Supplementary Material for: "A space and time-efficient index for the compacted colored de Bruijn graph"

Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava and Rob Patro

1 The need to post-process TwoPaCo results

The TwoPaCo compacted de Bruijn graph has two main differences with the format that is expected by pufferfish. First, it is not the case that k-mers and their reverse complements will appear only once in the TwoPaCo compacted de Bruijn graph. Specifically, the inflection point of "palindromic" sequences, where a string is followed by its own reverse complement are not necessarily treated as junctions by the TwoPaCo algorithm. Thus, a unitig may contain both a k-mer and its reverse complement. pufferfish does not allow this. Second, the GFA generated by TwoPaCo assumes that *edges* of size at least $k+1$ will act as GFA segments, implying that they will overlap by k nucleotides. However, we require that segments be of at least size k and overlap by exactly $k-1$ nucleotides.

2 Differences between the edge-explicit and induced-edge de Bruijn graph

Dataset	Tool	num. of unitigs	num. equivalence classes
Human Transciptome	TwoPaCo	528,931	NA
	Pufferfish	526,935	343,407
	Kallisto	542,540	360,747
Human Genome	TwoPaCo	37,069,091	NA
	Pufferfish	35,776,802	1,894,231
	Kallisto	38,967,126	1,894,235
Bacterial	TwoPaCo	82,644,380	NA
	Pufferfish	82,991,439	29,267,832
	Kallisto	84,935,209	36,234,837

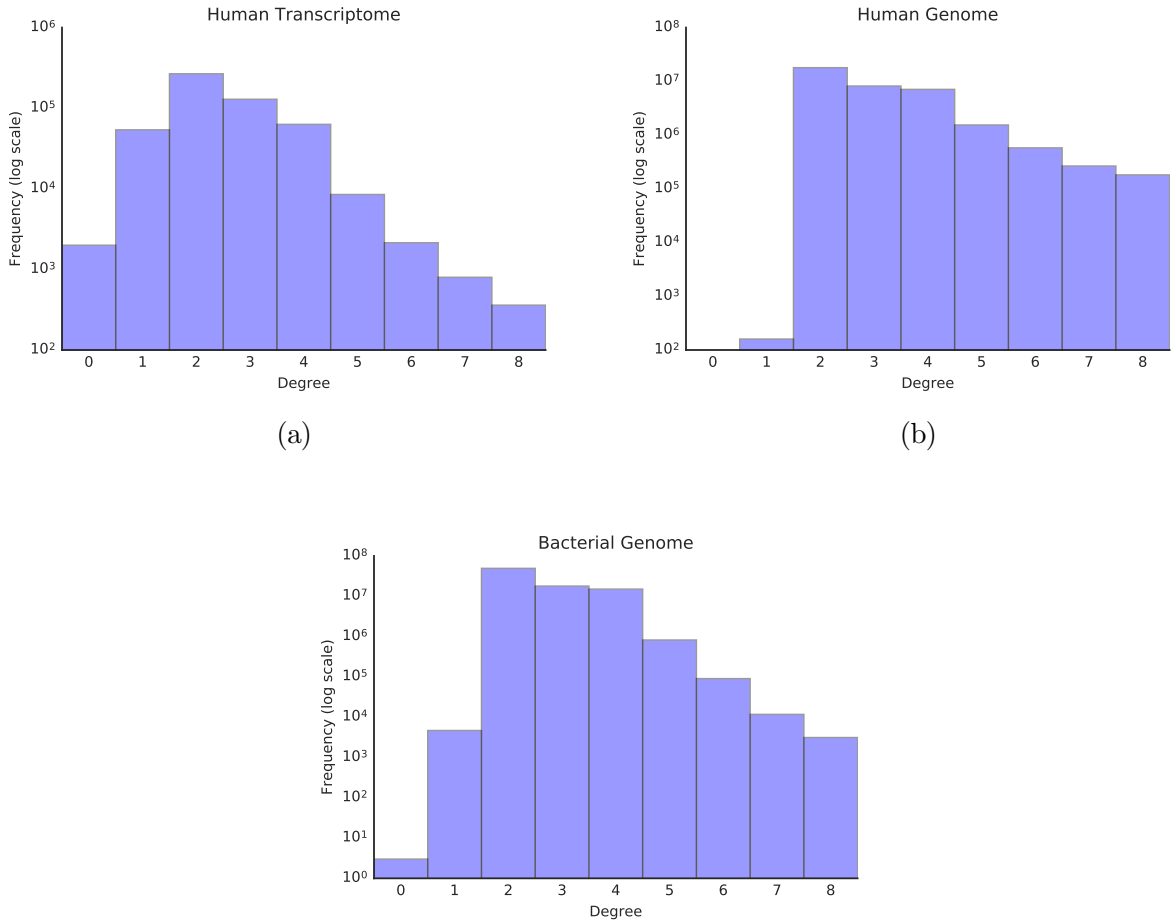
Supplementary Table 1: The number of unitigs in the compacted de Bruijn graph reported by different tools. We also provide total number of equivalence classes for pufferfish and kallisto since these numbers are computed, and differ as well.

