

# Sassy: fuzzy searching DNA sequences using SIMD

Rick Beeloo<sup>1,\*</sup>,†,  and Ragnar Groot Koerkamp<sup>2,3,\*</sup>,†, 

<sup>1</sup>Department of Theoretical Biology and Bioinformatics, Utrecht University, Utrecht, 3584CH, Netherlands

<sup>2</sup>Department of Computer Science, ETH Zurich, Zurich, 8092, Switzerland

<sup>3</sup>Department of Computer Science, Karlsruhe Institute of Technology, Karlsruhe, 76131, Germany

\*Corresponding authors. Rick Beeloo, Department of Theoretical Biology and Bioinformatics, Utrecht University, Utrecht, 3584CH, Netherlands.

E-mail: beelooinformatics@gmail.com; Ragnar Groot Koerkamp, Department of Computer Science, ETH Zurich, Zurich, 8092, Switzerland.

E-mail: ragnar.grootkoerkamp@inf.ethz.ch.

†Equal contribution.

Associate Editor: Inanc Birol

## 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.

**Results:** We introduce Sassy, a library and tool for ASM of short patterns in long texts. Sassy splits the text into four parts that are searched in parallel, and uses bitvectors in the text direction rather than the pattern direction. This has complexity  $O(k\lceil n/W \rceil)$  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.

Sassy is  $4\times$  to  $15\times$  faster than Edlib for patterns  $\leq 1000$ bp, 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 min. Sassy also scales well to larger  $k$ , unlike CHOPOFF whose index took over 10 h to build for  $k=5$ .

**Availability and implementation:** Sassy is available as library and binary at <https://github.com/RagnarGrootKoerkamp/sassy>, and archived at [swh:1:dir:e884758dce5777a441bc2799dc8824e563c5f97b](https://swh.1.dir:e884758dce5777a441bc2799dc8824e563c5f97b).

## 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 article, 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 ASM. Sassy targets patterns with length up to around a 2000 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.

Received: 11 August 2025. Revised: 10 March 2026. Accepted: 22 April 2026

© The Author(s) 2026. Published by Oxford University Press.

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

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 (Fig. 2).
- 3) We optimize ASM for searching through long texts. Algorithmically this has two small novelties: (i) bitpacking in the text direction, rather than the pattern direction, and (ii) intra-sequence parallelism by splitting the text into four 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.
- 4) In Appendix B, available as [supplementary data](#) at *Bioinformatics* online, 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 bitpacking algorithm of Myers (1999). For word size  $w = 64$  (compared to SIMD ‘width’  $W = 256$ ), 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.

### 1.2.1 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  $O(n^{2-\epsilon})$  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 bitpacking, pioneered by 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 bitpacking 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 (Wozniak 1997, Farrar 2007, Liu and Steinegger 2023), allowing parallel processing such as in KSW2 (Li 2018, Suzuki and Kasahara 2018), BSAAlign (Zhang et al. 2019), and SeqMatcher (Espinosa et al. 2025). 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 2023, 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 retain the guarantee that an optimal alignment is found (Groot Koerkamp 2024, Groot Koerkamp and Ivanov 2024).

### 1.2.2 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* 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). 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. 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.

### 1.2.3 Cost models

A *cost model* defines what constitutes an *error* and the cost associated with 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), which 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.

### 1.2.4 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 preprocess 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.* 2022, Gottlieb and Reinert 2025).

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 of different cost models in the 1980s, algorithms were created to detect inexact matches with  $k$  errors. Initially, ASM 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  $O(m^2 + k^2n)$  (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 2012), fixed pattern lengths (Iliopoulos *et al.* 2001, Ho *et al.* 2018), 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 preprocessing 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.* 2010), or search schemes on top of a bidirectional FM-index or move-index (Renders *et al.* 2022, 2024, Depuydt *et al.* 2024, Gottlieb and Reinert 2025, Renders *et al.*

2025). Such methods thus implement completely different algorithms than the streaming-based method that we focus on in this article, and are suitable for different applications.

### 1.2.5 SIMD parallelism

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.* 2025), where AVX-512 instructions are used to effectively use 512-bit integers. In contrast, Sassy splits the text into four chunks that are processed in parallel, somewhat similar to Farrar's striped method (Farrar 2007) 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.

