Sassy: Fuzzy Searching DNA Sequences using SIMD

Rick Beeloo[1,*,†] **and Ragnar Groot Koerkamp**[2,3,*,†]

[1]Department of Theoretical Biology and Bioinformatics, Utrecht University, [2]Department of Computer Science, ETH Zurich and [3]Department of Computer Science, Karlsruhe Institute of Technology

*Corresponding authors: biobeeloo@gmail.com (RB) and ragnar.grootkoerkamp@gmail.com (RGK)
†These authors contributed equally to this work.

Abstract

Motivation. Approximate string matching (ASM) is the problem of finding all occurrences of a pattern in a text while allowing up to k errors. Many modern methods use *seed-chain-extend*, which is fast in practice, but does not guarantee finding *all* matches with $\leq k$ errors. However, applications such as CRISPR off-target detection require exhaustive results.

Methods. We introduce Sassy, a library and tool for ASM of short patterns in long texts. Sassy splits the text into 4 parts that are searched in parallel, and uses bitvectors in the text direction rather than the pattern direction. This has compexity $O(k\lceil n/W\rceil)$ when searching a random text of length n, where $W = 256$ is the SIMD width, and provides significant speedups for small k.

Separately, we allow matches of the pattern to extend beyond the text for an *overhang cost* of e.g. $\alpha = 0.5$ per character, to find matches near contig or read ends.

Results. Sassy is $4\times$ to $15\times$ faster than Edlib for patterns ≤ 1000bp, and can search text with a throughput near 2 Gbp/s. Likewise, Sassy is over $100\times$ faster than parasail. We apply Sassy to CRISPR off-target detection by searching 61 guide sequences in a human genome. Sassy is $100\times$ faster than SWOffinder and only slightly slower (for $k \leq 3$) than CHOPOFF, for which building its index takes 20 minutes. Sassy also scales well to larger k, unlike CHOPOFF whose index took over 10 hours to build for $k = 5$.

Availibility. Sassy is available as library and binary at https://github.com/RagnarGrootKoerkamp/sassy, and archived at swh:1:dir:e884758dce5777a441bc2799dc8824e563c5f97b.

Key words: Approximate string matching, pattern matching, fuzzy string searching, semi-global alignment, edit distance, bitpacking, SIMD, DNA

1. Introduction

Approximate string matching (ASM) is the problem of finding all matches of a pattern P of length m in a text T of length n with at most k errors [Navarro, 2001]. In this paper, we consider errors under the unit-cost edit distance, also known as Levenshtein distance [Levenshtein et al., 1966]. ASM has applications in many different fields. Specifically in bioinformatics, instances of ASM are CRISPR off-target detection [Yaish et al., 2024, Roux et al., 2025] and searching barcodes for demultiplexing [Cheng et al., 2024, Beeloo et al., 2025].

Recent years have seen a large number of papers on speeding up the related problem of (semi-)global alignment by using faster implementations (bitpacking, SIMD), faster algorithms (A*), and better banding heuristics (see Section 1.2). Simultaneously, there is a lot of research on *mapping*: aligning, say, 1 kbp reads against static text indices that can range from megabases to gigabases in size, without the guarantee of finding *all* matches. Since this guarantee is important for many bioinformatic applications, we identify that there is no modern, SIMD-based tool for ASM. Sassy (SIMD Approximate String Searcher) fills this gap.

1.1. Contributions

Sassy is a conceptually simple but highly efficient command line tool and Rust library for approximate string matching. Sassy targets patterns with length up to around a thousand characters. It supports both ASCII and (IUPAC) DNA sequences, runs on both AVX2 (x86-64) and NEON (ARM), and comes with C and Python bindings. The underlying algorithm does not require a precomputed text index and instead can operate directly on the records of an input stream. This makes it especially suitable for e.g. searching a pattern while streaming DNA reads, one-off searches in assembled genomes, and reference-free analysis.

On a high level, our main contributions are:

1. We argue that while similar, semi-global alignment, mapping, and ASM are all distinct problems, and that for certain applications, exact methods for ASM are required and currently not available.
2. We define what it means to "report all matches", and choose to report only local minima by default. We note that reverse-complementing inputs can give different results (Figure 2).
3. We develop an efficient implementation of ASM. Algorithmically this has two small novelties: 1) bit-packing in the text direction, rather than the pattern direction, and 2) intra-sequence parallellism by splitting the text into 4 chunks that are processed in parallel using $W = 256$ bit SIMD. This leads to expected-case complexity $O(k\lceil n/W\rceil)$ when matching against random text, and $O(m\lceil n/W\rceil)$ in the worst-case when excluding the time for tracebacks.

© The Author 2026.

4. In Appendix B, we introduce an *overhang cost* $\alpha = 0.5$ that allows and controls the cost of *overhanging* alignments extending beyond the text.
5. Sassy is $4 - 15\times$ faster than Edlib for patterns up to length 1000 with up to 5% divergence.
6. Sassy is $100\times$ faster than Swoffinder for CRISPR off-target detection, and equally fast or faster than the index-based CHOPOFF while reporting identical matches.

1.2. Previous work

ASM has been extensively studied between 1980 and 2000, mostly concluding in the bit-packing algorithm of Myers [1999]. For word size $w = 64$, this has worst-case complexity $O(\lceil m/w \rceil n)$, or expected-case complexity $O(\lceil k/w \rceil n)$ on random text. Since then, research has shifted to other types of pairwise alignment. Indeed, both *global alignment* and *mapping* are very active areas of research on similar but slightly different problems. Some methods developed for those problems can also be applied to ASM. Unfortunately, they usually do not guarantee to return all matches, either because they only return best matches, as in *semi-global alignment*, or because of their heuristic nature in case of mappers. Before discussing the older results on ASM itself in detail, we first cover some recent work on these related problems, so that the differences can be appreciated.

Global alignment. In *global alignment*, the pattern P is aligned against the *entire* text T. The lengths $m := |P|$ and $n := |T|$ are typically relatively close to each other, and may range from tens to millions of bases. The classical Needleman-Wunsch [Needleman and Wunsch, 1970] (or Wagner-Fischer/Levenshtein [Wagner and Fischer, 1974, Levenshtein et al., 1966]) algorithm requires $O(nm)$ time and space, although space can be reduced to $O(\min(m, n))$ when only the alignment cost is needed. While no algorithm breaks the worst-case $O(n^{2-\varepsilon})$ barrier under SETH [Backurs and Indyk, 2015], many practical methods achieve sub-quadratic performance on typical inputs. For instance, Ukkonen's band-doubling method [Ukkonen, 1985b] runs in $O(ns)$ time, where s is the edit distance, and diagonal-transition approaches [Myers, 1986, Ukkonen, 1985a] attain $O(n + s^2)$ both in expectation on random texts and in practice. A recent implementation of this (for affine costs) is in WFA [Marco-Sola et al., 2021] and its extension BiWFA [Marco-Sola et al., 2023] with reduced memory usage. Another key technique in accelerating global alignment is bit-packing, pioneered by Myers in 1999 [Myers, 1999]. Rather than processing each DP cell individually, cost differences can be stored in word-size $w = 64$ bit vectors, allowing the processing of 64 DP states at once. This reduced the time complexity to $O(n\lceil m/w \rceil)$, or $O(n\lceil s/w \rceil)$ with banding. This bit-packing is implemented in the commonly used tool Edlib [Šošić and Šikić, 2017]. Modern CPUs can process more than 64 bits in SIMD registers (e.g. 256 bits for AVX2 or 128 bits for NEON). To effectively use parallelization, the DP matrix is often broken into smaller regions [Farrar, 2006, Wozniak, 1997, Liu and Steinegger, 2021], allowing parallel processing such as in KSW2 [Li, 2018, Suzuki and Kasahara, 2018], BSAlign [Zhang et al., 2019], and SeqMatcher [Espinosa et al., 2024]. Additionally, unlike for ASM, heuristics such as X-drop can be employed to reduce the search space [Altschul et al., 1990, Suzuki and Kasahara, 2018, Liu and Steinegger, 2021, Walia et al., 2024], while losing the guarantee that the best alignment is found. QuickEd [Doblas et al., 2025] first computes an approximate banded alignment and uses this as input for an exact alignment. A*PA and A*PA2 instead bound the search region by using A*, and retaining the guarantee that an optimal alignment is found [Groot Koerkamp and Ivanov, 2024, Groot Koerkamp, 2024].

Semi-global alignment. In *semi-global* alignment, a pattern P is aligned to a *substring* of a longer text T, and gaps at the start and end of T do not incur a penalty. Like in global alignment, only the alignment(s) with the lowest number of errors are reported. Semi-global alignment can either be between a short pattern and a much longer text ($m \ll n$, e.g. searching a read in a reference genome), or between two sequences of similar length ($m \approx n$, e.g. refining a mapped read). This approach was first introduced by Sellers [1979, 1980], and later termed *semi-global*[1] by Gotoh [1999], who also termed *global* alignment. It is implemented in tools such as Parasail [Daily, 2016], SeqAn [Reinert et al., 2017], Edlib [Šošić and Šikić, 2017], and more recently Ish [Stadick, 2025]. When $m \approx n$, semi-global alignment can benefit from adaptive banding methods as developed for global alignment, but this is not the case when $m \ll n$. There, some methods (parasail, Seqan, Ish) simply compute the entire $O(mn)$ DP matrix, while others (Edlib, Sassy) often only compute the top $O(k)$ rows [Myers, 1999]. Thus, these two regimes lead to completely different algorithms.

Cost models. A *cost model* defines what constitutes an *error* and the cost associated to each error. This concept originated in the early 1900s with systems designed to detect misspelled names by sound [Odell and Russell, 1918]. In the 1950s, Hamming distance was introduced for binary codes, measuring the number of differing bit positions [Hamming, 1950], or in the context of DNA, the number of mismatches between two equal-length strings. Around a decade later, the Levenshtein distance was formalized [Damerau, 1964, Levenshtein et al., 1966], that allows insertions and deletions alongside substitutions. Importantly, Levenshtein distance uses a *unit cost* model, assigning a cost of 1 to each edit, making it computationally efficient. However, this assumption is unrealistic for large insertions or deletions, as deleting or inserting long segments often represents a single biological event. This led to the development of gap-affine models, where gap opening and extension have different costs [Gotoh, 1982, Altschul and Erickson, 1986, Marco-Sola et al., 2021]. For Sassy, we use unit-cost edit distance for its computational efficiency and the assumption that long indels are rare when aligning relatively short patterns.

Approximate string matching. As mentioned before, the goal of ASM is to find all matches of a pattern P in a text T with $\leq k$ errors [Galil and Giancarlo, 1988, Navarro, 2001]. The key distinction from semi-global alignment is that not just the single best match should be reported, but that *all* matches with $\leq k$ errors should be reported. We first discuss *streaming* algorithms, where the text is not known in advance, as opposed to algorithms that pre-process the text T. Moreover, we focus on the *k-difference* variant that uses edit distance rather than the *k-mismatch* variant that uses Hamming distance [Chhabra et al., 2025, Fiori et al., 2021, Gottlieb and Reinert, 2025b].