Dataset	Tool	k-mer	num. of unitigs	num. equivalence classes
Human Transciptome	Pufferfish	29	556,637	357,667

Supplementary Table 2: The number of unitigs and number of equivalence classes in the compacted de Bruijn graph reported by Pufferfish while changing the k-mer size to 29.

Decreasing the k-mer size from 31 to 29 for, (i.e. using $k-2$ instead of k), yeilds the statistics shown in Supplementary Table 2 from the pufferfish produced gfa file. We observe that the number of unitigs and number of equivalence classes are closer to those of the induced-edge compacted de Bruijn graph using a value of 31. We note that since all tools considered here require k to be

odd, it was not possible to build the induced-edge compacted de Bruijn graph using a value of $k=32$ or the edge-explicit compacted de Bruijn graph using a value of $k=30$.



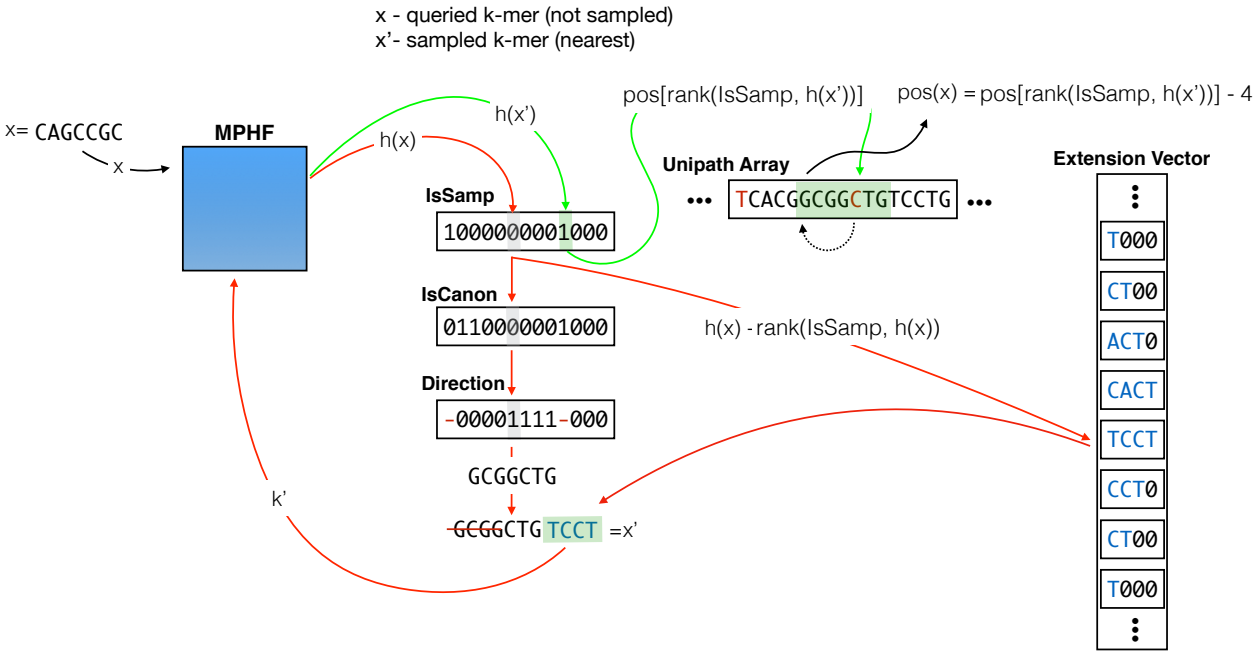
Supplementary Figure 1:
The edge density histogram of unipaths in the compacted de Bruijn graph of the human transcriptome (a), the human genome (b), and the collection of bacterial genomes (c). The distribution is highly skewed toward low degrees, suggesting that only a small fraction of possible edges are, in fact, present.