### 1.2.6 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 favor of speed. Thus, we consider mapping to be approximate ASM. (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, the number of seeds also 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.* 2026), or by hashing subsequences instead of substrings (e.g. SubseqHash2) (Li *et al.* 2025).

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

### 1.2.7 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)—e.g. 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 15 May 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 developed. 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 article, 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  $\Sigma$  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  $\text{rev}(T) := t_{n-1} \dots t_1 t_0$  for the reverse of  $T$ , and for DNA sequences, we define the *complement*  $\text{comp}(T)$  as the sequence where each base is replaced by its complement ( $A \leftrightarrow T$  and  $C \leftrightarrow G$ , extended to IUPAC as well). The *reverse complement* is then  $\text{rc}(T) := \text{rev}(\text{comp}(T))$ .

### 2.1 Approximate string matching

We define AMS 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,  $\text{search}(P, T, k)$ , 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 from  $j-k$  to  $j+k$  will have a cost  $\leq k$  (see Fig. 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 cannot be shrunk or 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  $\text{search}(P, T, k)$  are in one-to-one correspondence with those reported by  $\text{search}(\text{rev}(P), \text{rev}(T), k)$ , since the number of (local/global minima) end positions  $\leq k$  can differ in the forward and reverse case, as exemplified in Fig. 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  $\text{rc}(T)$ , so that both searches are in the natural direction of the pattern and searching  $P$  in  $\text{rc}(T)$  and  $\text{rc}(\text{rc}(T)) = T$  gives the same result. (As an implementation detail, we actually search  $\text{comp}(P)$  against  $\text{rev}(T)$ , so that we can avoid taking the complement of  $T$ . Invariance under complements is trivial.) Searching  $\text{rc}(P)$  in  $T$  or  $\text{rc}(T)$  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 to 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. This way, SIMD lanes process different parts of the text, ensuring speedups even for short patterns. Consequently, the bitpacking is also in the text direction to allow for efficient *profile* lookups (Appendix C, available as supplementary data at Bioinformatics online). Pseudocode of the main search function of *Sassy* can be found in Appendix A, available as supplementary data at Bioinformatics online.

**SIMD.** We use 256 bit SIMD widths using either AVX2 or two parallel NEON registers to compute four lanes of 64 bit words in parallel. (Using AVX-512 did not provide further gains since iterating eight text chunks in parallel became a bottleneck.) We avoid dependencies between SIMD lanes by splitting the text into four 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 2007), 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 parallelism 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 Fig. 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.

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

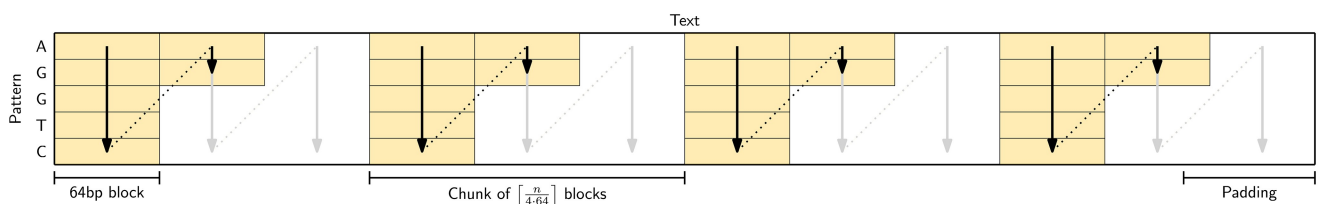
The main entrypoint of the *Sassy* Rust library is a function `search(pattern, text, k)` that returns one match for each rightmost local minimum endpoint. It can optionally return matches for *all* endpoints, and supports reverse complements. The input can be either (case insensitive) ASCII text, simple 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, available as supplementary data at Bioinformatics online). 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:

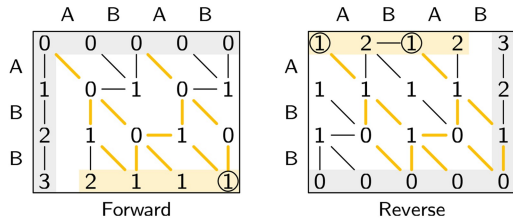
```
sassy search --alphabet dna --no-rc -k 0 --pattern
CAT data.fa
sassy search --alphabet iupac -k 1 -f patterns.
fa genome.fa.gz
sassy crispr -k 5 --guide guides.fa ref.fa
```

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 five errors. Here, PAM sequences must match exactly and the preceding sgRNA can contain up to five errors.

