Supplementary materials for "Spectrum preserving tilings enable sparse and modular reference indexing",

Jason Fan, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro

S.1 The mapped position query (MRP) for unitig-tilings

Algorithm 4: The MRP query for unitig-tilings

```
def mrp(x):
    tup ← k2u(x)
    if tup = ∅ then
        return [ ]
    (i, p) ← tup
    occ_i ← num-occs(U_i)
    ans ← [ ]
    for r ← 0 to occs_i do
        (n, s) ← u2occ(i, r)
        ans[r] = (n, s + p)
    return ans
```

S.2 Spectrum preserving tilings with *orientations*

We extend the definition of spectrum preserving tilings (without orientations) given in Section 3.1, to formally define spectrum preserving tilings (SPT) *with orientations*. An SPT with orientation allows tiles (members of a spectrum preserving string set) to occur in either a *forward* orientation as stored in memory as a nucleotide sequence, or a *backwards* orientation as the *reverse complement* of the stored sequence.

With respect to representing reference genomic sequences, using SPTs with orientations is particularly useful because it avoids redundantly encoding and storing occurrences of a k-mer *and* the reverse complement of said k-mer. Furthermore, since most sequencing technologies are agnostic to strands of DNA sequences, considering orientations enables the simultaneous and canonical representation of both corresponding strands of an indexed genomic sequence.

Also, as in [9], we consider only *odd* k-mer sizes so that no k-mer is its own reverse complement.

Tiling sequences of tile *and orientation* pairs. Given a fixed k-mer size, k, a tiling sequence T_n in $\mathcal{T}$ is instead sequences of *tile-orientation* pairs where each occurrence is defined to be $T_{n,m} = (U_i, o)$, for some unitig $U_i \in \mathcal{U}$ and an orientation $o \in \{0, 1\}$. Here, $o = 1$ indicates that the unitig U_i occurs in a forward orientation and $o = 0$ indicates that it occurs in the *backwards* orientation with reverse complement sequence $\overline{U_i}$. Notationally, $\overline{U_i}$ is the string that is the reverse complement of U_i where U_i is reversed and each nucleotide is replaced with its complement.

Let us define the $\texttt{spell}(U_i, o)$ function for a unitig orientation pair to return the forward sequence U_i if $o = 1$ and the backwards, reverse complement sequence $\overline{U_i}$ otherwise. Abusing some notation, when $T_{n,m} = (U_i, o)$, let $\texttt{spell}(T_{n,m}) = \texttt{spell}(U_i, o)$.

Then formally, a spectrum preserving tiling with orientations for a reference collection $\mathcal{R} = \{R_1, \ldots, R_N\}$ tiles each reference sequence R_n with sequences *spelled by* occurrences of tile-orienation pairs. Specifically, each R_n can be reconstructed by gluing together the sequences that tile-orientation pairs *spell*. For each R_n that is tiled by M_n tile occurrences, the SPT satisfies the property that:

$$R_n = \texttt{spell}(T_{n,1})[w_{n,1} : w_{n,1} + l_{n,1}] \ \oplus_k \ \ldots \ \oplus_k \texttt{spell}(T_{n,M_n})[w_{n,M_n} : w_{n,M_n} + l_{n,M_n}]$$

S.2.1 Returning queries with orientations

Accordingly, when indexing an SPT with orientations the mapped reference position query, k-mer-to-tile query, and tile-to-occurrence query also return orientations. Here, we extend and reintroduce the queries defined in Section 3.

