Benchmark Results for Gap Decorator Variants

August 21, 2018

1 The Approaches

Three gap decorator variations were benchmarked against each other and against a sequence allowing regular `seqan3::dna15` letters and the `gap::GAP` symbol. Let's denote with n the ungapped sequence length, k the number of gaps (not number of gap symbols!) and m the number of gap symbols.

1. **Gapped Sequence** An `std::vector` of type `gapped<dna15>` which grows and shrinks on direct insertion/deletion of gaps.
   - Direct addressing
   - Overhead because of explicit gap character storage
   - `dna` cast to wider data type `gapped<dna>`
2. **Gap vector** Using a compressed bit vector of length |sequence| + # gaps
   - Bit vector is `sdsl::sd_vector` with rank and select support structures with access time depending on the number of bits set.
   - Set bit indicates gap, else underlying sequence minus the rank
   - Bit vectors are not modifyable (except the `sdsl::int_vector`), hence any insertion or deletion operation results in a rebuilt with costs of $\Theta(n)$
3. **Anchor List** Gaps are stored as anchors, i.e. a list of tuples with **container-relative** positions and lengths of gaps
   - Insertion, deletion, and random access positions are virtual (=gapped sequence space), the anchor gaps have to be therefore accumulated to compute the virtual position
4. **Anchor set** A red-black tree stores the **virtual** anchor positions, an additional dictionary resolves the gap length for a given position.

- An `std::set`* is used for key management. The `std::set` does not allow handing over a lambda function for user-defined sorting. Therefore, key-value tuples which have to be sorted only by key cannot be used when one needs the guarantee of $\log_2(k)$ access time.
- Associated gap lengths are stored in an `std::unordered_map` with constant access, but larger storage consumption.
- To improve random access runtimes from $\Theta(k)$ to $\Theta(\log_2 k)$, the gap lengths are stored in an accumulated fashion. To retrieve the actual gap length one substracts the accumulator of the preceeding gap.

Operation	**Gapped Sequence**	**Gap Vector**	**Anchor List**	**Anchor Set**
`operator[]`	$\Theta(1)$	$\mathcal{O}(\log(\frac{n+m}{m}))^\dagger$	$\Theta(k)$	$\Theta(\log_2 k)$
`erase_gap`	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(k)$	$\Theta(\log_2 k)$
`insert_gap`	$\Theta(n+m)$	$\Theta(n+m)$	$\Theta(k)$	$\Theta(\log_2 k)$

Table 1: Theoretical runtimes and space consumptions with k = number of gaps, m = accumulated gap length, n = underlying (gap-free) sequence length.

	Gapped Sequence	**Gap Vector**	**Anchor List**	**Anchor Set**
Space	$\Theta\big((n+m)\cdot\lvert$`gapped<dna>`$\rvert\big)$	$\Theta\big(m(2+\log\frac{n}{m})+n\cdot\lvert$`dna`$\rvert\big)$	$\Theta(k+n\cdot\lvert$`dna`$\rvert)$	$\Theta(k+n\cdot\lvert$`dna`$\rvert)$

Table 2: Theoretical space consumptions with k = number of gaps, m = accumulated gap length, n = underlying (gap-free) sequence length including the unaligned sequence.

2 Benchmark Results

To create realistic gapped sequences or gaps for insertion, gap lengths are sampled from a histogram distribution published by Goonesekere et al. ‡ for each position, which results in a rather dense distribution of short gaps (40%-45% gap proportion). After sampling the "gapped" sequences were resized to have total sequence size of 2^i. All four data structures are then initialized with the same gap vector within the same experiment. Each experiment is repeated 32 times for sequences with lengths in range $[2, 2^2, ..., 2^{15}]$, and 8 times for lengths in range $[2^{16}, ..., 2^{18}]$. The benchmarks cover three user scenarios:

*which is implemented via a red-black tree

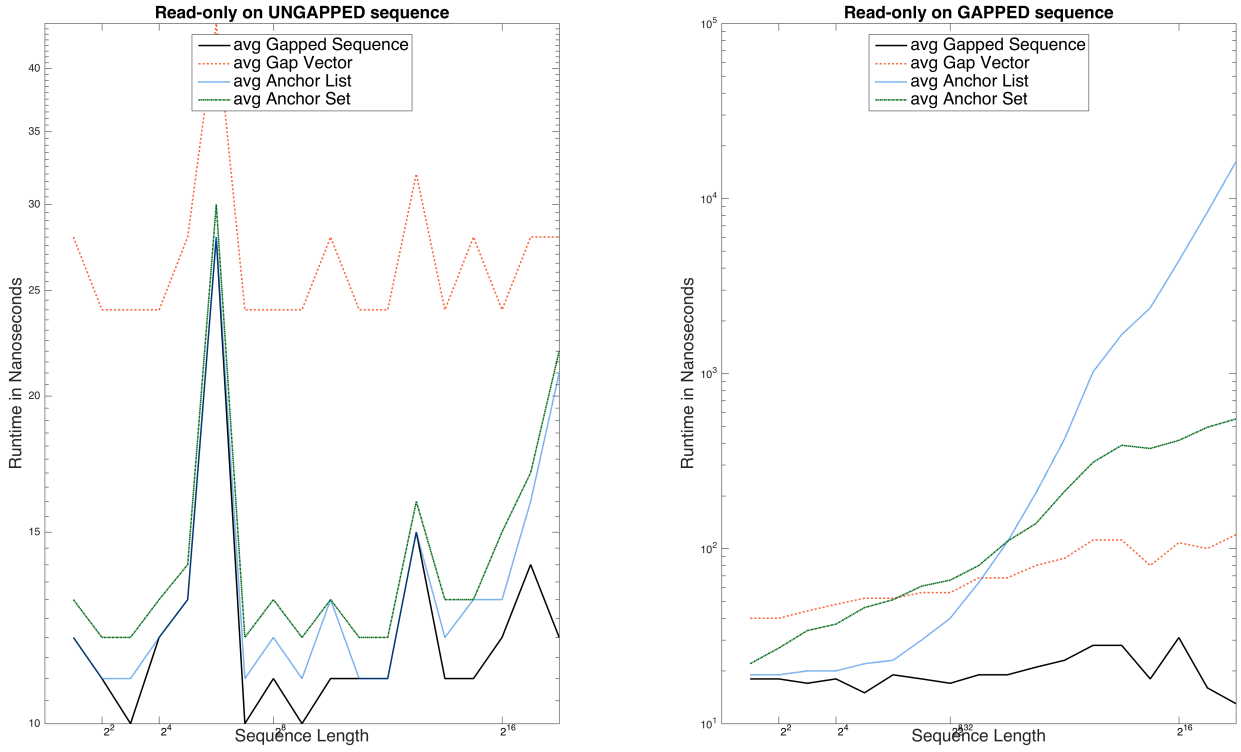
†i.e. costs of `rank_support_sd`

‡see Table 1 in `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC419611/`, columns were summed up and

Benchmark 1 Read-only on ungapped or gapped sequence

Benchmark 2 Insert/Delete at **random positions** into ungapped or gapped sequence

Benchmark 3 Insert/Delete from **left to right** into ungapped or gapped sequence