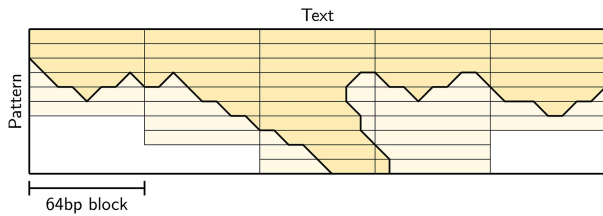
Whereas the library is single-threaded, the command line tool maintains a queue of  $(P, T)$  tuples that are distributed (in batches) over all threads. Further details on the implementation can be found in Appendix C, available as supplementary data at Bioinformatics online.



**Figure 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 four 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 Fig. 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).



**Figure 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 versus 2) and global minima (3 versus 2) both differ.



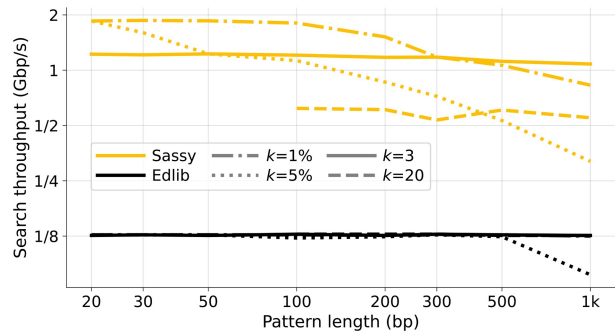
**Figure 3** Early break. We are only interested in entries of the DP matrix with value  $\leq k$ , as shown in the bold-outlined area. As soon as all entries in a row are  $> k$ , we can stop processing that block of text, as in the first and second block. Then, we only reach the bottom of the matrix 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 four chunks in parallel (not shown here), in practice we continue until the values in all four lanes are  $> k$ .

## 3 Results

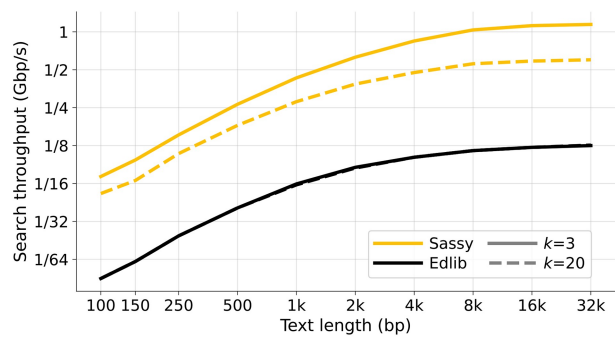
We compare `Sassy` against `Edlib` (Šošić and Šikić 2017) (which uses bitpacking but not SIMD) in two metrics: the throughput of searching random DNA sequences without matches (the number of text bases processed per second), and the throughput of finding and tracing matches (the number of matches that 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.

### 3.1 Throughput of text searching

In [Figs 4 and 5](#), we compare the text throughput of searching with `Sassy` and `Edlib`. Each data point is the average of searching 1000 random DNA patterns of length  $m$  in a random text of length  $n$ . [Figure 4](#) compares searching patterns of varying length ( $20 \leq m \leq 1000$ ) in a long text ( $n = 10^5$ ) for varying error



**Figure 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.



**Figure 5** Throughput of searching texts of varying length. We search a pattern of length  $m = 100$  against texts with length varying from  $n = 100$  to  $n = 32000$  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.

thresholds ( $0 \leq k \leq 50 = 0.05 \cdot 1000$ ), while in [Fig. 5](#), a pattern of length  $m = 100$  is searched in texts of length  $100 \leq n \leq 32000$  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](#), available as [supplementary data](#) at *Bioinformatics* online. 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 1.2Gbp/s whereas the throughput of `Edlib` does not exceed 130Mbp/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 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 Fig. 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  $5\times$   $9\times$  speedup for longer texts.