1. **The mapped reference position (MRP) query** Given any k-mer x, the MRP query enumerates the positions and orientations of all occurrences of x in $\mathcal{R}$. Precisely, each returned occurrence is a tuple (n, p, o), that specifies that k-mer x occurs in reference n at position p with orientation o. That is, if $o = 1$, then x occurs in the forward orientation as $R_n[p : p+k] = x$. Otherwise, the reverse complement occurs as $R_n[p : p + k] = \overline{x}$. If a k-mer does not occur in some $R_n \in \mathcal{R}$, the query returns an empty list.
2. **The kmer-to-tile query**: Given a k-mer x, $\texttt{k2tile}(x)$ returns (i, p, o) — the identity of the tile U_i that contains x, the offset (position) into the tile U_i where x occurs, and the *orientation* of how x occurs. That is, $\texttt{k2tile}(x) = (i, p, 1)$ if $U_i[p : p + k] = x$, and $\texttt{k2tile}(x) = (i, p, 0)$ if $U_i[p : p + k] = \overline{x}$ where x occurs in the backwards orientation as the reverse complement. If x is not in $\mathcal{R}$, $\texttt{k2tile}$, $\texttt{k2tile}(x)$ returns $\emptyset$.
3. **The tile-to-occurrence query**: Given the r-th occurrence of the tile U_i, $\texttt{tile2occ}(i, r)$ returns the tuple (n, o, s, w, l) that encodes how *and in what orientation* U_i tiles the reference R_n. Let the r-th occurrence of U_i be a tile-occurrence $T_{n,m}$ on $\mathcal{T}$ where $T_{n,m} = U_i, o$ for some orientation o. Then $\texttt{tile2occ}(i, r)$ returns $(n, o, s_{n,m}, w_{n,m}, l_{n,m})$. When $\texttt{tile2occ}(i, r) = (n, o, s, w, l)$ and $o = 1$, the r-th occurrence of U_i occurs on R_n at position $(s + w)$, with the sequence $U_i[w : w + l]$. When $\texttt{tile2occ}(i, r) = (n, o, s, w, l)$ and $o = 0$, the r-th occurrence of U_i occurs on R_n at position $(s + w)$, with the sequence $\overline{U_i}[w : w + l]$.

With some arithmetic bookkeeping considering orientations and lengths, the MRP query with orientations can again be decomposed into the two corresponding k-mer-to-tile and tile-to-occurrence queries that also return orientations. Although not introduced with respect to an SPT, the `pufferfish` index developed by Almodaresi et al. [9] is implemented exactly this way as an index over an SPT *with orientations* of unitigs.

S.3 `Pufferfish2`: the `pred` query with orientations

`Pufferfish2`'s sampling scheme and the `pred` query can be applied when considering orientations — our implemented tool does exactly this. Below, we extend Section 4 to fully specify the introduced sampling scheme and the `pred` query when orientations are considered.

S.3.1 Predecessor and successor nucleotides

When SPT references, predecessor and successor nucleotides are defined and obtained with respect to sequences on the *references*. Specifically, the predecessor nucleotide is the first nucleotide of the last k-mer on of the preceding unitig-occurrence *as spelled with the corresponding orientation* of the occurrence. The successor nucleotide is defined in the same manner.

Suppose $T_{n,m} = (U_i, o)$, and $T_{n,m-1} = (U_j, \omega)$, and let the unitigs have lengths ℓ_i and ℓ_j, respectively. We say that, $T_{n,m-1}$ precedes $T_{n,m}$ with predecessor nucleotide p *and orientation* o. Concretely, p is the first nucleotide on the last k-mer of the preceding unitig, with $p = \texttt{spell}(T_{n,m-1})[\ell_j - k]$. We say that, $T_{n,m}$ succeeds $T_{n,m-1}$ with successor nucleotide s *and orientation* ω. Accordingly, the successor nucleotide, s, is the last nucleotide on the first k-mer of the succeeding unitig, with $s = \texttt{spell}(T_{n,m})[k]$.

S.3.2 Storing nucleotide-orientation pairs in `ptab` and `stab`

Instead of storing only nucleotides, `pufferfish2` stores nucleotide-orientation pairs in implementation. That is, for each occurrence $T_{n,m} = (U_i, o)$ that is the r-th occurrence of a not-sampled unitig U_i,

$$\texttt{ptab}[i][r] = (\texttt{pred-nuc}(T_{n,m}), o).$$

And for each occurrence $T_{n,m} = (U_i, o)$ that is the r-th occurrence of *any* unitig U_i,

$$\texttt{stab}[i][r] = (\texttt{succ-nuc}(T_{n,m}), o).$$

In summary, `ptab` and `stab` store for each corresponding unitig-occurrence, the nucleotides that succeed and precede it as they occur on a tiled reference, *and* the orientation of said occurrence.