3 Empirical edge density of different colored compacted de Bruijn graphs

In this section, we provide the results on the distribution of edge density over different datasets in figure Supplementary Figure 1.

4 The sparse pufferfish query procedure

This section contains the sparse query algorithm (algorithm 1) as well as a detailed example of performing a query in the sparse pufferfish index (Supplementary Figure 2).



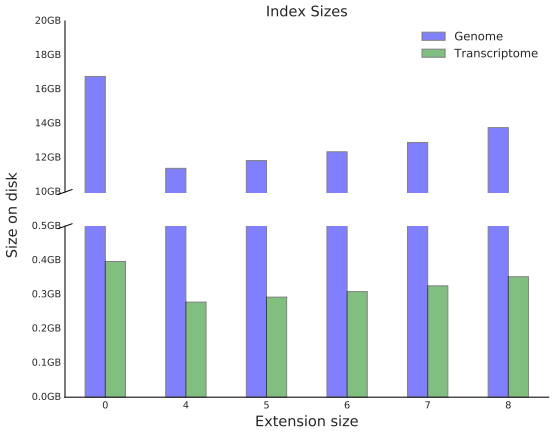
Supplementary Figure 2: An illustration of searching for a particular k-mer in the *sparse* pufferfish index with sample factor (s) of 9 and extension size (e) of 4. Vector `isSamp` has length equal to the number of valid k-mers, and `isCanon` and `Direction` have length equal to the total number of non-sampled k-mers. The minimum perfect hash yields the index $\mathrm{h}(\hat{x})$ for $x=$`CAGCCGC` in `isSamp`, where we discover that the k-mer's position is not sampled. Since `isCanon`$[\mathrm{h}(\hat{x})-$`rank(isSamp,`$\mathrm{h}(\hat{x}))]=0$ we know that the k-mer, if present, is not in the canonical orientation in `useq`. Since x *is* in the canonical orientation, we must reverse-complement it as $\bar{x}=$`GCGGCTG` before adding the extension nucleotides. Then, based on the value of `Direction`$[\mathrm{h}(\hat{x})-$`rank(isSamp,`$\mathrm{h}(\hat{x}))]$, we know that to get to the closest sampled k-mer we need to append the extension nucleotides to the right of $\bar{x}$. The extension is extracted from the `QueryExt` vector. Since extensions are recorded only for non-sampled k-mers, to find the index of the current k-mer's extension, we need to determine the number of non-sampled k-mers preceding index $\mathrm{h}(\hat{x})$. This can easily be computed as $\mathrm{h}(\hat{x})-$`rank(isSamp,`$\mathrm{h}(\hat{x})$, which is the index into `QueryExt` from which we retrieve this k-mers's extension. We create a new k-mer, x', by appending the new extension to $\bar{x}$, and also removing its first $e=4$ bases. Then, we repeat the same process for the new k-mer x'. This time, the k-mer is sampled. Hence, we go directly to the index in `useq` suggested by `pos[rank(isSamp,`$\mathrm{h}(\hat{x'})$`)]`. To check if the original k-mer we searched for exists, we need to compare the k-mer starting from $e=4$ bases to the left of the current position with the *non-canonical* version of the original k-mer (since the sampled k-mer x' was arrived at by extending the original query k-mer by 4 nucleotides to the right). Generally speaking, once we reach a sampled position, to check the original query k-mer, we need to move in `useq` to either the right or the left by exactly the distance we traversed to reach this sample, but in the opposite direction.

