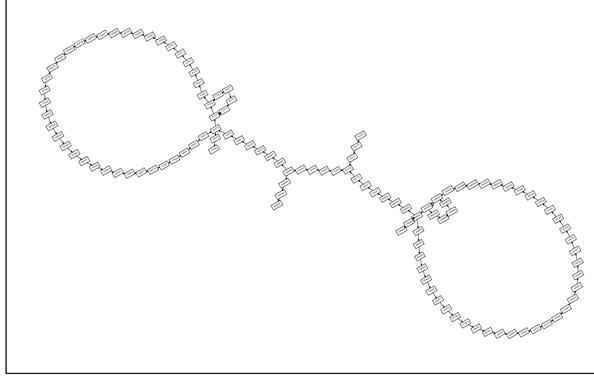
$$bits\text{-}required = \log_2 \binom{4^\rho}{|E|}$$

The figure on the lower left shows for a fixed |E|, how the space required scales with ρ. The dashed line shows 2ρ|E|, and the solid line shows the minimum bits required according to the formula above.

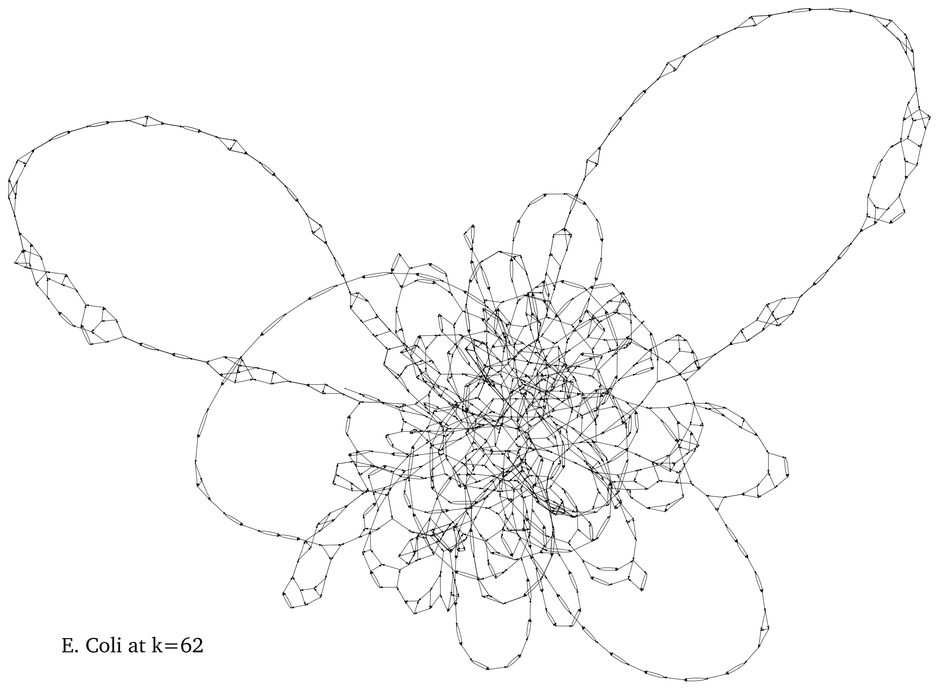
Succinct data structures (see e.g. [1]) are able to represent sparse sets in space within a few percent of the lower bound. The representations in the literature assume that the distribution of edges is uniform. The figure below right shows for the human genome, the density of edges across the bitmap for k=27. Clearly the density of the bitmap is highly non-uniform. Further work will find succinct representations that deal with this non-uniformity more efficiently than current approaches.

The ρ-mer counts are stored using a directly addressable variable length code [4]. The code works by storing the least significant byte of each value and an associated bitmap with a 1 for each position that has further bytes. For each value that has further bytes, the next least significant value is stored, and so on. This can be thought of as an instance of Golomb-Rice code. It is illustrated in the bottom figure.





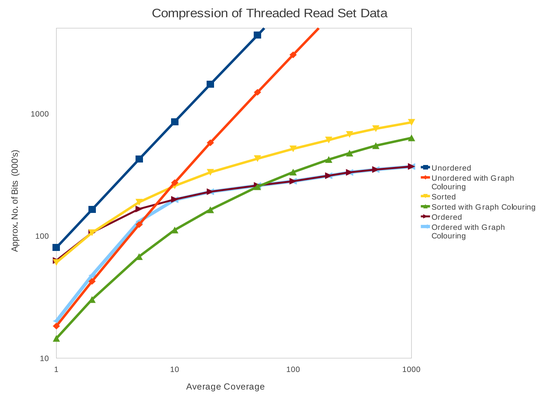
Four views of the same sequence. The top panel shows the original sequence. The panel above left shows the 6-mers present in the sequence. The panel above right shows a 64x64 bitmap depicting which 6-mers (or reverse complements) are present. The figure on the lower left shows the de Bruijn graph corresponding to the bitmap.