Searching exact matches of patterns became popular through algorithms such as Knuth-Morris-Pratt [Knuth et al., 1977] and Boyer-Moore [Boyer and Moore, 1977]. With the development

[1] Confusingly, the term semi-global is sometimes also used for different variants of alignment. Parasail [Daily, 2016] uses it for all types of alignment that are not exactly global alignment, while Suzuki and Kasahara [2018] use it for *extension* alignment where the pattern has to match at the start of the text.

of different cost models in the 1980s, algorithms were created to detect inexact matches with k errors. Initially, *approximate string matching* described comparison of two strings [Ukkonen, 1983, Hall and Dowling, 1980], but later also described searching for a pattern as a substring of a text with $\leq k$ errors [Landau and Vishkin, 1985]. Sellers proposed an $O(mn)$ time algorithm [Sellers, 1980], which was improved to $O(m^2 + k^2 n)$ [Landau and Vishkin, 1985], and then $O(kn)$ [Ukkonen, 1985a]. The introduction of bit-parallelism [Baeza-Yates and Gonnet, 1992] led to complexities involving the word size w, such as $O(k\lceil m/w \rceil n)$ [Wu and Manber, 1992] in the famous agrep tool, $O(mn \log(\sigma)/w)$ [Wright, 1994], and eventually $O(\lceil m/w \rceil n)$ in Myers' algorithm [Myers, 1999]. Later optimizations targeted specific scenarios: short patterns or small k [Baeza-Yates and Navarro, 1999, Navarro and Baeza-Yates, 1999, Bille, 2011], fixed pattern lengths [Iliopoulos et al., 2001, Ho et al., 2017], and periodic texts [Cole and Hariharan, 2002]. Some were optimized for multi-pattern search [Muth and Manber, 1996, Baeza-Yates and Navarro, 1997], though here we focus on a single pattern.

Additionally, many algorithms leverage text pre-processing and indexing, such as text compression [Mäkinen et al., 2003, Kärkkäinen et al., 2000], suffix arrays [Landau and Vishkin, 1986, Manber and Myers, 1993, Huynh et al., 2004], and suffix trees [Ukkonen, 1993]. Others use pre-filtering with n-grams [Owolabi and McGregor, 1988, Jokinen and Ukkonen, 1991, Ukkonen, 1992, Sutinen and Tarhio, 1995, Bingmann et al., 2019], inexact hashing [Yao et al., 2010, McCauley, 2021], heuristics [Koehn and Senellart, 2010, Salmela et al., 2009], or search schemes on top of a birectional FM-index or move-index [Renders et al., 2022, 2024, Gottlieb and Reinert, 2025a, Depuydt et al., 2024, Renders et al., 2025]. Such methods thus implement completely different algorithms than the streaming-based method that we focus on in this paper, and are suitable for different applications.

SIMD parallellism. Overall, the complexity of index-free methods did not improve beyond Myers' $O(\lceil m/w \rceil n)$. Practical speedups emerged with larger SIMD word sizes with $W = 256$ or $W = 512$ bits. Improvements then involved optimal utilization of W. For example, BGSA [Zhang et al., 2019] uses inter-sequence parallelism to compare multiple sequences to the same pattern, since intra-sequence parallelism is limited when $m \leq W$ [Zhang et al., 2019]. An alternative approach is taken by A*PA2 [Groot Koerkamp, 2024], where the dependency between SIMD lanes is broken by tiling them diagonally. Yet another approach is taken by SeqMatcher [Espinosa et al., 2024], where AVX512 instructions are used to effectively use 512-bit integers. In contrast, Sassy splits the text into 4 chunks that are processed in parallel, somewhat similar to Farrar's striped method [Farrar, 2006] and as also used by SimdMinimizers [Groot Koerkamp and Martayan, 2025], for a complexity of $O(m\lceil n/W \rceil)$. This way, intra-sequence parallelism is maximized.

Mapping. In modern applications, the text is often an assembled genome of many gigabases, and the number of patterns (reads) to be searched is very large. This means that index-free methods are infeasible, and in practice, *mappers* drop the guarantee to find all matches in favour of speed. Thus, we consider mapping to be approximate[2] ASM.

[2] ASM is *approximate* in the sense that matches are allowed up to k errors. Mapping is *approximate* in the sense that it is an approximate algorithm that does not guarantee to find *all* such matches.

In the 1980s, with increasing sequence availability and the release of GenBank [Bilofsky et al., 1986] previous exact methods were no longer fast enough. Early mapping methods performed *exact* substring matches between the pattern P and database sequences, beginning with Wilbur and Lipman [1983] and followed by others using similar approaches [Lipman and Pearson, 1985, Ning et al., 2001, Wu and Watanabe, 2005] such as BLAST [Altschul et al., 1990].

Some methods controlled sensitivity based on the pigeonhole principle [Weese et al., 2012], while others tried to identify similar regions between P and the database sequences through spaced seeds [Ma et al., 2002, Rumble et al., 2009] or locality-sensitive hashing [Buhler, 2001]. As sequences got longer, also the number of seeds increased, leading to algorithms that reduced the number of seeds being stored, such as minimizers (e.g. Minimap2) [Li, 2018, Jain et al., 2020], strobemers (e.g. StrobeAlign) [Tolstoganov et al., 2024], or by hashing subsequences instead of substrings (e.g. SubseqHash2) [Li et al., 2024].

However, in benchmarks these mappers do not detect all mapping locations: they can achieve over 99% sensitivity but not full coverage [Banović Đeri et al., 2024], and their performance heavily depends on parameter settings such as the seed length [Oliva et al., 2021].

Applications of ASM. The earliest methods for ASM, developed in the 1980s, proved directly useful for biological problems. For example, Myers [1986] used ASM to find a 16-nucleotide binding site of the LexA protein in a 48 kb virus genome. Today, ASM supports diverse applications, including read demultiplexing [Cheng et al., 2024], genome polishing [Tonkin-Hill et al., 2020], and CRISPR off-target searching [Chaudhari et al., 2020].

We focus on the latter due to its clinical relevance. CRISPR and its associated Cas proteins form an adaptive immune system in bacteria and archaea, evolved to defend against foreign nucleic acids such as bacteriophage and plasmid DNA [Mojica et al., 2009]. In this system, foreign DNA is precisely cut using a template called single guide RNA (sgRNA). When the target DNA is flanked by a protospacer adjacent motif (PAM) — for example, 5'-NGG-3' in *Streptococcus pyogenes* — the CRISPR-Cas complex binds and cleaves the DNA, thereby neutralizing the invader. By modifying the sgRNA sequence, the CRISPR-Cas system can be programmed to cut virtually any DNA sequence. This technology has been applied to treat genetic diseases [Ledford, 2019], enhance crop traits [Jaganathan et al., 2018], and engineer microorganisms [Shapiro et al., 2018]. For an in-depth review, see Koonin and Makarova [2019]. Notably, on May 15th, 2025, CRISPR was used for the first time as a personalized treatment for a baby with carbamoyl phosphate synthetase 1 (CPS1) deficiency, a rare and life-threatening condition [Musunuru et al., 2025].

When CRISPR is engineered to target a specific sequence it is crucial that no other, unintended sequences are cut. This is called *off-target* cutting. Hence, computational tools to screen for such off-target sites have been developped. These include Cas-OFFinder [Bae et al., 2014], CRISPRitz [Cancellieri et al., 2020], SWOffinder [Yaish et al., 2024], and CHOPOFF [Labun et al., 2025], with the latter two representing the current state-of-the-art.

While CHOPOFF is much faster than SWOffinder, it requires a time-consuming step of building an index before searching. Given that human genetic variation affects off-target profiles [Scott and Zhang, 2017], we argue that with the advancement of personalized CRISPR therapies, there is a need for fast, index-free tools that are user friendly and robust to ambiguous bases.

2. Methods

We now describe our tool, Sassy. We start with some brief notation. Throughout the paper, we assume that we are given a *pattern* P of length $m := |P|$, and a *text* $T = t_0 \dots t_{n-1}$ of length $n := |T|$, which are both strings over an alphabet Σ of size $\sigma := |\Sigma|$. We write $T[i \dots j] := t_i \dots t_{j-1}$ for a right-exclusive substring of T, and we use $d(P, T[i \dots j])$ for the edit distance between P and $T[i \dots j]$. We write $\mathsf{rev}(T) := t_{n-1} \dots t_1 t_0$ for the reverse of T, and for DNA sequences, we define the *complement* $\mathsf{comp}(T)$ as the sequence where each base is replaced by its complement ($\mathsf{A} \leftrightarrow \mathsf{T}$ and $\mathsf{C} \leftrightarrow \mathsf{G}$, extended to IUPAC as well). The *reverse complement* is then $\mathsf{rc}(T) := \mathsf{rev}(\mathsf{comp}(T))$.

2.1. Approximate String Matching

We define approximate string matching following Navarro [2001], but restrict ourselves to edit distance only.

Definition 1 (Approximate String Matching, ASM.) Let P be a *pattern* of length $m := |P|$, and let T be a *text* of length $n := |T|$. Further, let $k \in \mathbb{N}_{\geq 0}$ be the maximum number of errors allowed. The problem of *approximate string matching*, $\mathsf{search}(P, T, k)$, is to find *all* end positions $j \in \{0, \dots, n\}$ in the text such that there exists an $i \in \{0, \dots, j\}$ such that the edit distance $d(P, T[i \dots j])$ between P and $T[i \dots j]$ is at most k.

What is a match? As defined above, a *match* is a position j in the text where an alignment of cost $\leq k$ ends. In practice, one might rather care about all *substrings* of T that have edit distance $\leq k$ to the pattern, i.e., all tuples (i, j) such that $d(P, T[i \dots j]) \leq k$ [Sellers, 1980, Definition 1]. Or, even more exhaustively, one could consider all *alignments* of P to substrings of T, where an alignment is a specific sequence of edits transforming P into $T[i \dots j]$.

In Sassy, we choose the first option: we find all end positions, and then do a *single* traceback for each of them.

When do we have a match? In practice, one is usually not quite interested in *all* matches. In particular, if there is an exact match ending in position j, all positions in $j - k$ to $j + k$ will have a cost $\leq k$ (see Figure 2). Thus, there are numerous options for which matches to report:

1. **All.** Report (matches ending in) *all* end positions with cost $\leq k$.
2. **Single best.** Report only a *single best* end position (if $\leq k$). Supported by Seqan.
3. **All best.** Report *all* positions where a match of *globally optimal* cost ends (if $\leq k$) as done in semi-global alignment as defined by Sellers [1980] and supported by Edlib and Seqan.
4. **Non-overlapping.** Report only end positions that are at least (roughly) m apart.