Algorithm 1 Find Query Offset

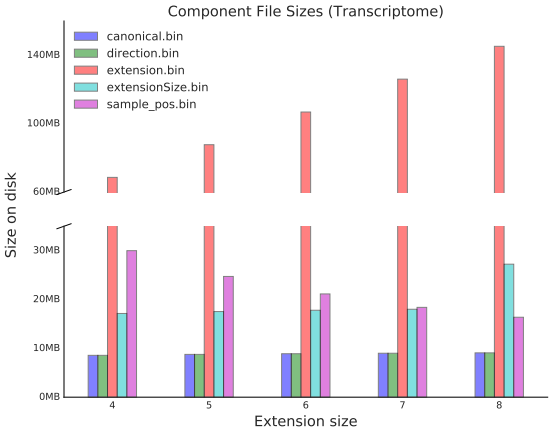
procedure FINDQUERYOFFSET
$x \leftarrow$ the query k-mer
$\hat{x}_q \leftarrow \hat{x}$
$i \leftarrow \mathrm{h}(\hat{x})$
$offset \leftarrow 0$
while $i < N$ and **not** `isSamp`$[i]$ **do**
$extIdx \leftarrow$ `i-rank` (`isSamp`, i)
$extNuc \leftarrow$ `QueryExt` [extIdx]
$extLen \leftarrow$ `len`(extNuc)
if `isCanon`$[extIdx]$ **and** `Direction`$[extIdx]$ **then**
$x \leftarrow \hat{x}[extLen:] + extNuc$
$offset \leftarrow offset + e$
if not `isCanon`$[extIdx]$ **and** `Direction`$[extIdx]$ **then**
$x \leftarrow \bar{\hat{x}}[extLen:] + extNuc$
$offset \leftarrow offset + e$
if `isCanon`$[extIdx]$ **and not** `Direction`$[extIdx]$ **then**
$x \leftarrow extNuc + \hat{x}[:-extLen]$
$offset \leftarrow offset - e$
if not `isCanon`$[extIdx]$ **and not** `Direction`$[extIdx]$ **then**
$x \leftarrow extNuc + \bar{\hat{x}}[:-extLen]$
$offset \leftarrow offset - e$
$i \leftarrow \mathrm{h}(\hat{x})$
if $i \geq N$ **then return** -1
$p_i \leftarrow$ `pos` [`rank` (`isSamp`,i)] - offset
if `useq`$[p_i : p_i + k] == \hat{x}_q$ or `useq`$[p_i : p_i + k] == \bar{\hat{x}}_q$ **then**
return p_i
else
return -1

5 The effect of different extension sizes on the sparse pufferfish index

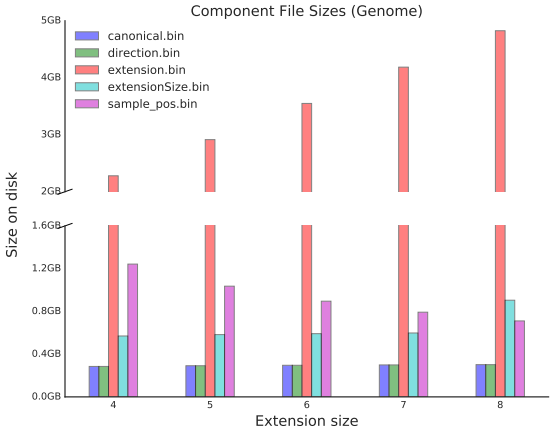
We explored the empirical effect of selecting different extension sizes for the sparse pufferfish index. Specifically, as one increases the extension size, the amount of space in the index required for storing the extensions of non-sampled k-mers increases, while the number of sampled k-mers (and hence the space requires to store the `pos` table) decreases. For moderate size references, since we require only $\sim\lg(|\texttt{useq}|)$ bits per position, and we are storing many more extensions than sampled positions, the extension vector quickly reaches a similar size to the position table. We consider a range of different values in Supplementary Figure 3. We note that we here only consider varying the extension size e while keeping the sampling rate s at $2{\cdot}e{+}1$, which ensures that at most one extra lookup is required for non-sampled positions. Further space savings could easily be obtained (at the cost of slower lookups) by considering $s{>}2{\cdot}e{+}1$.