S.3.3 Computing the `pred` query by matching ranks of predecessor-orientation and successor-orientation pairs

When orientations are considered, computing the `pred` query requires matching ranks of predecessor-orientation and successor-orientation pairs. Critically, any time a pair of unitigs occur as a successor-predecessor pair in *fixed* orientations, the corresponding pair of predecessor and successor nucleotides are *consistent* and also *fixed*. Furthermore, if an occurrence of the U_j in orientation ω precedes a unitig U_i with orientation o, *any other* occurrence of U_j that precedes U_i with orientation o must also occur with orientation ω. We state and prove this property with Theorem 1 and illustrate examples of both possible and impossible unitig-occurrences with Fig. S1.

Theorem 1 guarantees that whenever U_i occurs with orientation o with predecessor nucleotide p preceded by U_j, U_j must occur with fixed orientation ω with a fixed successor nucleotide s. We illustrate this correspondence in Fig. S1. Algorithm 5 implements `pred` with orientations considered.

To find the identity, j, of the preceding unitig occurrence, Algorithm 5 must construct the last k-mer of the corresponding occurrence U_j as it appears on the reference. Specifically, in Line 3 it spells U_i before extracting the overlapping $(k-1)$-mer. Here, `k2u` returns orientation, ω, of the queried k-mer on U_j, which must also be the orientation of the preceding unitig occurrence on the reference. Furthermore, the successor nucleotide, s, for the preceding occurrence of U_j must be the k-th nucleotide on U_i spelled with orientation o (Line 5).

Now, Algorithm 5 has all it needs to compute q, the unitig-rank of the preceding occurrence of U_j. Computing the rank of (p, o) in $\texttt{ptab}[i]$ yields the rank of the corresponding successor-orientation

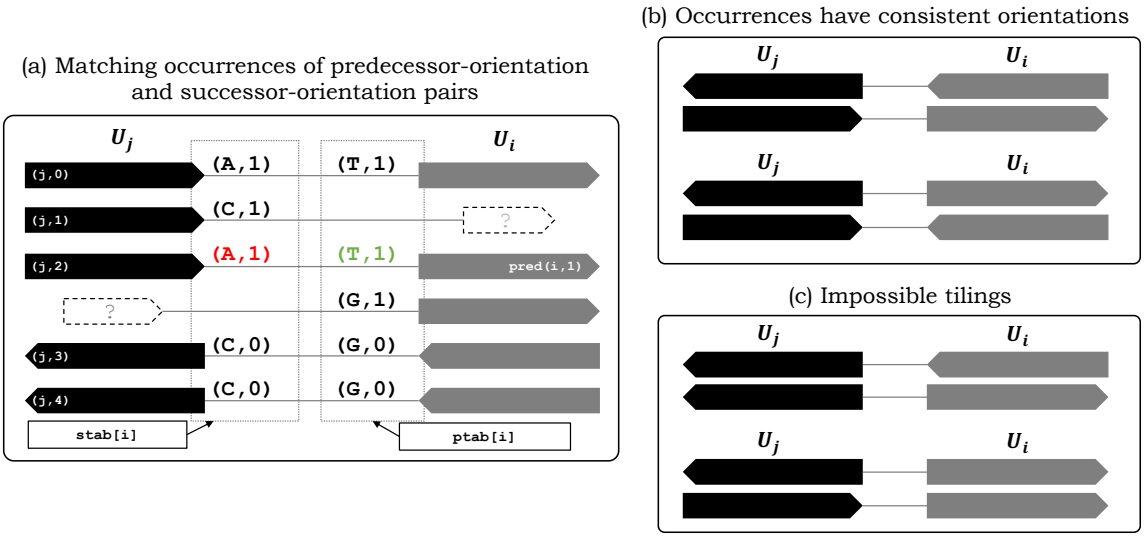

Fig. S1. Properties of the `pred` query for unitig-tilings with orientations. (a) Adjacent pairs of successor and predecessor unitigs have consistent and unique co-occurring pairs of predecessor nucleotide-orientation successor nucleotide-orientation pairs. (b) Whenever a pair of unitigs occur adjacently on the tiling, the orientation of *one* fixes the orientation of the other (for odd k-mer sizes). (c) That is, if U_j with orientation ω precedes a unitig U_i with fixed orientation o once, it cannot precede another occurrence (of U_i with orientation o) in the opposite orientation.

pair stored for the preceding unitig-occurrence. Finally, selecting for the successor-orientation pair (s, ω) in $\texttt{stab}[j]$ yields q.