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

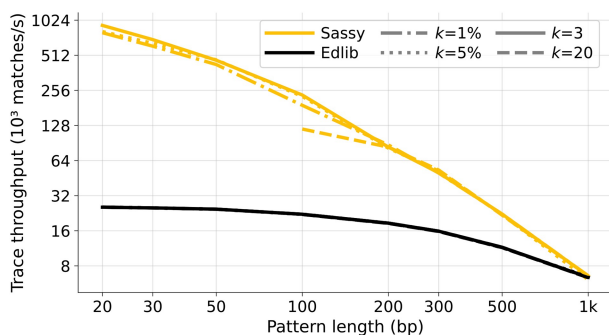
### 3.2 Affine-cost aligners

`Sassy` is over  $10\times$  faster than tools implementing affine-cost alignments. For example, `Sassy` needs 4.5s 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 53–69s, depending on the exact configuration (8 or 16bit values, and SSE4.1 versus AVX2), reporting up to 1.45GCUPS (giga cell updates per second). `Ish` (Stadick 2025) with default parameters takes 69s (SSE4.1) to 110s (AVX2). Running `parasail` with the same costs as `Ish` takes 81–198s. 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 exact affine-cost alignment could be achieved by using a fast edit distance alignment to identify candidate regions.

### 3.3 Throughput of tracing

In Fig. 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



**Figure 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 as the number of matches that can be processed every second.

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 compare 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 match with the minimum edit distance, it reduces  $k$  whenever it finds a match with cost  $< k$ . If the match was 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  $10\times$  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 one 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 10h. `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 an Intel(R) Xeon(R) Gold 6240, using 16 threads for each job. Results are in Table 1.

**Table 1** Time (mm:ss) to search 61 sgRNAs in Human genome CHM13 using 16 threads, and number of matches.<sup>a</sup>

k	Sassy		CHOPOFF		SWOfffinder	
	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	1284	0:19 (23:59)	1284	39:54	1411
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

<sup>a</sup> For CHOPOFF the time to build the index is shown in parentheses, and was terminated after 10 h for  $k = 5$ . All tools require exact matches of the 3bp PAM sequence. SWOfffinder 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 one indel.

For large values of  $k$ , Sassy outperforms both competitors by a wide margin. In fact, for  $k \geq 4$ , Sassy is more than  $100 \times$  faster than SWOfffinder and over  $4 \times$  faster than the index-based CHOPOFF. Notably, Sassy completes the  $k = 5$  search in just 44 s, whereas CHOPOFF's index building for  $k = 5$  exceeded 10 h (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 parallelized over multiple threads.

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](#), available as [supplementary data](#) at *Bioinformatics* online.

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 ASM, 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 parallelism using SIMD ([Fig. 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  $15 \times$  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 (<https://github.com/pinellolab/crisprapido>), 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.

## 5.1 Limitations and future work

When the text that is searched is short ( $n \leq 1000$  or so, and in particular for  $n = 150$ ), Sassy fails to reach its maximum throughput ([Fig. 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 ([Beeloo and Groot Koerkamp 2026](#)), 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 ([Fig. 3](#)) and the traceback, 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 100–1000) patterns: in that case, search schemes for ASM ([Renders et al. 2022, 2024, Gottlieb and Reinert 2025](#)) that use a bidirectional index, such as Columba ([Depuydt et al. 2024, Renders et al. 2025](#)), should be preferred instead.

## Acknowledgements

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

## Author contributions

Rick Beeloo (Conceptualization [equal], Data curation [equal], Formal analysis [equal], Funding acquisition [equal], Investigation [equal], Methodology [equal], Project administration [equal], Resources [equal], Software [equal], Supervision [equal], Validation [equal], Visualization [equal], Writing—original draft [equal], Writing—review & editing [equal]), and Ragnar Groot Koerkamp (Conceptualization [equal], Data curation [equal], Formal analysis [equal], Funding acquisition [equal], Investigation [equal], Methodology [equal], Project administration [equal], Resources [equal], Software [equal], Supervision [equal], Validation [equal], Visualization [equal], Writing—original draft [equal], Writing—review & editing [equal])

## Supplementary material

[Supplementary material](#) is available at *Bioinformatics* online.

## Conflict of interests

None declared.