(a)

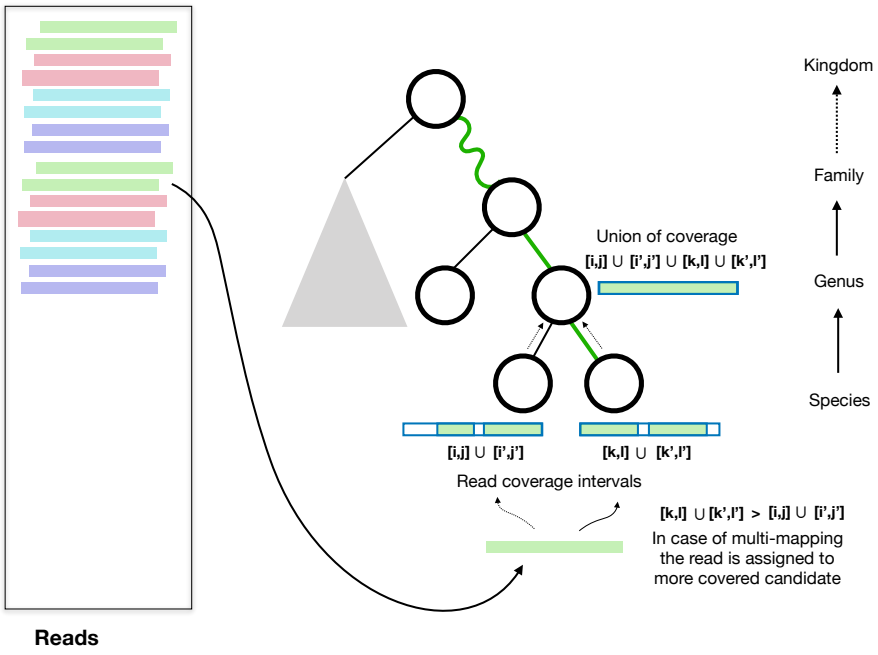


(b)



(c)

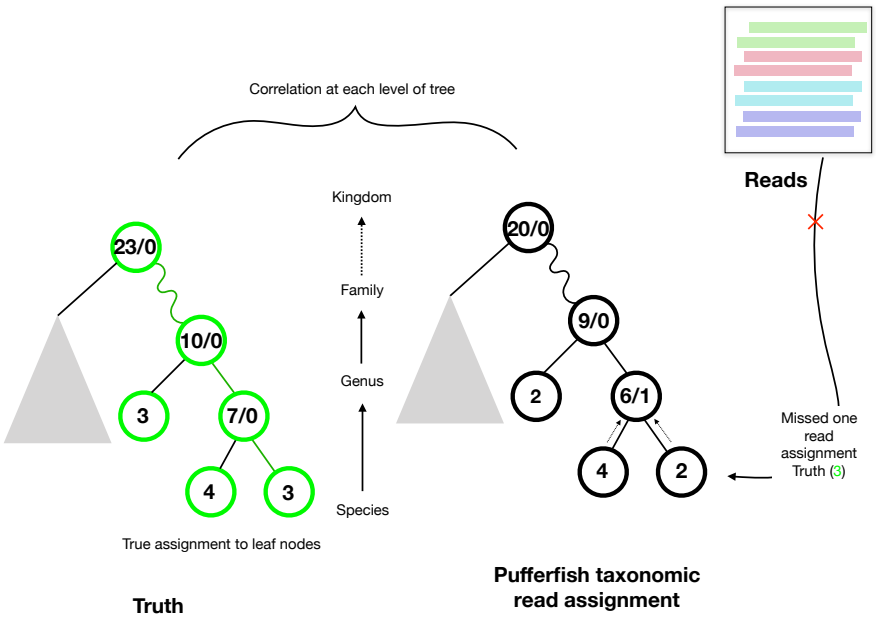
Supplementary Figure 3:
(a) Total size of indices produced by pufferfish on human genome and transcriptome sequence for dense (extension size 0) and sparse indices with different levels of sparsity (extension sizes 4,5,6,7,8). Individual sizes for different components of sparse indices are plotted for human transcriptome (b) and genome (c).



Supplementary Figure 4: An example showing the process by which pufferfish maps reads to leaf nodes, propagates coverage information up the taxonomy and assigns a read to a given node.

6 Read assignment and accuracy assessment methodology

Supplementary Figure 4 provides an illustration of how the Kraken algorithm for taxonomic read assignment has been modified to account for coverage by uni-MEMs under the mappings produced by pufferfish. Supplementary Figure 5 illustrates how true and predicted taxonomic assignments are aggregated over the entire tree to allow full-taxonomy assessment of accuracy.