In Sassy, we take a different approach, that we argue is more principled:

5. **Local minima.** Report only *rightmost local minima* $\leq k$.

This is similar to the idea of Sellers [1980] to report all substrings $T[i \dots j]$ that can not be shrunk nor grown into a substring with lower edit distance, with the difference that we only report end positions, and that we only report a single match for each plateau of local minima.

ASM is not reverse-invariant. We note here that when reporting end positions, it is typically hard to guarantee that the matches reported by $\mathsf{search}(P, T, k)$ are in one-to-one correspondence with those reported by $\mathsf{search}(\mathsf{rev}(P), \mathsf{rev}(T), k)$, since the number of (local/global minima) end positions $\leq k$ can differ in the forward and reverse case, as exemplified in Figure 2. Reporting *all* matching substrings $T[i \dots j]$ would avoid this, but neither Sassy nor other tools do this in practice.

When searching reverse-complements is enabled, Sassy *is* invariant to reverse complements of the text: we search P in T and $\mathsf{rc}(T)$[3], so that both searches are in the natural direction of the pattern and searching P in $\mathsf{rc}(T)$ and $\mathsf{rc}(\mathsf{rc}(T)) = T$ gives the same result. Searching $\mathsf{rc}(P)$ in T or $\mathsf{rc}(T)$ *would* change the set of matches.

Traceback. Given the set of end positions that define a match, we can run a traceback from each of them to obtain both the position in the text where the match starts, and the corresponding alignment. Sassy simply recomputes the part of the DP matrix preceding each end position and traces back through that. The traceback greedily chooses matches and substitutions if possible, and then falls back in deletions and insertions, in that order.

2.2. Bitpacking and SIMD tiling

Figure 1 shows how Sassy applies both Myers' bitpacking [Myers, 1999] and SIMD. Using bitpacking, a *block* of $w = 64$ states of the DP matrix can be computed in parallel. Whereas other methods typically tile these bitvectors in the direction of the pattern, we tile them in the direction of the text. Pseudocode of the main search function of Sassy can be found in Appendix A.

SIMD. We use 256 bit SIMD widths using either AVX2 or two parallel NEON registers to compute 4 *lanes* of 64 bit words in parallel. We avoid dependencies between SIMD lanes by splitting the text into 4 chunks of $\lceil n/256 \rceil$ blocks each. Each lane then processes one chunk: we iterate over the 64 character blocks of each chunk, and for each block compute the m rows of the matrix.

As with Farrar's striped method [Farrar, 2006], there may be some "patching up" to do when a good alignment crosses the boundary between two chunks. In our case, we extend each chunk to the right (overlapping with the next chunk) as long as there is a partial alignment of cost $\leq k$ that started inside the original chunk. Especially when the text is long (so that $\lceil n/W \rceil \approx n/W$), this intra-sequence parallellism is near-optimal.

2.3. Early break

The complexity of computing the entire DP matrix as described so far is $O(m(\lceil n/W \rceil + \lceil m/w \rceil))$, where the final $+\lceil m/w \rceil$ accounts for overlaps between chunks. In ASM, we only care about matches with a cost at most k, and thus, parts of the DP matrix where values are $> k$ can be skipped [Ukkonen, 1983, 1985b, Myers, 1986], as shown in Figure 3. In particular, the edit distance between two uniform random and equal-length DNA sequences is typically around 45% of their length, so that most of the time, the cost of aligning a prefix of length $2 \cdot k$ of the pattern already incurs a cost $> k$. More formally, Chang and Lampe [1992] proved that the number of states with cost $\leq k$ is $O(kn)$ when searching a random text.

[3] As an implementation detail, we actually search $\mathsf{comp}(P)$ against $\mathsf{rev}(T)$, so that we can avoid taking the complement of T. Invariance under complements is trivial.

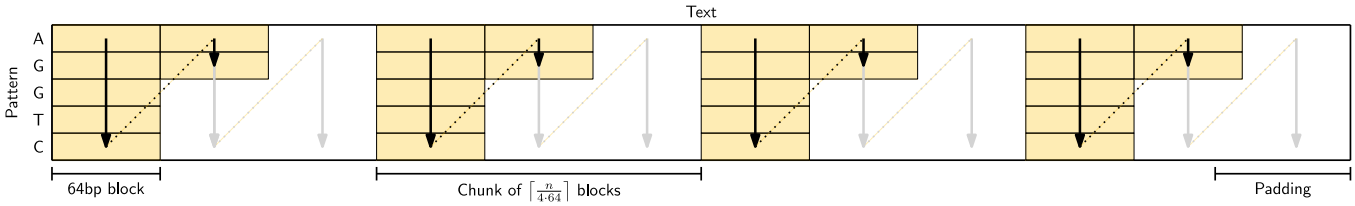

Fig. 1: The tiling strategy used by Sassy. The text is first split into word-size blocks of 64 bases. Then, the list of blocks is split into 4 chunks that are processed in parallel, with one SIMD lane per chunk. The text is implicitly padded as needed. Within a chunk, filling the matrix proceeds block-by-block. For each block, all (up to, see Section 2.3 and Figure 3) $m = |P|$ rows are computed before proceeding to the next block. Each chunk is extended into the succeeding chunk as long as there is a sufficiently good "in progress" alignment (not shown).

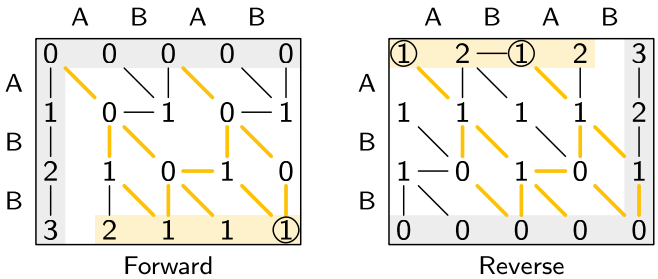

Fig. 2: Approximate string matching. An example of finding all occurrences of ABB in ABAB. On the left, the forward search initializes the top and left of the matrix (shaded in grey). Then, it shows all optimal paths to each state. On the bottom, the final distances are highlighted, and all optimal alignments of cost 1 are highlighted in yellow. By default, Sassy only starts a trace in the circled 1, a rightmost local minimum. The right figure shows the reverse alignment, where the matrix is filled from the bottom right to the top left. Note that the set of optimal alignments is the same, but that the number of local minima (1 vs 2) and global minima (3 vs 2) both differ.

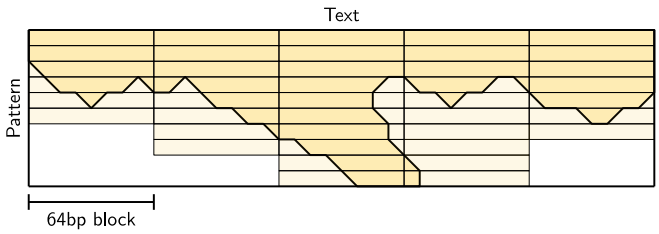

Fig. 3: Early break. We are only interested in entries of the DP matrix with value $\leq k$, as shown in the bold-outlined area. As soon as all entries in a row are $> k$, we can stop processing that block of text, as in the first and second block. Then, we only reach the bottom of the matrix when matches are present, as in the third block. One exception is shown in the fourth block: when there are states at the end of the previous block at distance $\leq k$, we must continue at least one row beyond that point. Since we use SIMD to process 4 chunks in parallel (not shown here), in practice we continue until the values in all 4 lanes are $> k$.

Thus, as soon as all W columns corresponding to a SIMD vector contain a value $> k$, and additionally there are no remaining states with cost $\leq k$ at the end of the preceding blocks, we can stop with the current four blocks and move on to the next.

2.4. Library and Command Line Tool

The main entrypoint of the Sassy Rust library is a function `search(pattern, text, k)` that returns one match for each rightmost local minium endpoint. It can optionally return matches for *all* endpoints, and supports reverse complements. The input can be either (case insensitive) ASCII text, simple ACGT DNA, or more general IUPAC-encoded DNA where sequences (*both* the pattern and the text) may contain bases such as N (matching ACTG), Y (matching CT), and R (matching AG). This is handled by selecting a *profile* (Appendix C.1). In case of simple DNA, we provide a function to validate that no non-ACGT bases are present. We provide both C and Python bindings.

We also provide a simple command line tool for searching a sequence in all records of a Fasta file, that can be installed via `cargo install sassy` or `conda install -c bioconda sassy`. Examples of commands are:

```
sassy search --alphabet dna --no-rc -k 0 --pattern CAT data.fa
sassy search --alphabet iupac -k 1 -f patterns.fa genome.fa.gz
sassy crispr -k 5 --guide guides.fa ref.fa
```

The first searches for an exact match of CAT in all records of `data.fa`. The second searches each record of `patterns.fa` in `genome.fa.gz`, while allowing up to 1 error and also searching the reverse-complement text. The last searches each of the guides in `guides.fa` against `ref.fa` while allowing at most 5 errors. Here, PAM sequences must match exactly and the preceding sgRNA can contain up to 5 errors.

Whereas the library is single-threaded, the command line tool maintains a queue of (P, T) tuples that are distributed (in batches) over all threads. Further details on the implementation can be found in Appendix C.

3. Results

We compare Sassy against Edlib [Šošić and Šikić, 2017] in two metrics: the throughput of searching random DNA sequences without matches (the number of text bases processed per second), and the throughput of finding and tracing matches (the number of matches that can be found and processed per second). Section 4 shows specific applications of Sassy. The code and data for the benchmarks can be found in the `evals` directory at `https://github.com/RagnarGrootKoerkamp/sassy`. These experiments were run on an Intel Core i7-10750H with 6 cores, 12 threads, AVX2 support, and running at a fixed frequency of 3.6 GHz.

Throughput of text searching. In Figure 4 and Figure 5, we compare the text throughput of searching with Sassy and Edlib. Each data point is the average of searching 1000 random DNA pattern of length m in a random text of length n. Figure 4

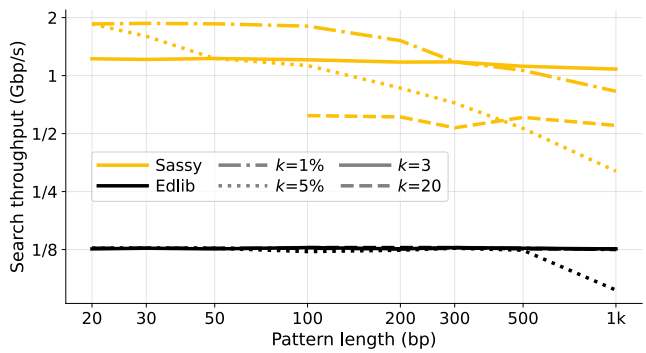

Fig. 4: Throughput of searching patterns of varying length. The pattern length m (x-axis) ranges from 20 to 1000, and the error threshold k (line style) is either fixed at 3 or 20, or computed as $\lceil m/100 \rceil$ or $\lceil m/20 \rceil$. Only points with $m > 3k$ are shown to avoid spurious matches. All points are computed by averaging over 1000 random patterns and texts of length $n = 10^5$, and then converting to throughput. Note that this does not include searching the reverse-complement strand. Sassy has up to $10\times$ higher throughput than Edlib when k is small.