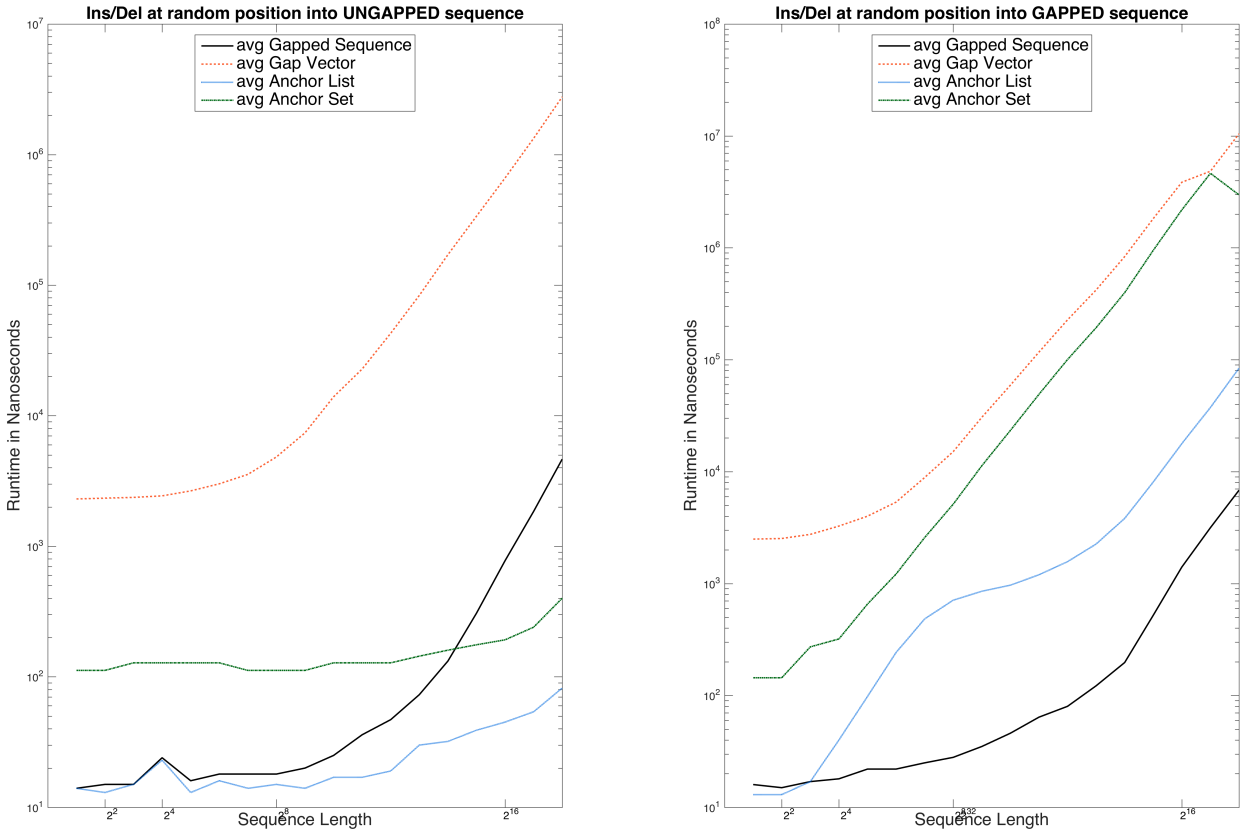


Benchmark 1: Left: Read-only on ungapped sequence, i.e. gap structures are empty. Right: Read-only on gapped sequence.