Algorithm 5: The `pred` query with orientations

```
1 def pred(i, r):
2     (p, o) ← ptab[i][r]
3     y ← p ∘ spell(U_i, o)[: k − 1]
4     (j, _, ω) ← k2u(y)
5     s ← spell(U_i, o)[k]
6     t ← rank_(p,o)(ptab[i], r)
7     q ← select_(s,ω)(stab[j], t)
8     return (j, q)
```

S.3.4 Unitig-unitig occurrences have consistent orientations and predecessor-successor nucleotides

The key to the correctness of `pufferfish2`'s reference tiling traversal, by way of successor-orientation and predecessor-orientation pairs, is that predecessor-successor nucleotide pairs for adjacent unitig-occurrences are consistent and unique up to orientation. Whenever unitigs U_a and U_b overlap and tile with some given *fixed* orientations, corresponding successor and predecessor nucleotides are consistent and always the same. Below, we prove Theorem 1 that formally states this property.

Theorem 1. *Let unitigs U_a and U_b overlap and tile in orientations o and ω, with successor and predecessor nucleotides p and s. If any occurrence of U_a with orientation o is preceded by the nucleotide p, it must always be preceded by the same unitig U_b in the same orientation ω. Simultaneously, if any unitig U_b with orientation ω is succeeded by the nucleotide s, it must always be succeeded by the same unitig U_a in the same orientation o.*

Proof. Theorem 1 is result of the lemmas proved below. Lemmas 1 and 2 state that with fixed orientations and predecessor and successor nucleotides, the identities of successor-predecessor unitig pairs must be unique. Lemmas 3 and 4 state that with fixed predecessor and successor nucleotides for fixed unitig identities, the orientations of a successor-predecessor unitig pair must be unique.

Lemma 1. *Consider unitigs $U_i, U_j, U_k \in \mathcal{U}$. Let adjacent unitig occurrences $T_{a,b} = (U_i, o)$ and $T_{a,b+1} = (U_j, \omega)$ occur with successor nucleotide s. For any c, d, there does not exist another pair of adjacent occurrences $T_{c,d} = (U_i, o)$ and $T_{c,d+1} = (U_k, \omega')$ with the same succeeding nucleotide s but with $U_j \neq U_k$.*

Proof. Let us assume the contrary. Let z be the last $(k-1)$-mer on $\texttt{spell}(T_{a,b})$, which is the same as $\texttt{spell}(T_{c,d})$. Then the k-mer $z \circ s$ occurs on different unitigs U_j and U_k. However, this is a contradiction since any unique k-mer occurs in only one unique unitig.

Lemma 2. *Consider unitigs $U_i, U_j, U_k \in \mathcal{U}$. Let the occurrences $T_{a,b} = (U_i, o)$ and $T_{a,b-1} = (U_j, \omega)$ occur with preceding nucleotide p. There does not exist another pair $T_{c,d} = (U_i, o)$, $T_{c,d-1} = (U_k, \omega')$ in $\mathcal{R}$ where $U_j \neq U_k$, with the same preceding nucleotide s.*

Proof. This is symmetrical to Lemma 1.

Lemma 3. *Let $\{U_i, U_j\} \in \mathcal{U}$ Given unitig occurrences $T_{a,b} = (U_i, o)$ and $T_{a,b+1} = (U_j, 1)$ that tile R_a with successor nucleotide s. There does not exist another pair $T_{c,d} = (U, o)$, $T_{c,d+1} = (U_j, 0)$ in $\mathcal{R}$ with the same successor nucleotide s.*

Proof. Let us assume the contrary. Let z be the last $(k-1)$-mer on $T_{a,b}$ and $T_{c,d}$. Suppose $z \circ s$ is the first k-mer on U_i. The tiling on R_c implies that $\overline{z \circ s}$ is the first k-mer on $\overline{U}_j$ and that $z \circ s$ is the last k-mer on U_j. But the tiling on R_a implies that $z \circ s$ is the first k-mer on U_j. If $|U_j| = k$ and U_j is itself a k-mer, then the above implies $U_j = \overline{U_i}$. This cannot be the case, since we consider only odd-length k-mers, and no odd length k-mer can be equal to its reverse complement. If $|U_j| > k$, then z occurs in two distinct positions in U_j, this is again a contradication since any unique k-mer occurs in only one unique unitig.