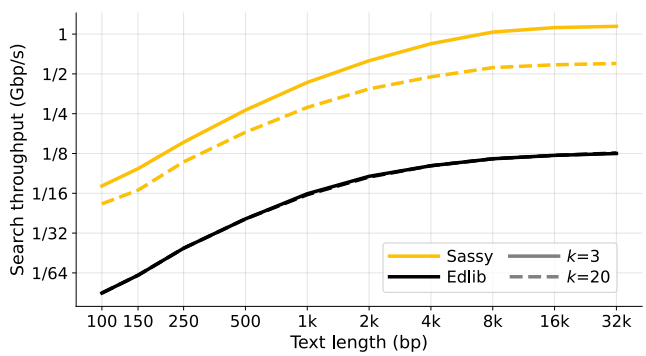

Fig. 5: Throughput of searching texts of varying length. We search a pattern of length $m = 100$ against texts with length varying from $n = 150$ to $n = 128\,000$ bp, with $k \in \{3, 20\}$. All points are computed by averaging over 1000 random texts and then converting to throughput. Note that this does not include searching the reverse-complement strand. While Sassy is consistently faster than Edlib, its relative advantage is smaller for shorter texts.

compares searching patterns of varying length ($20 \leq m \leq 1000$) in a long text ($n = 10^5$) for varying error thresholds ($0 \leq k \leq 50 = 0.05 \cdot 1000$), while in Figure 5, a pattern of length $m = 100$ is searched in texts of length $150 \leq n \leq 128\,000$ for $k \in \{3, 20\}$.

We exclude Seqan, since it is consistently outperformed by Edlib. A comparison against parasail, an affine-cost aligner, can be found in Appendix E. It is around $10\times$ slower than Edlib and $100\times$ slower than Sassy. Ish is an up-to 35% faster re-implementation of parasail, but does not provide Rust bindings. Lastly, Other tools such as agrep do not accept FASTA input.

Sassy is faster across all m, n, and k. For short patterns ($m \leq 50$ bp), Sassy has throughput over 1.2 Gbp/s whereas the throughput of Edlib does not exceed 130 Mbp/s. Since both the pattern and text are random, no matches occur, and the early break causes Sassy to have complexity $O(k\lceil n/W \rceil)$. Indeed, for constant k the throughput is independent of m, while it decreases

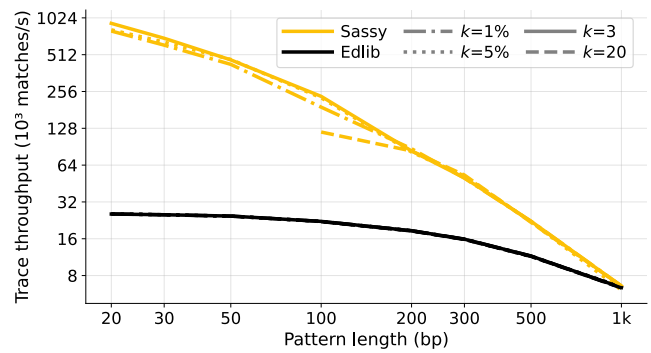

Fig. 6: Throughput of finding matches. When the text contains a match, this causes a larger part of the matrix to be computed since the early break does not trigger. Later, this part of the matrix is recomputed in full and stored in $O(m^2/w)$ words of memory so that a traceback can be done. To measure the total time it takes to process a match, we use the same setup as in fig. 4 with the addition of a single copy of the pattern planted at the end of the text. We measure the time difference with the version without match and report the corresponding throughput.

when $k = 0.05 \cdot m$. Edlib, on the other hand, has throughput nearly independent of k: in nearly all cases we have $k \leq 20$, so that crossing the first $w = 64$ rows of the DP matrix already incurs a cost $> k$. This matches the complexity of $O(\lceil k/w \rceil n)$. In Figure 5, we see that both Edlib and Sassy are faster when searching longer texts and have large constant overhead when searching small texts, in case of Sassy due to relatively large overheads between text chunks. At text length $n = 150$, Sassy is $4\times$ to $6\times$ faster than Edlib, which increases to $5\times$ to $9\times$ speedup for longer texts.

The throughput when searching ASCII (slightly faster) or IUPAC (slightly slower) text is within 5% compared to DNA.

Affine-cost aligners. Sassy is over $10\times$ faster than tools implementing affine-cost alignments. For example, Sassy needs 4.5 seconds to search a pattern of length 23 in a human genome (with up to $k = 4$ errors, excluding searching the reverse-complement). For the same task, parasail [Daily, 2016] in semi-global mode with default cost parameters takes 53 to 69 seconds, depending on the exact configuration (8 or 16 bit values, and SSE4.1 vs AVX2), reporting up to 1.45 GCUPS (giga cell updates per second). Ish [Stadick, 2025] with default parameters takes 69 seconds (SSE4.1) to 110 seconds (AVX2). Running parasail with the same costs as Ish takes 81 seconds to 198 seconds. These methods are slower both because they store larger values (instead of bitpacking), and because they compute two additional *affine layers* of the DP matrix.

We propose that fast edit distance alignments could be used to identify candidate regions which can then be refined with affine-cost methods.

Throughput of tracing. In Figure 6, we show the throughput of finding matches. This includes the time to locally compute all rows of the matrix (rather than just the top $O(k)$ rows), the time to recompute the matrix region containing the match, and the time for tracing through the filled matrix. We use the same setup as in previous experiments, and "plant" a single match at the end of the text. We then compared the run time of the same text with and without match.

Placing the match at the end avoids triggering the dynamic reduction of k in Edlib: Since Edlib performs semi-global alignment

and only searches matches with the minimum edit distance, it reduces k whenever it finds a match with cost $< k$. If the match were placed earlier in the text, this reduction would confound the measurement by timing both the tracing time and its k reduction strategy. By placing the match at the end of the text, we largely isolate the tracing cost from the k reduction strategy.

For short patterns, Sassy is over 10× faster per match than Edlib. For longer patterns, Sassy's throughput goes down quadratically as it naively computes the $O(m^2/w)$ words to fill the part of the matrix where a match is. Edlib does not slow down as much, likely due to $O(ns)$ banding, but is nevertheless still slower than Sassy for patterns up to length $m \leq 1000$.

4. Application: Finding CRISPR off-targets

Finding short sequences has many important applications, with CRISPR off-target searching being of particular recent interest [Musunuru et al., 2025]. SWOffinder [Yaish et al., 2024] and CHOPOFF [Labun et al., 2025] are currently among the fastest and most accurate tools for identifying off-target sequences. We extended Sassy with CRISPR off-target searching, enabling the search for PAM motifs—such as the Cas9 NGG motif—preceded by a guide RNA (gRNA) sequence [Ran et al., 2013].

Unlike SWOffinder, `sassy crispr` reports at most one match for every location where the PAM has an exact match.

4.1. Searching 61 guide RNAs in the human genome

First, we briefly summarize the algorithms used by the benchmarked tools. SWOffinder uses Smith-Waterman alignment to fill the entire $m \times n$ matrix, identifying all end positions with $\leq k$ edits. It then filters these alignments to only those with at most 1 indel [Yaish et al., 2024]. CHOPOFF takes a different approach by first indexing all PAM locations in the target genome and storing their prefixes i.e. the sgRNA flank. Then they precompute all the edit distance paths, within k, through these prefixes. This allows instant lookup of sgRNA sequences with at most k edits of previously identified PAM sites. These indexes range from 3.2Gbp ($k = 0$) to 4.5Gbp ($k = 4$) for the human genome. The prefix edit paths are shipped with CHOPOFF for $k \leq 4$. However, computing this for $k = 5$ did not finish in 10 hours. Sassy's algorithm is similar to its regular search, but with an additional filter prior to traceback: when the user requests the PAM sequence to be unmutated, then the traceback is only performed when the exact PAM is present.

To compare Sassy to the other off-target search tools we used the benchmark from the CHOPOFF paper [Labun et al., 2025]. This searches 61 guide sequences with the NGG PAM against the human genome. Experiments were run on a Intel(R) Xeon(R) Gold 6240, using 16 threads for each job. Results are in Table 1.

For large values of k, Sassy outperforms both competitors by a wide margin. In fact, for $k \geq 4$, Sassy is more than 100× faster than SWOffinder and over 4× faster than the index-based CHOPOFF. Notably, Sassy completes the $k = 5$ search in just 44s, whereas CHOPOFF's index building for $k = 5$ exceeded 10hours (and was therefore omitted). For smaller values ($k \leq 3$), Sassy and CHOPOFF have comparable performance, with Sassy trailing by only a few seconds. We do note that CHOPOFF is faster when there are substantially more sgRNA patterns, as most time is spent on loading the index, which is not parallellized over multiple threads.

	Sassy		CHOPOFF		SWOffinder	
k	Time	Matches	Time	Matches	Time	Matches
0	0:19	62	0:18 (19:47)	62	40:21	62
1	0:20	127	0:19 (23:53)	127	42:07	155
2	0:23	1,284	0:19 (23:59)	1,284	39:54	1,411
3	0:28	32,033	0:26 (23:39)	32,033	40:34	35,434
4	0:30	405,401	2:23 (23:56)	405,401	41:38	471,395
5	0:44	4,093,387	–	–	40:43	1,476,640

Table 1. Time (mm:ss) to search 61 sgRNAs in Human genome CHM13 using 16 threads, and number of matches. For CHOPOFF the time to build the index is shown in parentheses, and was terminated after 10 hours for $k = 5$. All tools require exact matches of the 3 bp PAM sequence. SWOffinder reports more matches for small k because it searches with the PAM sequence at the *front* of the pattern, sometimes resulting in multiple matches for each match of the PAM. For $k = 5$, it has *fewer* matches, because it only allows each match to have up to 1 indel.

We note that full support for IUPAC bases, as Sassy and CHOPOFF have, is important for this application, since human genome assemblies may not always be fully resolved—see Appendix D.

Thus, Sassy is an extremely fast tool that does not require building an index, making it ideal for personalized, reference-free, off-target screening.

5. Discussion

Sassy solves approximate string matching, and allows fast searching for short DNA sequences without using an index. The main algorithmic novelty is to use *horizontal* bitpacking of deltas, and intra-sequence parallellism using SIMD (Figure 1), leading to a complexity of $O(k \cdot \lceil n/W \rceil)$ when searching random text. This improved complexity allows searching text at nearly 2 Gbp/s, and up to 15× speedup over Edlib.