## Funding

R.B. was financed by ZonMw [541003001]. R.G.K. was financed by ETH Research Grant ETH-17 21–1 to Gunnar Rätsch.

## Data availability

The code and data for this article are available on Github at <https://github.com/ragnarGrootKoerkamp/sassy>.

## References

- Altschul SF, Erickson BW. Optimal sequence alignment using affine gap costs. *Bull Math Biol* 1986;**48**:603–16.
- Altschul SF, Gish W, Miller W *et al.* Basic local alignment search tool. *J Mol Biol* 1990;**215**:403–10. [https://doi.org/10.1016/S0022-2836\(05\)80360-2](https://doi.org/10.1016/S0022-2836(05)80360-2)
- Backurs A, Indyk P. 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*. New York, NY, United States: Association for Computing Machinery, 2015, 51–58.
- Bae S, Park J, Kim J-S. Cas-OFFinder: a fast and versatile algorithm that searches for potential off-target sites of Cas9 RNA-guided endonucleases. *Bioinformatics* 2014;**30**:1473–5.
- Baeza-Yates R, Gonnet GH. A new approach to text searching. *Commun ACM* 1992;**35**:74–82. <https://doi.org/10.1145/135239.135243>
- Baeza-Yates R, Navarro G. Multiple approximate string matching. In: *Algorithms and Data Structures*. Berlin, Heidelberg, Germany: Springer, 1997, 174–184. [https://doi.org/10.1007/3-540-63307-3\\_57](https://doi.org/10.1007/3-540-63307-3_57)
- Baeza-Yates R, Navarro G. Faster approximate string matching. *Algorithmica* 1999;**23**:127–58. <https://doi.org/10.1007/pl00009253>
- Banović Deri B, Nešić S, Vikić I *et al.* Benchmarking of five NGS mapping tools for the reference alignment of bacterial outer membrane vesicles-associated small RNAs. *Front Microbiol* 2024;**15**. <https://doi.org/10.3389/fmicb.2024.1401985>
- Beeloo R, Groot Koerkamp R. Sassy2: batch searching of short DNA patterns. bioRxiv, <https://doi.org/10.64898/2026.03.10.710811>, March 2026, preprint: not peer reviewed.
- Beeloo R, Groot Koerkamp R, Jia X *et al.* Barbell resolves demultiplexing and trimming issues in Nanopore data. bioRxiv. <https://doi.org/10.1101/2025.10.22.683865>, October 2025, preprint: not peer reviewed.
- Bille P. Faster approximate string matching for short patterns. *Theory Comput Syst* 2012;**50**:492–515. <https://doi.org/10.1007/s00224-011-9322-y>
- Bilofsky HS, Burks C, Fickett JW *et al.* The GenBank genetic sequence databank. *Nucleic Acids Res* 1986;**14**:1–4. <https://doi.org/10.1093/nar/14.1.1>
- Bingmann T, Bradley P, Gauger F *et al.* COBS: a compact bit-sliced signature index. In: *Proceedings 26, String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019*. Springer, 2019, 285–303.
- Boyer RS, Moore JS. A fast string searching algorithm. *Commun ACM* 1977;**20**:762–72. <https://doi.org/10.1145/359842.359859>
- Buhler J. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics* 2001;**17**:419–28. <https://doi.org/10.1093/bioinformatics/17.5.419>
- Cancellieri S, Canver MC, Bombieri N *et al.* CRISPRitz: rapid, high-throughput and variant-aware in silico off-target site identification for CRISPR genome editing. *Bioinformatics* 2020;**36**:2001–8. <https://doi.org/10.1093/bioinformatics/btz867>