E. Coli at k=62

Threading Reads Through the Assembly Graph

The de Bruijn assembly graph discards some of the information present in the original sequence reads. In order to capture and utilize that information, as well as utilizing the information gained from paired reads, we need to thread the reads through the graph. This is done by annotating certain points in the graph with the set of reads that pass through those points. The most obvious ways of doing this result in space usage proportional to the number of reads. First, we note that reads do not need to be assigned a unique number; reads overlaying separate parts of the graph can share a number (if carefully assigned). We do this by constructing an interference graph for each read, and performing heuristic graph colouring to assign numbers to reads. However, while this improves the constant factors for the required space, the space required is still linear in the number of reads. We have a proof that there is a numbering of the reads that enables us to represent the sets of reads using an amount of space that grows with the logarithm of the number of reads. The proof assumes an oracle which tells us which position in the genome each read comes from, and the numbering of the reads arises from sorting the reads in order of that position. However, since in practice we don't have such an oracle, knowing neither the underlying genome nor the source location of each read, we sort the reads according to a carefully chosen key, which does a good job of approximating the order. The figure below shows the number of bits required to represent random sets of synthetic reads from the *s. aureus* genome. The figure shows the result of six different algorithms for generating an assignment of read numbers: random ordering, without graph colouring, and with it; sorting the reads, then numbering without graph colouring and with it; and using the oracularly determined ordering, without graph colouring, and with it. Note the log-log scale.

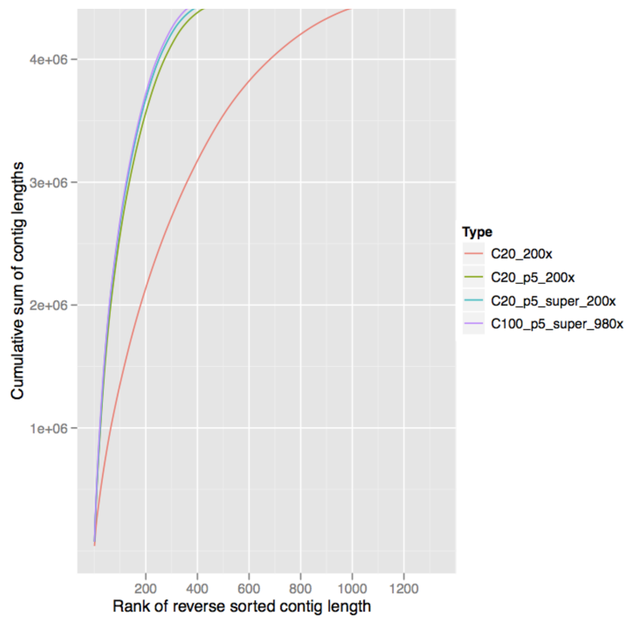


Results

Our results in [3] show that using the succinct representation we have developed, we are able to build a *proof-of-concept* assembly of a human genome (Yoruba male, sample no. NA18507) on a single server with 32GB RAM in 51 hours. The table below summarizes the results in comparison to the published results for ABySS on the same data set. Note that our *proof-of-concept* assembly is very simplistic, and has only very elementary error removal, and contig construction algorithms.

	ABySS		Proof-of-Concept	
Cores		168		8
Nodes		21		1
Total RAM (GB)		336		32
Min. contig size	≥ 100bp	≥ 1kbp	≥ 100bp	≥ 1kbp
Number of contigs	4,348,132	549,522	7,693.288	41,292
Median size (bp)	253	1,463	165	1,146
Mean size (bp)	484	1,703	224	1,219
Max. size (bp)	15,911	15,911	22,032	22,032
N50 size (bp)	870	1,731	250	1,176
Number of contigs > N50	674,953	188,171	1,994,863	17,939
Sum (Gbp)	2.10	0.94	1.72	0.05

The figure below shows some more recent results demonstrating improved assembly resulting from more complete error removal, and read threading. Current work involves using pairing information from paired-end and mate-pair libraries to gain a further improvement in the quality of the assembly.



Get Gossamer at
http://www.nicta.com.au/bioinformatics

[1] Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In Proc. ALENEX Workshop on Algorithm Engineering and Experiments, SIAM, Philadelphia (2007)
[2] Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences, In Proc. SPIRE Int. Conf. on String Processing and Information Retrieval, LNCS 5280 (2008)
[3] Conway, T., Bromage, A.: Succinct data structures for assembling large genomes. Bioinformatics 2011 27 (3).
[4] Brisaboa, N. R., Ladra, S., and Navarro, G. (2009). Directly addressable variable-length codes. In J. Karlgren, J. Tarhio, and H. Hyyro, editors, SPIRE, volume 5721 of Lecture Notes in Computer Science, pages 122–130. Springer.
[5] Ramen, R., Ramen, V., and Rao, S. S.: Succinct dynamic data structures, In Proc. 7th International Workshop on Algorithms and Data Structures (WADS '01), pages 246-252. Springer-Verlag