Supplementary Figure 5: An example showing how aggregated values are propagated up the taxonomy to enable full-taxonomy assessment. On the left, true reads are generated from leaf nodes, and the abundance of internal nodes is populated based on the number of true nodes originating from each subtree. On the right, reads are assigned by a taxonomic assignment method (sometimes to internal nodes — e.g. when they are ambiguous at a lower level). The assessment metrics are computed based on the correspondence of the true and predicted abundances over *all* nodes of the taxonomy.

	Unfiltered		Filtered	
	Time(mm:ss)	Memory (G)	Time(mm:ss)	Memory (G)
Kraken	24:57	70.5	37:08	70.5
Clark	06:21	72.4	09:06	86
pufferfish	10:58	34.7	10:58	34.7

Supplementary Table 3: The time and memory requirements of Kraken, Clark, and pufferfish for performing taxonomic read assignment for the $\sim 100M$ of an experimental dataset (sample P00497 from SRR1749083) in both the unfiltered (left) and filtered (right) modes. All tools were run with 16 threads. Clark is the fastest of the tools, followed by pufferfish and then Kraken. Clark and Kraken require a similar amount of memory (except when Clark is run in "full" mode), approximately twice as much memory as is required by pufferfish.

7 Time and space benchmark for a large dataset

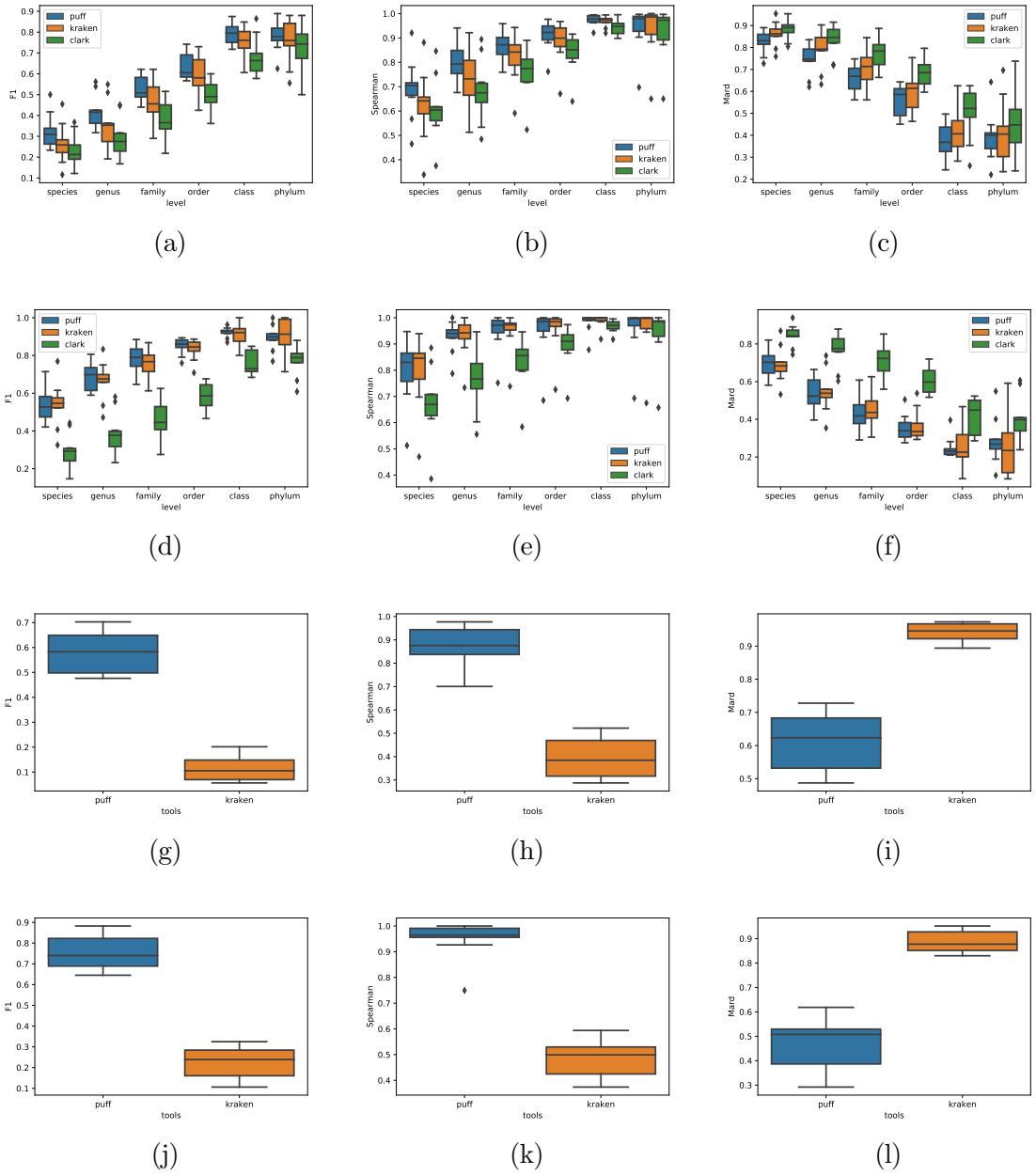
In table 3 we provide the time and memory required to map reads and then aggregate the counts for each taxon in the taxonomy tree by pufferfish, Kraken, and Clark. The benchmarks are provided for both cases of running the tools with and without their default filtering options. we ran all the tools on a set of 100M reads (P00497_Deep.100.Mreads derived from the database provided by (McIntyre *et al.*, 2017)). The filtering options we used for each of the tools are as follows:

- For Kraken we ran `kraken-filter` with the option `--thresh 0.2` which filters any mapping with fewer k-mers than 20% of the non-ambiguous k-mers for the read (this is the setting recommended by (McIntyre *et al.*, 2017)).
- For Clark we ran the abundance estimation command with `--highconfidence` which applies the default recommended confidence filtering for assigning reads.
- For pufferfish we filtered any mapping with less than 44 nucleotides coverage. The motivation for this cutoff is that a coverage of 44 nucleotides corresponds to the minimum number of nucleotides in a 100bp read that can be covered using only 20% of the k-mers— this default value, therefore, was inspired by the filtering option we use for Kraken.

Full details of the experimental protocols used to produce these results and evaluate the different methods can be found in the repository at `https://github.com/COMBINE-lab/pufferfish_experiments`.

8 Level-specific evaluation of taxonomic read assignment for tools Kraken, Clark, and Pufferfish

In Supplementary Figure 6 we provide a breakdown of the performance of the different taxonomic read assignment methods under a number of different, specific taxonomic ranks, as measured using the Spearman correlation, the mean absolute relative difference (MARD) and the F1 score.



Supplementary Figure 6: The F1 score (a), Spearman correlation (b) and MARD (c) at different, specific taxonomic ranks for the three taxonomic read classifiers when using their unfiltered setting. The F1 score (d), Spearman correlation (e) and MARD (f) at different, specific taxonomic ranks for the three taxonomic read classifiers when using their default filtering mechanisms (i.e., 20% for Kraken, 44 nucleotides for pufferfish and "high-confidence" assignments for Clark). The third and fourth rows show the accuracy metrics for pufferfish and Kraken for nodes in the tree that are not considered at the labeled ranks covered in the first two rows (with and without default filtering, respectively). Specifically, these are nodes in the tree that nonetheless give rise to reads (but are labeled either as "no-rank" or with some other taxonomic rank not considered in the first two rows). Clark is not included in this comparison since it is not able to make assignments to such nodes.

9 Time and space benchmarks for running TwoPaCo in single-threaded mode

The results reported as benchmarks for TwoPaCo in Table 1 in the main paper are generated by running the construction step with multiple threads (specifically 16 threads). Here we report the time and space requirements for running TwoPaCo using only a single thread.

Memory (MB)			Time (h:m:s)		
Human Transcriptome	Human Genome	Bacterial Genomes	Human Transcriptome	Human Genome	Bacterial Genomes
4,251	8,430	16,817	0:08:49	01:29:26	14:15:54

Supplementary Table 4: This table shows time and memory requirements for running TwoPaCo with a single thread to construct a compacted de Bruijn graph.

References

McIntyre, A. B. R., Ounit, R., Afshinnekoo, E., Prill, R. J., Hénaff, E., Alexander, N., Minot, S. S., Danko, D., Foox, J., Ahsanuddin, S., Tighe, S., Hasan, N. A., Subramanian, P., Moffat, K., Levy, S., Lonardi, S., Greenfield, N., Colwell, R. R., Rosen, G. L., and Mason, C. E. (2017). Comprehensive benchmarking and ensemble approaches for metagenomic classifiers. *Genome Biology*, **18**(1).