Practically speaking, Sassy is a simple-to-use tool with many applications. Since Sassy is index-free, it easily supports IUPAC characters in both the pattern and text. It is significantly faster than other index-free methods for searching CRISPR off-target matches, and is being integrated into other tools such as CRISPRapido[4], which uses Sassy as a pre-filter for off-target detection with a more fine-grained (affine-cost) cost model, and Barbell [Beeloo et al., 2025], a demultiplexer for Nanopore reads.

It is left to the user to choose a suitable edit distance threshold k that captures all biologically relevant matches, and to post-process and/or refine the matches with a more accurate affine or position-specific cost model if needed.

Limitations and future work. When text that is searched is short ($n \leq 1000$ or so, and in particular for $n = 150$), Sassy fails to reach its maximum throughput (Figure 5), because the overhead of initializing each search is relatively large and there is a lot of overlap between adjacent text chunks. This is particularly relevant when searching barcodes in reads, and motivates ongoing work on Sassy2, which searches batches of multiple patterns at once.

Sassy is primarily designed to search patterns with length $m \leq 100$ or so, and includes a quadratic $O(m^2/w)$ component in both the initial filling of the matrix (Figure 3) and the traceback,

[4] `https://github.com/pinellolab/crisprapido`

whereas a banded $O(mk/w)$ approach would be sufficient in theory.

Lastly, Sassy only provides limited benefit when a *static* text is searched with many (at least 100 to 1000) patterns: in that case, search schemes for approximate string matching [Renders et al., 2022, 2024, Gottlieb and Reinert, 2025a] that use a bidirectional index, such as Columba [Depuydt et al., 2024, Renders et al., 2025], should be preferred instead.

6. Acknowledgments

We thank Erik Garrison for encouraging us to build Sassy, and Rob Patro for feedback on this paper. We thank Seth Stadick for help with the evaluation of parasail and Ish.

RB is financed by ZonMw [541003001]. RGK is financed by ETH Research Grant ETH-17 21-1 to Gunnar Rätsch.

Conflict of Interest: none declared.

References

K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, Dec. 1987. ISSN 1095-7111. doi: 10.1137/0216067. URL http://dx.doi.org/10.1137/0216067.

A. Abramova, A. Karkman, and J. Bengtsson-Palme. Metagenomic assemblies tend to break around antibiotic resistance genes. *BMC genomics*, 25(1):959, 2024.

S. F. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of mathematical biology*, 48: 603–616, 1986.

S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990. ISSN 0022-2836. doi: 10.1016/s0022-2836(05)80360-2. URL http://dx.doi.org/10.1016/S0022-2836(05)80360-2.

A. Backurs and P. Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 51–58, 2015.

S. Bae, J. Park, and J.-S. Kim. Cas-OFFinder: a fast and versatile algorithm that searches for potential off-target sites of Cas9 RNA-guided endonucleases. *Bioinformatics*, 30(10):1473–1475, 2014.

R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, Oct. 1992. ISSN 1557-7317. doi: 10.1145/135239.135243. URL http://dx.doi.org/10.1145/135239.135243.

R. Baeza-Yates and G. Navarro. *Multiple approximate string matching*, page 174–184. Springer Berlin Heidelberg, 1997. ISBN 9783540694229. doi: 10.1007/3-540-63307-3_57. URL http://dx.doi.org/10.1007/3-540-63307-3_57.

R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, Feb. 1999. ISSN 0178-4617. doi: 10.1007/pl00009253. URL http://dx.doi.org/10.1007/PL00009253.

B. Banović Đeri, S. Nešić, I. Vićić, J. Samardžić, and D. Nikolić. Benchmarking of five NGS mapping tools for the reference alignment of bacterial outer membrane vesicles-associated small RNAs. *Frontiers in Microbiology*, 15, July 2024. ISSN 1664-302X. doi: 10.3389/fmicb.2024.1401985. URL http://dx.doi.org/10.3389/fmicb.2024.1401985.

R. Beeloo, R. Groot Koerkamp, X. Jia, M. J. Broekhuizen-Stins, L. van IJken, E. M. Broens, A. Zomer, and B. E. Dutilh. Barbell resolves demultiplexing and trimming issues in nanopore data. Oct. 2025. doi: 10.1101/2025.10.22.683865. URL http://dx.doi.org/10.1101/2025.10.22.683865.

P. Bille. Faster approximate string matching for short patterns. *Theory of Computing Systems*, 50(3):492–515, Apr. 2011. ISSN 1433-0490. doi: 10.1007/s00224-011-9322-y. URL http://dx.doi.org/10.1007/s00224-011-9322-y.

H. S. Bilofsky, C. Burks, J. W. Fickett, W. B. Goad, F. I. Lewitter, W. P. Rindone, C. Swindell, and C.-S. Tung. The GenBank genetic sequence databank. *Nucleic Acids Research*, 14(1):1–4, 1986. ISSN 1362-4962. doi: 10.1093/nar/14.1.1. URL http://dx.doi.org/10.1093/nar/14.1.1.

T. Bingmann, P. Bradley, F. Gauger, and Z. Iqbal. COBS: a compact bit-sliced signature index. In *String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings 26*, pages 285–303. Springer, 2019.

R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, Oct. 1977. ISSN 1557-7317. doi: 10.1145/359842.359859. URL http://dx.doi.org/10.1145/359842.359859.

J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, May 2001. ISSN 1367-4803. doi: 10.1093/bioinformatics/17.5.419. URL http://dx.doi.org/10.1093/bioinformatics/17.5.419.

S. Cancellieri, M. C. Canver, N. Bombieri, R. Giugno, and L. Pinello. CRISPRitz: rapid, high-throughput and variant-aware in silico off-target site identification for CRISPR genome editing. *Bioinformatics*, 36(7):2001–2008, 2020.

W. I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. *Lecture Notes in Computer Science*, page 175–184, 1992. ISSN 1611-3349. doi: 10.1007/3-540-56024-6_14.

H. G. Chaudhari, J. Penterman, H. J. Whitton, S. J. Spencer, N. Flanagan, M. C. Lei Zhang, E. Huang, A. S. Khedkar, J. M. Toomey, C. A. Shearer, A. W. Needham, T. W. Ho, J. D. Kulman, T. Cradick, and A. Kernytsky. Evaluation of homology-independent CRISPR-Cas9 off-target assessment methods. *The CRISPR Journal*, 3(6):440–453, Dec. 2020. ISSN 2573-1602. doi: 10.1089/crispr.2020.0053. URL http://dx.doi.org/10.1089/crispr.2020.0053.

O. Cheng, M. H. Ling, C. Wang, S. Wu, M. E. Ritchie, J. Göke, N. Amin, and N. M. Davidson. Flexiplex: a versatile demultiplexer and search tool for omics data. *Bioinformatics*, 40(3), Feb. 2024. ISSN 1367-4811. doi: 10.1093/bioinformatics/btae102. URL http://dx.doi.org/10.1093/bioinformatics/btae102.

T. Chhabra, S. S. Ghuman, and J. Tarhio. String searching with mismatches using AVX2 and AVX-512 instructions. *Information Processing Letters*, 189:106557, Mar. 2025. ISSN 0020-0190. doi: 10.1016/j.ipl.2025.106557. URL http://dx.doi.org/10.1016/j.ipl.2025.106557.

R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6): 1761–1782, 2002.

J. Daily. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC bioinformatics*, 17: 1–11, 2016.

F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

L. Depuydt, L. Renders, S. Van de Vyver, L. Veys, T. Gagie, and J. Fostier. b-move: Faster bidirectional character extensions in a run-length compressed index. volume 312, pages 10:1–10:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi: 10.4230/LIPICS.WABI.2024.10. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.WABI.2024.10.

M. Doblas, O. Lostes-Cazorla, Q. Aguado-Puig, C. Iñiguez, M. Moreto, and S. Marco-Sola. QuickEd: high-performance exact sequence alignment based on bound-and-align. *Bioinformatics*, 41(3), Mar. 2025. ISSN 1367-4811. doi: 10.1093/bioinformatics/btaf112. URL http://dx.doi.org/10.1093/bioinformatics/btaf112.

E. Espinosa, R. Quislant, R. Larrosa, and O. Plata. SeqMatcher: efficient genome sequence matching with AVX-512 extensions. *The Journal of Supercomputing*, 81(1), Dec. 2024. ISSN 1573-0484. doi: 10.1007/s11227-024-06789-0. URL http://dx.doi.org/10.1007/s11227-024-06789-0.

M. Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, Nov. 2006. ISSN 1367-4803. doi: 10.1093/bioinformatics/btl582. URL http://dx.doi.org/10.1093/bioinformatics/btl582.

F. J. Fiori, W. Pakalén, and J. Tarhio. Approximate string matching with SIMD. *The Computer Journal*, 65(6):1472–1488, Feb. 2021. ISSN 1460-2067. doi: 10.1093/comjnl/bxaa193. URL http://dx.doi.org/10.1093/comjnl/bxaa193.

Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1): 33–72, Mar. 1988. ISSN 0885-064X. doi: 10.1016/0885-064x(88)90008-8. URL http://dx.doi.org/10.1016/0885-064X(88)90008-8.

O. Gotoh. An improved algorithm for matching biological sequences. *Journal of molecular biology*, 162(3):705–708, 1982.

O. Gotoh. Multiple sequence alignment: Algorithms and applications. *Advances in Biophysics*, 36:159–206, 1999. ISSN 0065-227X. doi: 10.1016/s0065-227x(99)80007-0. URL http://dx.doi.org/10.1016/s0065-227x(99)80007-0.

S. Gottlieb and K. Reinert. Search schemes for approximate string matching. *NAR Genomics and Bioinformatics*, 7(1), Jan. 2025a. ISSN 2631-9268. doi: 10.1093/nargab/lqaf025. URL http://dx.doi.org/10.1093/nargab/lqaf025.

S. G. Gottlieb and K. Reinert. SeArcH schemes for Approximate stRing mAtching. *NAR Genomics and Bioinformatics*, 7(1), Jan. 2025b. ISSN 2631-9268. doi: 10.1093/nargab/lqaf025. URL http://dx.doi.org/10.1093/nargab/lqaf025.

R. Groot Koerkamp. A*PA2: up to 20 times faster exact global alignment. Mar. 2024. doi: 10.1101/2024.03.24.586481. URL http://dx.doi.org/10.1101/2024.03.24.586481.

R. Groot Koerkamp and P. Ivanov. Exact global alignment using A* with chaining seed heuristic and match pruning. *Bioinformatics*, 40(3), Jan. 2024. ISSN 1367-4811. doi: 10.1093/bioinformatics/btae032. URL http://dx.doi.org/10.1093/bioinformatics/btae032.