- Chang WI, Lampe J. Theoretical and empirical comparisons of approximate string matching algorithms. In: *Lecture Notes in Computer Science*. 1992, 175–84. [https://doi.org/10.1007/3-540-56024-6\\_14](https://doi.org/10.1007/3-540-56024-6_14)
- Chaudhari HG, Penterman J, Whitton HJ *et al.* Evaluation of homology-independent CRISPR-Cas9 off-target assessment methods. *CRISPR J* 2020;**3**:440–53. <https://doi.org/10.1089/crispr.2020.0053>
- Cheng O, Ling MH, Wang C *et al.* Flexiplex: a versatile demultiplexer and search tool for omics data. *Bioinformatics* 2024;**40**:btae102. <https://doi.org/10.1093/bioinformatics/btae102>
- Chhabra T, Ghuman SS, Tarhio J. String searching with mismatches using AVX2 and AVX-512 instructions. *Inf Process Lett* 2025;**189**:106557. <https://doi.org/10.1016/j.ipl.2025.106557>
- Cole R, Hariharan R. Approximate string matching: a simpler faster algorithm. *SIAM J Comput* 2002;**31**:1761–82.
- Daily J. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics* 2016;**17**:81.
- Damerau FJ. A technique for computer detection and correction of spelling errors. *Commun ACM* 1964;**7**:171–6.
- Depuydt L, Renders L, Van de Vyver S *et al.* b-Move: Faster Bidirectional Character Extensions in a Run-Length Compressed Index. Vol. **312**. Wadern, Saarland, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 10: 1–18. <https://doi.org/10.4230/LIPICS.WABI.2024.10>
- Doblas M, Lostes-Cazorla O, Aguado-Puig Q *et al.* QuickEd: high-performance exact sequence alignment based on bound-and-align. *Bioinformatics* 2025;**41**:btaf112. <https://doi.org/10.1093/bioinformatics/btaf112>
- Espinosa E, Quisilant R, Larrosa R *et al.* SeqMatcher: efficient genome sequence matching with AVX-512 extensions. *J Supercomput* 2025;**35**:5–79. <https://doi.org/10.1007/s11227-024-06789-0>
- Farrar M. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics* 2007;**23**:156–61. <https://doi.org/10.1093/bioinformatics/btl582>
- Fiori FJ, Pakalén W, Tarhio J. Approximate string matching with SIMD. *Comput J* 2022;**65**:1472–88. <https://doi.org/10.1093/comjnl/bxaa193>
- Galil Z, Giancarlo R. Data structures and algorithms for approximate string matching. *J Complex* 1988;**4**:33–72. [https://doi.org/10.1016/0885-064x\(88\)90008-8](https://doi.org/10.1016/0885-064x(88)90008-8)
- Gotoh O. An improved algorithm for matching biological sequences. *J Mol Biol* 1982;**162**:705–8.
- Gotoh O. Multiple sequence alignment: algorithms and applications. *Adv Biophys* 1999;**36**:159–206. [https://doi.org/10.1016/S0065-227x\(99\)80007-0](https://doi.org/10.1016/S0065-227x(99)80007-0)
- Gottlieb S, Reinert K. Search schemes for approximate string matching. *NAR Genom Bioinform* 2025;**7**:lqaf025. <https://doi.org/10.1093/nargab/lqaf025>
- Groot Koerkamp R. A\*PA2: Up to 19× faster exact global alignment WABI 2024. Volume 312. WABI of lipics. 312 of Lipics. Of Lipics. 312 of Lipics. Schloss Dagstuhl – Leibniz-Zentrum für Informatik Wadern Germany, 2024, 17: Wadern Schloss Dagstuhl – Leibniz-Zentrum für Informatik Wadern Germany, Wadern, Saarland, Germany, 171–171. <https://doi.org/10.4230/LIPICS.WABI.2024.17>