Lemma 4. *Let $\{U_i, U_j\} \in \mathcal{U}$ Given unitig occurrences $T_{a,b} = (U, o)$ and $T_{a,b-1} = (V, 1)$ that tile R_a with predecessor nucleotide p. There does not exist another pair $T_{c,d} = (U, o)$, $T_{c,d-1} = (V, 0)$ in $\mathcal{R}$ with the same precedecessor nucleotide p.*

Proof. This is symmetrical to Lemma 3.

S.4 Constructing `pufferfish2` from `pufferfish`

Building `pufferfish2` requires a linear scan over the n total unitig-occurrences in the tiling sequences indexed by a given `pufferfish` index to collect predecessor and successor nucleotides.

The construction process is dominated by the time it takes to build the pair of wavelet matrices over all the predecessor and successor nucleotides of every unitig-occurrence. We note that since `pufferfish2` sparsifies and compresses and existing `pufferfish` index, it adopts `pufferfish`'s upstream preprocessing of unknown bases, where each `N` is replaced by a pseudo-random nucleotide. Constructing a wavelet matrix over an alphabet of size σ requires $O(n \lg \sigma)$ time and amounts to successive stable partitions of the encoded characters according to their bitwise representations. In the future, we plan to update `pufferfish2` to index input reference sequences directly.

S.5 Greedy unitig sampling with bounded traversal length s

Here we describe a *greedy* sampling scheme that greedily bounds traversal lengths to be at most of length s. To ensure that all backwards traversals terminate, the greedy sampling with integer paramater s first samples all unitigs that occur as the first occurrence of a tiling sequence, adds them to the set of sampled unitigs $\mathcal{U}_S$, and sets the corresponding bits in `isSamp` to 1 and all other bits to zero. Then, traversing tiling sequences in the order in which they appear, the greedy scheme maintains a counter of the distance to the last sampled unitig-occurrence. At each unitig occurrence $T_{n,m} = U_i$, if U_i is already sampled (i.e., `isSamp`$[i]$ is 1), the greedy scheme resets the counter to zero. Otherwise, the greedy scheme increases the counter by one. When the counter is greater than s, it samples the current unitig and resets the counter.

Although the greedy scheme is able to explicitly bound the traversal length it samples almost *all* unitigs when the length of tiling sequences become much larger than the number of unique unitigs. This is because, as implemented, `pufferfish2` samples *all occurrences* of a unitig, if said unitig is sampled. For example, when applying this sampling scheme to index a collection of seven human genomes, a greedy scheme with $s = 3$ samples 40% of unique unitigs that constitute more than 70% of unitig-*occurrences*. In this example, over 70% of `utab` must then be kept and uncompressed.

S.6 Optimizations for `pufferfish2`

Caching traversals. When enumerating all positions of a unitig with `u2occ`, `pufferfish2` caches the `k2u` query — the empirically slowest constant-time operation in the `pred` query (Line 4 in Algorithm 3). The purpose of this `k2u` query is only to find the *identity* of the preceding unitig given a unitig-occurrence's predecessor nucleotide. While a unitig may be preceded by *many* occurrences, preceding occurrences can have *at most* four unique unitig identities — one for each possible nucleotide. If U_i occurs more than once with U_j preceding it, U_j must always precede U_i with the same fixed predecessor nucleotide each time. For MRP queries, `pufferfish2` can avoid executing the redundant `k2u` queries (within `pred` queries) when the *same* nucleotide is prepended to different occurrences of U_i. Specifically, during MRP queries where the `pred` query is executed, `pufferfish2` caches the mapping from predecessor nucleotides to preceding unitig identities. In practice, `pufferfish2` maintains an efficient LRU cache to memoize Lines 3 and 4 in Algorithm 3.