R. Groot Koerkamp and I. Martayan. SimdMinimizers: Computing random minimizers, fast. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi: 10.4230/LIPICS.SEA.2025.20.

P. A. V. Hall and G. R. Dowling. Approximate string matching. *ACM Comput. Surv.*, 12(4):381–402, Dec. 1980. ISSN 0360-0300. doi: 10.1145/356827.356830. URL https://doi.org/10.1145/356827.356830.

R. W. Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

T. Ho, S.-R. Oh, and H. Kim. New algorithms for fixed-length approximate string matching and approximate circular string matching under the Hamming distance. *The Journal of Supercomputing*, 74(5):1815–1834, Nov. 2017. ISSN 1573-0484. doi: 10.1007/s11227-017-2192-6. URL http://dx.doi.org/10.1007/s11227-017-2192-6.

T. N. D. Huynh, W.-K. Hon, T.-W. Lam, and W.-K. Sung. *Approximate String Matching Using Compressed Suffix Arrays*, page 434–444. Springer Berlin Heidelberg, 2004. ISBN 9783540278016. doi: 10.1007/978-3-540-27801-6_33. URL http://dx.doi.org/10.1007/978-3-540-27801-6_33.

C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. *The Max-Shift Algorithm for Approximate String Matching*, page 13–25. Springer Berlin Heidelberg, 2001. ISBN 9783540446880. doi: 10.1007/3-540-44688-5_2. URL http://dx.doi.org/10.1007/3-540-44688-5_2.

D. Jaganathan, K. Ramasamy, G. Sellamuthu, S. Jayabalan, and G. Venkataraman. CRISPR for crop improvement: an update review. *Frontiers in plant science*, 9:985, 2018.

C. Jain, A. Rhie, H. Zhang, C. Chu, B. P. Walenz, S. Koren, and A. M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinformatics*, 36:i111–i118, July 2020. ISSN 1367-4811. doi: 10.1093/bioinformatics/btaa435. URL http://dx.doi.org/10.1093/bioinformatics/btaa435.

P. Jokinen and E. Ukkonen. *Two algorithms for approxmate string matching in static texts*, page 240–248. Springer Berlin Heidelberg, 1991. ISBN 9783540475798. doi: 10.1007/3-540-54345-7_67. URL http://dx.doi.org/10.1007/3-540-54345-7_67.

J. Kärkkäinen, G. Navarro, and E. Ukkonen. *Approximate String Matching over Ziv—Lempel Compressed Text*, page 195–209. Springer Berlin Heidelberg, 2000. ISBN 9783540451235. doi: 10.1007/3-540-45123-4_18. URL http://dx.doi.org/10.1007/3-540-45123-4_18.

D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi: 10.1137/0206024.

P. Koehn and J. Senellart. Fast approximate string matching with suffix arrays and A* parsing. In *Proceedings of the 9th Conference of the Association for Machine Translation in the Americas: Research Papers*, 2010.

E. V. Koonin and K. S. Makarova. Origins and evolution of CRISPR-Cas systems. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 374(1772):20180087, Mar. 2019. ISSN 1471-2970. doi: 10.1098/rstb.2018.0087. URL http://dx.doi.org/10.1098/rstb.2018.0087.

K. Labun, O. Rio, H. Tjeldnes, M. Swirski, A. Z. Komisarczuk, E. Haapaniemi, and E. Valen. CHOPOFF: symbolic alignments enable fast and sensitive CRISPR off-target detection. *bioRxiv*, 2025(01), 2025.

G. M. Landau and U. Vishkin. Efficient string matching in the presence of errors. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, page 126–136. IEEE, 1985. doi: 10.1109/sfcs.1985.22. URL http://dx.doi.org/10.1109/SFCS.1985.22.

G. M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing - STOC '86*, STOC '86, page 220–230. ACM Press, 1986. doi: 10.1145/12130.12152. URL http://dx.doi.org/10.1145/12130.12152.

H. Ledford. Precise CRISPR tool could tackle host of genetic diseases. *Nature*, 574:464–465, 2019.

V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

H. Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. ISSN 1367-4811. doi: 10.1093/bioinformatics/bty191. URL http://dx.doi.org/10.1093/bioinformatics/bty191.

X. Li, K. Chen, and M. Shao. Efficient seeding for error-prone sequences with SubseqHash2. June 2024. doi: 10.1101/2024.05.30.596711. URL http://dx.doi.org/10.1101/2024.05.30.596711.

D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, Mar. 1985. ISSN 1095-9203. doi: 10.1126/science.2983426. URL http://dx.doi.org/10.1126/science.2983426.

D. Liu and M. Steinegger. Block Aligner: fast and flexible pairwise sequence alignment with SIMD-accelerated adaptive blocks. Nov. 2021. doi: 10.1101/2021.11.08.467651. URL http://dx.doi.org/10.1101/2021.11.08.467651.

B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, Mar. 2002. ISSN 1367-4803. doi: 10.1093/bioinformatics/18.3.440. URL http://dx.doi.org/10.1093/bioinformatics/18.3.440.

Mäkinen, Ukkonen, and Navarro. Approximate matching of run-length compressed strings. *Algorithmica*, 35(4):347–369, Apr. 2003. ISSN 1432-0541. doi: 10.1007/s00453-002-1005-2. URL http://dx.doi.org/10.1007/s00453-002-1005-2.

U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, Oct. 1993. ISSN 1095-7111. doi: 10.1137/0222058. URL http://dx.doi.org/10.1137/0222058.

S. Marco-Sola, J. C. Moure, M. Moreto, and A. Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, 2021.

S. Marco-Sola, J. M. Eizenga, A. Guarracino, B. Paten, E. Garrison, and M. Moreto. Optimal gap-affine alignment in O(s) space. *Bioinformatics*, 39(2):btad074, 2023.

S. McCauley. Approximate similarity search under edit distance using locality-sensitive hashing. ICDT '21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi: 10.4230/LIPICS.ICDT.2021.21.

F. J. M. Mojica, C. Díez-Villaseñor, J. García-Martínez, and C. Almendros. Short motif sequences determine the targets of the prokaryotic CRISPR defence system. *Microbiology*, 155(3):733–740, Mar. 2009. ISSN 1465-2080. doi: 10.1099/mic.0.023960-0. URL http://dx.doi.org/10.1099/mic.0.023960-0.

K. Musunuru, S. A. Grandinette, X. Wang, T. R. Hudson, K. Briseno, A. M. Berry, J. L. Hacker, A. Hsu, R. A. Silverstein, L. T. Hille, et al. Patient-specific in vivo gene editing to treat a rare genetic disease. *New England Journal of Medicine*, 392(22):2235–2243, 2025.

R. Muth and U. Manber. *Approximate multiple string search*, page 75–86. Springer Berlin Heidelberg, 1996. ISBN 9783540683902. doi: 10.1007/3-540-61258-0_7. URL http://dx.doi.org/10.1007/3-540-61258-0_7.

E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(1–4):251–266, Nov. 1986. ISSN 1432-0541. doi: 10.1007/bf01840446. URL http://dx.doi.org/10.1007/BF01840446.

G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.

G. Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72(1–2):65–70, Oct. 1999. ISSN 0020-0190. doi: 10.1016/s0020-0190(99)00121-0. URL http://dx.doi.org/10.1016/S0020-0190(99)00121-0.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: A fast search method for large DNA databases. *Genome Research*, 11(10):1725–1729, Oct. 2001. ISSN 1549-5469. doi: 10.1101/gr.194201. URL http://dx.doi.org/10.1101/gr.194201.

S. Nurk, S. Koren, A. Rhie, M. Rautiainen, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, Apr. 2022. ISSN 1095-9203. doi: 10.1126/science.abj6987. URL http://dx.doi.org/10.1126/science.abj6987.

M. Odell and R. Russell. RC 1918-1922. *US patent*, (1,261,167), 1918.

A. Oliva, R. Tobler, A. Cooper, B. Llamas, and Y. Souilmi. Systematic benchmark of ancient DNA read mapping. *Briefings in Bioinformatics*, 22(5), Apr. 2021. ISSN 1477-4054. doi: 10.1093/bib/bbab076. URL http://dx.doi.org/10.1093/bib/bbab076.

O. Owolabi and D. R. McGregor. Fast approximate string matching. *Software: Practice and Experience*, 18(4):387–393, Apr. 1988. ISSN 1097-024X. doi: 10.1002/spe.4380180407. URL http://dx.doi.org/10.1002/spe.4380180407.

F. A. Ran, P. D. Hsu, J. Wright, V. Agarwala, D. A. Scott, and F. Zhang. Genome engineering using the CRISPR-Cas9 system. *Nature protocols*, 8(11):2281–2308, 2013.

K. Reinert, T. H. Dadi, M. Ehrhardt, H. Hauswedell, S. Mehringer, R. Rahn, J. Kim, C. Pockrandt, J. Winkler, E. Siragusa, G. Urgese, and D. Weese. The SeqAn C++ template library for efficient sequence analysis: A resource for programmers. *Journal of Biotechnology*, 261:157–168, Nov. 2017. ISSN 0168-1656. doi: 10.1016/j.jbiotec.2017.07.017. URL http://dx.doi.org/10.1016/j.jbiotec.2017.07.017.

L. Renders, L. Depuydt, and J. Fostier. *Approximate Pattern Matching Using Search Schemes and In-Text Verification*, page 419–435. Springer International Publishing, 2022. ISBN 9783031078026. doi: 10.1007/978-3-031-07802-6_36. URL http://dx.doi.org/10.1007/978-3-031-07802-6_36.

L. Renders, L. Depuydt, S. Rahmann, and J. Fostier. *Automated Design of Efficient Search Schemes for Lossless Approximate Pattern Matching*, page 164–184. Springer Nature Switzerland, 2024. ISBN 9781071639894. doi: 10.1007/978-1-0716-3989-4_11. URL http://dx.doi.org/10.1007/978-1-0716-3989-4_11.

L. Renders, L. Depuydt, T. Gagie, and J. Fostier. Columba: fast approximate pattern matching with optimized search schemes. *Bioinformatics*, 41(12), Dec. 2025. ISSN 1367-4811. doi:

10.1093/bioinformatics/btaf652. URL http://dx.doi.org/10.1093/bioinformatics/btaf652.

T. Rognes and E. Seeberg. Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, Aug 2000. ISSN 1367-4803. doi: 10.1093/bioinformatics/16.8.699.

S. Roux, U. Neri, B. Bushnell, B. Fremin, N. A. George, U. Gophna, L. A. Hug, A. P. Camargo, D. Wu, N. Ivanova, N. Kyrpides, and E. Eloe-Fadrosh. Planetary-scale metagenomic search reveals new patterns of CRISPR targeting. June 2025. doi: 10.1101/2025.06.12.659409. URL http://dx.doi.org/10.1101/2025.06.12.659409.