- Groot Koerkamp R, Ivanov P. Exact global alignment using A\* with chaining seed heuristic and match pruning. *Bioinformatics* 2024;**40**:btac032. <https://doi.org/10.1093/bioinformatics/btae032>
- Groot Koerkamp R, Martayan I. *SimdMinimizers: Computing Random Minimizers, Fast*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. <https://doi.org/10.4230/LIPICS.SEA.2025.20>
- Hall PAV, Dowling GR. Approximate string matching. *ACM Comput Surv* 1980;**12**:381–402. <https://doi.org/10.1145/356827.356830>
- Hamming RW. Error detecting and error correcting codes. *Bell Syst Tech J* 1950;**29**:147–60.
- Ho T, Oh S-R, Kim H. New algorithms for fixed-length approximate string matching and approximate circular string matching under the Hamming distance. *J Supercomput* 2018;**74**:1815–34. <https://doi.org/10.1007/s11227-017-2192-6>
- Huynh TND, Hon W-K, Lam T-W *et al*. Approximate string matching using compressed suffix arrays. In: *Combinatorial Pattern Matching*. Berlin and Heidelberg, Germany: Springer, 2004, 434–44. [https://doi.org/10.1007/978-3-540-27801-6\\_33](https://doi.org/10.1007/978-3-540-27801-6_33)
- Iliopoulos CS, Mouchard L, Pinzon YJ. The Max-Shift algorithm for approximate string matching. In: *Algorithm Engineering*. Springer Berlin Heidelberg, 2001, 13–25. [https://doi.org/10.1007/3-540-44688-5\\_2](https://doi.org/10.1007/3-540-44688-5_2)
- Jaganathan D, Ramasamy K, Sellamuthu G *et al*. CRISPR for crop improvement: an update review. *Front Plant Sci* 2018;**9**:985.
- Jain C, Rhie A, Zhang H *et al*. Weighted minimizer sampling improves long read mapping. *Bioinformatics* 2020;**36**:i111–8. <https://doi.org/10.1093/bioinformatics/btaa435>
- Jokinen P, Ukkonen E. Two algorithms for approximate string matching in static texts. In: *Mathematical Foundations of Computer Science 1991*. Berlin and Heidelberg, Germany: Springer, 1991, 240–8. [https://doi.org/10.1007/3-540-54345-7\\_67](https://doi.org/10.1007/3-540-54345-7_67)
- Kärkkäinen J, Navarro G, Ukkonen E. Approximate string matching over Ziv–Lempel compressed text. In: *Combinatorial Pattern Matching*. Springer Berlin Heidelberg, 2000, 195–209. [https://doi.org/10.1007/3-540-45123-4\\_18](https://doi.org/10.1007/3-540-45123-4_18)
- Knuth DE, Morris JH Jr, Pratt VR. Fast pattern matching in strings. *SIAM J Comput* 1977;**6**:323–50. <https://doi.org/10.1137/0206024>
- Koehn P, Senellart J. 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*. Denver, Colorado, USA, 2010.
- Koonin EV, Makarova KS. Origins and evolution of CRISPR-Cas systems. *Philos Trans R Soc Lond B Biol Sci* 2019;**374**:20180087. <https://doi.org/10.1098/rstb.2018.0087>
- Labun K, Rio O, Tjeldnes H *et al*. CHOPOFF: symbolic alignments enable fast and sensitive CRISPR off-target detection. bioRxiv, <https://doi.org/10.1101/2025.01.06.603201>, January 2025, preprint: not peer reviewed.
- Landau GM, Vishkin U. Efficient string matching in the presence of errors. In: *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*. Portland, Oregon, USA: IEEE, 1985, 126–36. <https://doi.org/10.1109/sfcs.1985.22>
- Landau GM, Vishkin U. 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*. Portland, Oregon, USA: ACM Press, 1986, 220–30. <https://doi.org/10.1145/12130.12152>
- Ledford H. Precise CRISPR tool could tackle host of genetic diseases. *Nature* 2019;**574**:464–5.
- Levenshtein VI *et al*. Binary codes capable of correcting deletions, insertions, and reversals. In: *Soviet Physics Doklady*, Vol. 10. Soviet Union, 1966, 707–10.
- Li H. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 2018;**34**:3094–100. <https://doi.org/10.1093/bioinformatics/bty191>
- Li X, Chen K, Shao M. Efficient seeding for error-prone sequences with subseqhash2. *Bioinformatics* 2025;**41**:btaf418. <https://doi.org/10.1093/bioinformatics/btaf418>
- Lipman DJ, Pearson WR. Rapid and sensitive protein similarity searches. *Science* 1985;**227**:1435–41. <https://doi.org/10.1126/science.2983426>
- Liu D, Steinegger M. Block aligner: an adaptive SIMD-accelerated aligner for sequences and position-specific scoring matrices. *Bioinformatics* 2023;**39**:btad487. <https://doi.org/10.1093/bioinformatics/btad487>
- Ma B, Tromp J, Li M. PatternHunter: faster and more sensitive homology search. *Bioinformatics* 2002;**18**:440–5. <https://doi.org/10.1093/bioinformatics/18.3.440>
- Mäkinen, Ukkonen, Navarro. Approximate matching of run-length compressed strings. *Algorithmica* 2003;**35**:347–69. <https://doi.org/10.1007/s00453-002-1005-2>
- Manber U, Myers G. Suffix arrays: a new method for on-line string searches. *SIAM J Comput* 1993;**22**:935–48. <https://doi.org/10.1137/0222058>
- Marco-Sola S, Moure JC, Moreto M *et al*. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics* 2021;**37**:456–63.
- Marco-Sola S, Eizenga JM, Guarracino A *et al*. Optimal gap-affine alignment in O(s) space. *Bioinformatics* 2023;**39**:btad074.
- McCaughey S. Approximate similarity search under edit distance using locality-sensitive hashing. In: *ICDT'21*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, Saarland, Germany, 2021. <https://doi.org/10.4230/LIPICS.ICDT.2021.21>
- Mojica FJM, Díez-Villaseñor C, García-Martínez J *et al*. Short motif sequences determine the targets of the prokaryotic CRISPR defence system. *Microbiology (Reading)* 2009;**155**:733–40. <https://doi.org/10.1099/mic.0.023960-0>
- Musunuru K, Grandinette SA, Wang X *et al*. Patient-specific in vivo gene editing to treat a rare genetic disease. *N Engl J Med* 2025;**392**:2235–43.
- Muth R, Manber U. Approximate multiple string search. In: *Combinatorial Pattern Matching*. Berlin and Heidelberg, Germany: Springer, 1996, 75–86. [https://doi.org/10.1007/3-540-61258-0\\_7](https://doi.org/10.1007/3-540-61258-0_7)
- Myers EW. An O(ND) difference algorithm and its variations. *Algorithmica* 1986;**1**:251–66. <https://doi.org/10.1007/bf01840446>
- Myers G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J ACM* 1999;**46**:395–415.
- Navarro G. A guided tour to approximate string matching. *ACM Comput Surv* 2001;**33**:31–88.