Caching streaming MRP queries. In practice, a *stream* of successive MRP queries for different k-mers often land in the same unitig (e.g. when querying k-mers on a sequenced read). So, instead of performing redundant `u2occ` queries that may perform backwards traversals for the same unitig, `pufferfish2` maintains a cache for the `u2occ` query. When successive k-mers are found to be in the same unitig via the `k2u` query, `pufferfish2` checks a "streaming cache" to avoid performing

repeated `u2occ` queries for the same unitig. This caching scheme for "streaming" queries is also employed in [22].

Exiting early. In practice, programs such as read-mappers and aligners can exit early from the mapped reference position query if a queried k-mer is uninformative and occurs too frequently. With `pufferfish2`, instead of always computing the `u2occ` query for every occurrence in loop starting on Line 8, a caller of the MRP query can exit before the loop and avoid traversals altogether.

Interpolating between wavelet matrices and short linear scans. In practice, computing rank and select for short predecessor and successor nucleotide sequences (see Lines 6 and 7) is faster with a linear scan in an array than an operation in the wavelet matrix. So, for unitigs that occur at most 64 times, `pufferfish2` stores corresponding lists of nucleotides in packed arrays instead of wavelet matrices.

S.7 Cost estimate for Amazon Web Services (AWS) EC2 instances

Estimated prices for AWS EC2 instances in the "US East" region are obtained from `https://calculator.aws/#/estimate`. EC2 instances were specified with 500Gb of storage and 8 CPUs for 10 hrs per week of usage. Recommended EC2 `x2gd.4xlarge` instances have 258GiB of memory whereas `x2gd.2xlarge` instances have 128GiB of memory. As of writing, estimated cost per month for `x2gd.4xlarge` and `x2gd.2xlarge` are 651USD and 351USD, respectively.

S.8 Future work

1. Given the SPT definition, we have introduced strategies for sampling $\mathcal{U}$. That is, either a tile has *all* or *none* of its occurrences sampled. Yet, nothing theoretically prevents one from instead sampling over $\mathcal{T}$, so that *occurrences* are sampled according to their position on a tiling regardless of their unitig-identities. This approach introduces some extra complications but provides the benefit of allowing sampling schemes to *trivially* bound the worst-case traversal length, while also directly controlling the fraction of sampled entries by sampling every s-th occurrence. The question of what sampling strategy works better in practice is an interesting open question.
2. Does a smaller SPSS imply a smaller SPT? Currently, this is not clear, since working with unitigs dispenses entirely the space required for $\mathcal{W}$, $\mathcal{L}$, and $\mathcal{S}$, so that a smaller SPSS may increase the space for representing the tiling given the need to encode $\mathcal{W}$, $\mathcal{L}$, and $\mathcal{S}$.
3. We have provided an intuitive notion of how a "good" or "desirable" SPT looks: Ideally, an SPT amenable to indexing has few but long tiles and short tilings. Yet, rather than separating the problem of finding a set of tiles and then efficiently representing the tiling it induces, there is a more general optimization problem: Given a set $\mathcal{R}$ of references, what is the SPT that minimizes the overall space, or the query time? Likewise, we may ask, if one has knowledge of the queries that are to be performed, how might the selection of samples be optimized to minimize the expected query time?
4. We have implemented one, specific, sparsification and compression scheme to reduce the size of SPTs. However, as hybrid encoding strategies have proven successful in optimizing the representation of k-mer-to-tile mappings [22], we may expect the same to be true of the tile-to-occurrence

map. For example, long occurrence lists may compress well with traditional information retrieval compression schemes [31], and delta-encoding-like schemes may prove very effective in compressing the occurrence lists for tiles that almost always co-occur. In general, hybrid encoding and compression schemes likely hold great promise in tackling this problem.

5. Finally, we have considered here only exact and lossless representation of SPTs. However, many successful indexing schemes for problems like read mapping avoid indexing all sub-words, instead, for example, indexing only minimizers [32] or altering the sampling strategy in highly-repetitive regions. Thus, for many important applications it may not be necessary to have a *complete* and *lossless* index over the underlying SPT and it is possible that a *lossy* index over an SPT could be made much smaller and faster still.