S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: Accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):e1000386, May 2009. ISSN 1553-7358. doi: 10.1371/journal.pcbi.1000386. URL http://dx.doi.org/10.1371/journal.pcbi.1000386.

L. Salmela, J. Tarhio, and P. Kalsi. Approximate Boyer-Moore string matching for small alphabets. *Algorithmica*, 58(3):591–609, Feb. 2009. ISSN 1432-0541. doi: 10.1007/s00453-009-9286-3. URL http://dx.doi.org/10.1007/s00453-009-9286-3.

D. A. Scott and F. Zhang. Implications of human genetic variation in CRISPR-based therapeutic genome editing. *Nature Medicine*, 23(9):1095–1101, 2017. ISSN 1546-170X. doi: 10.1038/nm.4377. URL https://doi.org/10.1038/nm.4377.

P. H. Sellers. Pattern recognition in genetic sequences. *Proceedings of the National Academy of Sciences*, 76(7):3041–3041, July 1979. ISSN 1091-6490. doi: 10.1073/pnas.76.7.3041. URL http://dx.doi.org/10.1073/pnas.76.7.3041.

P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1 (4):359–373, Dec. 1980. ISSN 0196-6774. doi: 10.1016/0196-6774(80)90016-4. URL http://dx.doi.org/10.1016/0196-6774(80)90016-4.

R. S. Shapiro, A. Chavez, and J. J. Collins. CRISPR-based genomic tools for the manipulation of genetically intractable microorganisms. *Nature Reviews Microbiology*, 16(6):333–339, 2018.

M. Šošić and M. Šikić. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9): 1394–1395, 2017.

S. Stadick. Ish: SIMD and GPU accelerated local and semi-global alignment as a CLI filtering tool. June 2025. doi: 10.1101/2025.06.04.657890. URL http://dx.doi.org/10.1101/2025.06.04.657890.

E. Sutinen and J. Tarhio. *On using q-gram locations in approximate string matching*, page 327–340. Springer Berlin Heidelberg, 1995. ISBN 9783540449133. doi: 10.1007/3-540-60313-1_153. URL http://dx.doi.org/10.1007/3-540-60313-1_153.

H. Suzuki and M. Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19(S1), Feb. 2018. ISSN 1471-2105. doi: 10.1186/s12859-018-2014-8. URL http://dx.doi.org/10.1186/s12859-018-2014-8.

I. Tolstoganov, M. Martin, and K. Sahlin. Multi-context seeds enable fast and high-accuracy read mapping. Nov. 2024. doi: 10.1101/2024.10.29.620855. URL http://dx.doi.org/10.1101/2024.10.29.620855.

G. Tonkin-Hill, N. MacAlasdair, C. Ruis, A. Weimann, G. Horesh, J. A. Lees, R. A. Gladstone, S. Lo, C. Beaudoin, R. A. Floto, S. D. Frost, J. Corander, S. D. Bentley, and J. Parkhill. Producing polished prokaryotic pangenomes with the Panaroo pipeline. *Genome Biology*, 21(1), July 2020. ISSN 1474-760X. doi: 10.1186/s13059-020-02090-4. URL http://dx.doi.org/10.1186/s13059-020-02090-4.

E. Ukkonen. On approximate string matching. In *International Conference on Fundamentals of Computation Theory*, pages 487–495. Springer, 1983. doi: 10.1007/3-540-12689-9_129.

E. Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985a.

E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132–137, Mar 1985b. ISSN 0196-6774. doi: 10.1016/0196-6774(85)90023-9.

E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1): 191–211, Jan. 1992. ISSN 0304-3975. doi: 10.1016/0304-3975(92)90143-4. URL http://dx.doi.org/10.1016/0304-3975(92)90143-4.

E. Ukkonen. *Approximate string-matching over suffix trees*, page 228–242. CPM '93. Springer-Verlag, 1993. ISBN 354056764X. doi: 10.1007/bfb0029808. URL http://dx.doi.org/10.1007/BFb0029808.

R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan. 1974. ISSN 1557-735X. doi: 10.1145/321796.321811. URL http://dx.doi.org/10.1145/321796.321811.

S. Walia, C. Ye, A. Bera, D. Lodhavia, and Y. Turakhia. Talco: Tiling genome sequence alignment using convergence of traceback pointers. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Mar. 2024. doi: 10.1109/hpca57654.2024.00044.

D. Weese, M. Holtgrewe, and K. Reinert. RazerS 3: Faster, fully sensitive read mapping. *Bioinformatics*, 28(20):2592–2599, Aug. 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts505. URL http://dx.doi.org/10.1093/bioinformatics/bts505.

W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences*, 80(3):726–730, Feb. 1983. ISSN 1091-6490. doi: 10.1073/pnas.80.3.726. URL http://dx.doi.org/10.1073/pnas.80.3.726.

A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, 13(2):145–150, 1997. ISSN 1460-2059. doi: 10.1093/bioinformatics/13.2.145. URL http://dx.doi.org/10.1093/bioinformatics/13.2.145.

A. H. Wright. Approximate string matching using withinword parallelism. *Software: Practice and Experience*, 24(4):337–362, Apr. 1994. ISSN 1097-024X. doi: 10.1002/spe.4380240402. URL http://dx.doi.org/10.1002/spe.4380240402.

S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, Oct. 1992. ISSN 1557-7317. doi: 10.1145/135239.135244. URL http://dx.doi.org/10.1145/135239.135244.

T. D. Wu and C. K. Watanabe. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics*, 21(9):1859–1875, Feb. 2005. ISSN 1460-2059. doi: 10.1093/bioinformatics/bti310. URL http://dx.doi.org/10.1093/bioinformatics/bti310.

O. Yaish, A. Malle, E. Cohen, and Y. Orenstein. SWOffinder: Efficient and versatile search of CRISPR off-targets with

bulges by Smith-Waterman alignment. *iScience*, 27(1):108557, 2024. ISSN 2589-0042. doi: https://doi.org/10.1016/j.isci.2023.108557. URL `https://www.sciencedirect.com/science/article/pii/S2589004223026342`.

B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, page 545–556. IEEE, 2010. doi: 10.1109/icde.2010.5447836. URL `http://dx.doi.org/10.1109/ICDE.2010.5447836`.

J. Zhang, H. Lan, Y. Chan, Y. Shang, B. Schmidt, and W. Liu. BGSA: a bit-parallel global sequence alignment toolkit for multi-core and many-core architectures. *Bioinformatics*, 35(13):2306–2308, 2019.

A. Pseudocode

Alg. 1: Pseudocode for Sassy's main Search function, that takes as input the pattern P of length m and text T of length n, and returns a list of end-positions in the text where alignments of cost $\leq k$ end. Some of the operations operate on SIMD registers such as $[0, 0, 0, 0]$. ComputeBlock applies Myers' bitpacking algorithm to SIMD registers. Details of Eq, AllAboveK and FindEndPositionsAtMostK can be found in Appendix C.

```
1: function Search(P, T, k)
2:   B ← ⌈n/(4 · 64)⌉                    ▷ Number of 64bp blocks per chunk.
3:   j_≤k ← 0   ▷ Last row in previous block ending in a value ≤ k.
4:   j_max ← 0                          ▷ Last computed row in previous block.
5:   v⁺ ← [[1, 1, 1, 1]; m]             ▷ Initialize list of SIMD of +1 deltas.
6:   v⁻ ← [[0, 0, 0, 0]; m]             ▷ Modified when α < 1 for overhang.
7:   M ← [[], [], [], []]                ▷ Matches (pos, cost) per lane.
8:   for i ∈ {0, ..., B + ⌈(m + k)/64⌉ − 1} do
9:     i_max ← i
10:    h⁺, h⁻ ← [0, 0, 0, 0]            ▷ Deltas in current row.
11:    d_s, d_e ← [0, 0, 0, 0]          ▷ Dist to start and end of each block.
12:    j'_≤k, j'_max ← 0
13:    for j ∈ {0, ..., m − 1} do
14:      d_s ← d_s + v⁺[j] − v⁻[j]
15:      for ℓ ∈ {0, 1, 2, 3} do
16:        ▷ Profile determines a bitmask indicating equal chars. ◁
17:        eq[ℓ] ← Eq(P[j], T[B · ℓ + 64 · i ... B · ℓ + 64 · i + 64])
18:      ComputeBlock(v⁺[j], v⁻[j], h⁺, h⁻, eq)  ▷ Myers bitpacking.
19:      d_e ← d_e + v⁺[j] − v⁻[j]
20:      if min_ℓ d_e[ℓ] ≤ k then
21:        j'_≤k ← j
22:      j'_max ← j
23:      ▷ Check if all 256 values are > k.                          ◁
24:      if j > j_≤k and AllAboveK(d_s, h⁺, h⁻, k) then
25:        for j' ∈ {j + 1, ..., j_max} do
26:          v⁺[j'] ← [1, 1, 1, 1]       ▷ Reset "dirty" skipped rows.
27:          v⁻[j'] ← [0, 0, 0, 0]
28:        j_max ← j'_max
29:        j_≤k ← j'_≤k
30:        if i ≥ B and 64 · (i − B) − j > k then
31:          Goto 35, break out of i.    ▷ Done with chunk overlap.
32:        Goto 8, continue with next i                ▷ Early break.
33:    ▷ Find (local minima) end positions with cost ≤ k.              ◁
34:    FindEndPositionsAtMostK(M, d_s, h⁺, h⁻, k)
35:  for ℓ ∈ {1, 2, 3} do          ▷ Prune duplicate matches in overlap.
36:    M[ℓ] ← {x ∈ M[ℓ] : x ≥ B · ℓ + 64 · (i_max + 1)}
37:  return M
```

B. Overhang alignments

In some applications, it is useful to not only find occurrences of the pattern that are fully contained in the text, but also those that extend beyond the text. For example, this is the case for reads that may only partially contain a barcode, or in case of fragmented assemblies [Abramova et al., 2024]. Following Abrahamson [1987], we call these *overhanging* matches.

Definition 2 (Overhanging match) Given pattern P of length m and text T of length n:

1. there is a *left-overhanging* match when a suffix $P[l \ldots m]$ matches a prefix of T,
2. there is a *right-overhanging* match when a prefix $P[0 \ldots m-l]$ matches a suffix of T.