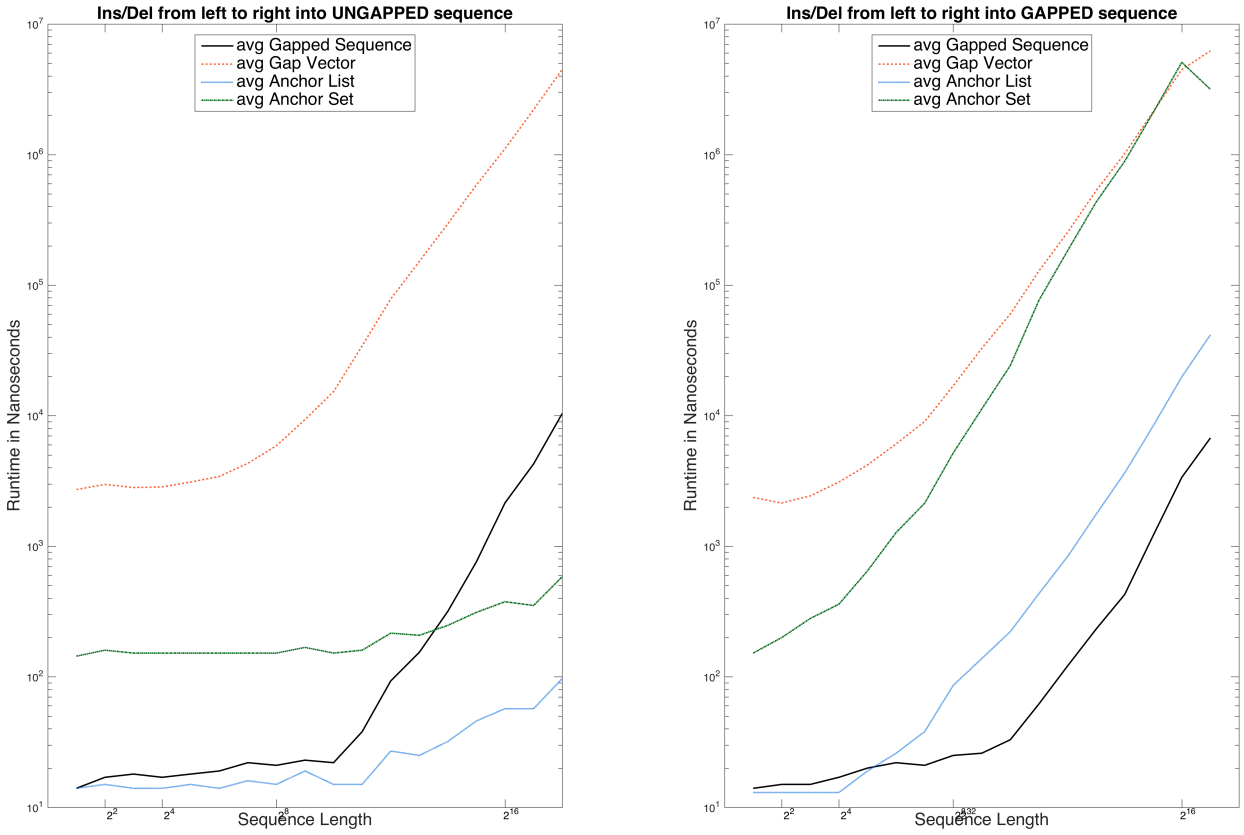
3 Summary

3.1 Results

- The gapped sequence (`gapped<dna15>`) always outperforms the other approaches, except when gaps are inserted from left to right into an empty sequence (see Benchmark 3). In this case the Anchor List approach is always faster, and the Anchor Set approach when sequence lengths exceeds 2^{12}.



Benchmark 2: Left: Insert and delete at random positions into ungapped sequence. Right: Insert and delete at random positions into gapped sequence.



Benchmark 3: Left: Insert and delete from left to right into an ungapped sequence. Right: Insert and delete from left to right into a gapped sequence.

- The Gap Vector (`sdsl::sd_vector`) never outperforms the three other approaches, but should be considered to be provided in SeqAn3 when only read operations are performed and a transformation to a `gapped` sequence is undesirable.
- When comparing anchor list versus anchor set for Benchmark 1 the anchor set outperforms the anchor list for sequence lengths exceeding 2^{12}. However, both approaches are outperformed by the gap vector. For Benchmark 2 (insert/delete at random position) and Benchmark 3 (insert/delete from left to right) the anchor list always outperforms the anchor set.

3.2 Recommendations

Usecase	**Data Structure**
Read-Only	Gap Vector or Gapped Sequence
Ins/Del at Random Position into **Ungapped** Sequence	Anchor List
Ins/Del at Random Position into **Gapped** Sequence	Anchor List or Gapped Sequence
Ins/Del Left to Right into **Ungapped** Sequence	Anchor List
Ins/Del Left to Right into **Gapped** Sequence	Anchor List or Gapped Sequence

normalized