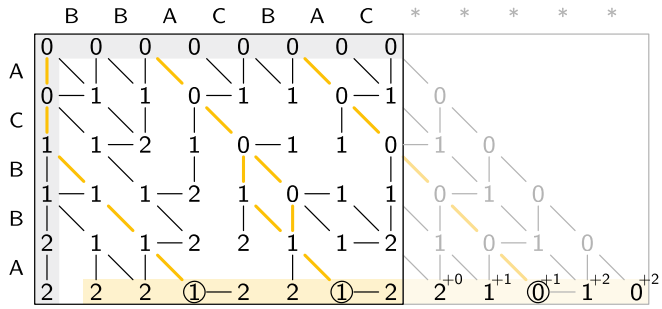

Fig. 7: Overhang alignment of ACBBA against BBACBAC, with overhang cost $\alpha = 0.5$. On the left, the state in row j is initialized with cost $\lfloor j\alpha \rfloor$, and a left-overhanging alignment (highlighted) is found where BBA matches and AC extends beyond the text for a cost of $\lfloor 2/2 \rfloor = 1$. On the right, the text is padded with m wildcard symbols, so that the costs on the right side of the matrix are replicated in the bottom row. Then, for each extended state in columns $i > n$, $\lfloor (i-n)\alpha \rfloor$ is added to the cost. This finds a right-overhanging alignment where AC matches and BBA extends beyond the text for a cost of $\lfloor 3/2 \rfloor = 1$.

In either case, each of the l overhanging (unmatched) characters of P incurs a cost of $0 \leq \alpha \leq 1$, for a total *overhang cost* of $\lfloor l \cdot \alpha \rfloor$.

Figure 7 shows an example with $\alpha = 0.5$ (Sassy's default) with three matches of cost 1: a left-overhanging match, a non-overhanging match, and a right-overhanging match. When $\alpha = 1$, this corresponds to semi-global alignment and ASM, whereas $\alpha = 0$ corresponds to *overlap* alignments.

Sassy only needs a slight modification to find overlapping matches: the left of the DP matrix is initialized with cost $\lfloor i\alpha \rfloor$ instead of i, and on the right we extend the text with m "wildcard" symbols, so that the costs in the right column of the matrix are "copied" to the bottom row. Then, we manually add the overhang cost for those extended states. In practice, we use the IUPAC alphabet for this, and simply append N characters.

C. Implementation Notes

We now provide some more details on the functions used in Algorithm 1.

AllAboveK(d_s, h^+, h^-, k). This function takes as input the distance d_s to the start of 4 blocks, a threshold k, and the bit-encoded horizontal differences in each block. It then checks if the represented values in all blocks are $> k$. In practice, this function is slightly slow, and we call it not every row (as shown in the pseudocode), but only at most every 8 rows.

It processes each lane ℓ independently. First, we *pack* the bits of $h^+[\ell]$ and $h^-[\ell]$ into p (using `_pext_u64` available in BMI2) to remove positions where the delta is 0 and both have an indicator bit of 0. Then, a 1 in p indicates -1 and a 0 indicates $+1$ (with $+1$ padding at the end). We split the 64-bit value p into bytes, and do a precomputed table lookup that gives the total delta across the byte, and the minimum prefix sum in the byte. The we do a rolling sum over the bytes to compute the minimum overall prefix.

FindEndPositionsAtMostK(M, d_s, h^+, h^-, k). This function again works lane-by-lane. Unlike AllAboveK, this function is only called at most once per block of text, when the iteration over j reaches the end of the pattern. Thus, it is less performance critical, and we implement it by simply iterating through the 64 bits of the

bitmasks and keeping a rolling sum for the current score in each column. Then, each time a value $\leq k$ is seen, the index of the text and the corresponding cost are pushed to the list of matches.

C.1. Profiles

The job of a *profile* is to take a single character $P[j]$ of the pattern and a slice of 64 text characters, and determine a bitmask $\mathsf{Eq}(P[j], T[x \ldots y])$ indicating which of the text characters equal $P[j]$ [Rognes and Seeberg, 2000].

We recommend having a look at the code, in the `src/profiles` directory of the git repository. Currently we only support AVX2, and thus, this is subject to change.

ASCII. For the ASCII profile, we implement this as follows. For each block of text, we precompute a 256 long array of 64-bit words, so that the mask for each ASCII (In fact, ASCII characters are < 128, but this way we can handle any raw data.) character of the pattern can simply be looked up. The array is filled by using 256-bit SIMD instructions to compare each byte up to 256 to both the first 32 characters (`[u8; 32]` is 256 bits) and last 32 characters of the text slice, and merging the two 32-bit values.

For efficiency, we first compute a list of all distinct bytes in the pattern, and then only fill table rows corresponding to those bytes.

Case-insensitive ASCII. In this case, we first lowercase all text and pattern characters before doing the equality check. This is done by xor'ing the value of all uppercase bytes by 32.

DNA. DNA only has 4 characters, and so we precompute a table of size 4. Each ACTG character is encoded into an integer in $\{0, 1, 2, 3\}$ by first shifting right 1 bit and then only looking at the bottom 2 bits.

Optionally, it can first be checked that the text only contains valid bases in ACTG. This is done by ensuring that each position case-insensitively equals one of ACTG.

IUPAC. Here, we start by building a table that maps each IUPAC character to a 4-bit mask indicating which subset of ACTG it matches. Since only letters are allowed input, it is sufficient to only consider the low 5 bits of each input character leaving 32 possible values. This automatically collapses upper and lower case values. We would now like to use a `[u8; 32]` SIMD register as a lookup table (via shuffle instructions), but unfortunately cross-128-bit lane byte shuffles are not supported on AVX2. We work around this: each byte only contains 4 bits of data, and thus, we can merge them, so that byte i in a `[u8; 16]` contains the 4 bits of both i (low half) and $i+16$ (high half). Then, we can use this as a lookup table on the low 4 bits of each text character, and use the 5th bit to select either the low or high half of the returned byte.

From here, we proceed similar to before: we first build a list of characters occurring in the profile. Then we encode each character to its 4-bit representation, and find the text characters that this "intersects" with.

D. Support for ambiguous bases

Depending on their quality, human genome assemblies can contain over 10% ambiguous bases, as seen in GRCh38 [Nurk et al., 2022]. In search applications with clinical implications, such as CRISPR off-target analysis, it is crucial to report matches in regions containing ambiguous bases (e.g., N), as these indicate sequence uncertainty and may harbor unintended cut sites. To evaluate tool performance in such scenarios, we searched for the sgRNA GGAAGACACACTGGCAGAAANGG with $k = 0$ against a mock sequence where the sgRNA base at position 14 (C) was replaced by N in one version of a text, and by Y in another. Sassy implements the IUPAC profile for CRISPR off-target searches and returned all matches according to IUPAC base pairing. CHOPOFF requires a user to specificy the max number of ambiguous bases (we used `-ambig-max=23`) and did also return all matches. SWOffinder does not have a command line option but a hardcoded boolean flag (default is `false`) which we set to `true` and recompiled. It did find the N version but not the Y version. Therefore, both Sassy and CHOPOFF have IUPAC support, and SWOffinder only supports N with source code modification. This result underscores the importance of selecting tools that correctly handle ambiguous bases in clinically relevant analyses.

E. Comparison with parasail

We compared Sassy to parasail, a SIMD-based affine-cost aligner [Daily, 2016], and to Edlib. To minimize overhead, we used the Rust bindings at `https://github.com/nsbuitrago/parasail-rs` to call parasail as a library, consistent with our setup for Sassy and Edlib.

Because affine scoring is not directly comparable to edit distance, we approximated edit distance costs by setting `-gap-open=1`, `-gap-extension=1`, `match-score=0`, and `mismatch-score=-1`. Given the small range of scores under this configuration, we used 8-bit output (`solution_width=8`).

parasail supports diagonal, striped, and prefix-scan vectorization. We used prefix-scan as it was the fastest for increasing pattern and text lengths.

As shown in Figure 8, when searching a text of $n = 100\,000$ bp with varying pattern lengths, Sassy achieves approximately $10\times$ higher throughput than Edlib and $100\times$ higher throughput than parasail. Similar trends hold when varying the text length (Figure 9).

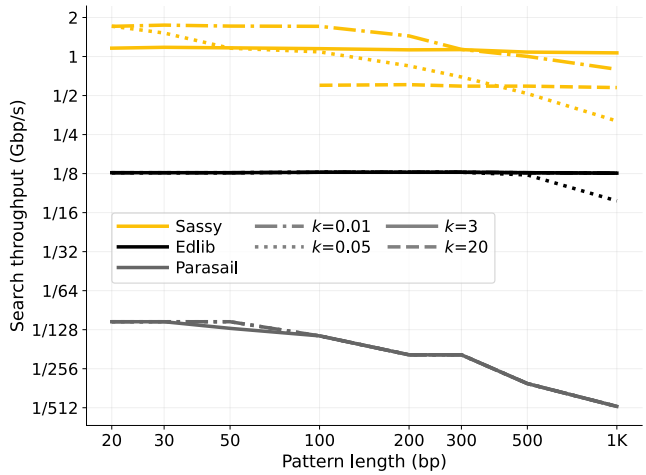

Fig. 8: Throughput of searching patterns of varying length. The pattern length m (x-axis) ranges from 20 to 1000, and the error threshold k (line style) is either fixed at 3 or 20, or computed as $\lceil m/100 \rceil$ or $\lceil m/20 \rceil$. Only points with $m > 3k$ are shown to avoid spurious matches. All points are computed by averaging over 1000 random patterns and texts of length $n = 10^5$, and then converting to throughput. Note that this does not include searching the reverse-complement strand. Sassy achieves up to $10\times$ higher throughput than Edlib for small k, and roughly two to three orders of magnitude higher throughput than parasail. The performance gap with parasail increases with pattern length, exceeding $500\times$ at $m = 1000$.

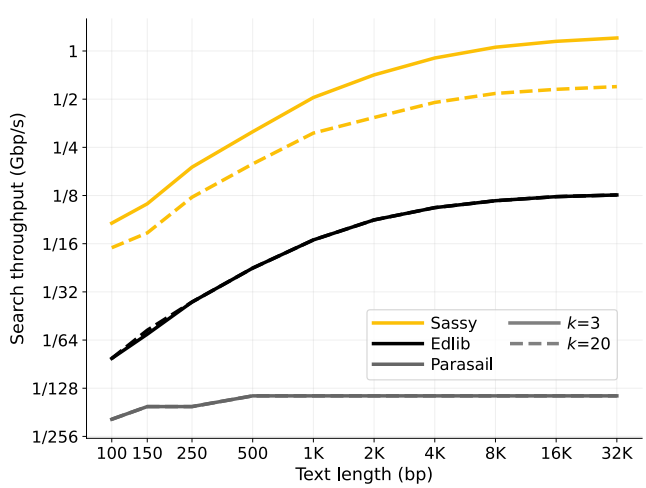

Fig. 9: Throughput of searching texts of varying length. We search a pattern of length $m = 100$ against texts with length varying from $n = 150$ to $n = 128\ 000$ bp, with $k \in \{3, 20\}$. All points are computed by averaging over 1000 random texts and then converting to throughput. Note that this does not include searching the reverse-complement strand. Sassy consistently outperforms Edlib by about one order of magnitude, while parasail remains roughly two to three orders of magnitude slower but less sensitive to text length.