- Navarro G, Baeza-Yates R. Very fast and simple approximate string matching. *Inf Process Lett* 1999;**72**:65–70. [https://doi.org/10.1016/s0020-0190\(99\)00121-0](https://doi.org/10.1016/s0020-0190(99)00121-0)
- Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol* 1970;**48**:443–53.
- Ning Z, Cox AJ, Mullikin JC. SSAHA: a fast search method for large DNA databases. *Genome Res* 2001;**11**:1725–9. <https://doi.org/10.1101/gr.194201>
- Odell M, Russell R. RC 1918–1922. US patent 1,261,167 (1918).
- Oliva A, Tobler R, Cooper A *et al*. Systematic benchmark of ancient DNA read mapping. *Brief Bioinform* 2021;**22**:bbab076. <https://doi.org/10.1093/bib/bbab076>
- Owolabi O, McGregor DR. Fast approximate string matching. *Softw Pract Exp* 1988;**18**:387–93. <https://doi.org/10.1002/spe.4380180407>
- Ran FA, Hsu PD, Wright J *et al*. Genome engineering using the CRISPR-Cas9 system. *Nat Protoc* 2013;**8**:2281–308.
- Reinert K, Dadi TH, Ehrhardt M *et al*. The SeqAn C++ template library for efficient sequence analysis: a resource for programmers. *J Biotechnol* 2017;**261**:157–68. <https://doi.org/10.1016/j.jbiotec.2017.07.017>
- Renders L, Depuydt L, Fostier J. Approximate pattern matching using search schemes and in-text verification. In: *Bioinformatics and Biomedical Engineering*. Berlin and Heidelberg, Germany: Springer International Publishing, 2022, 419–35. [https://doi.org/10.1007/978-3-031-07802-6\\_36](https://doi.org/10.1007/978-3-031-07802-6_36)
- Renders L, Depuydt L, Rahmann S *et al*. Automated design of efficient search schemes for lossless approximate pattern matching. In: *Research in Computational Molecular Biology*. Berlin and Heidelberg, Germany: Springer Nature Switzerland, 2024, 164–84. [https://doi.org/10.1007/978-1-0716-3989-4\\_11](https://doi.org/10.1007/978-1-0716-3989-4_11)
- Renders L, Depuydt L, Gagie T *et al*. Columba: fast approximate pattern matching with optimized search schemes. *Bioinformatics* 2025;**41**:btaf652. <https://doi.org/10.1093/bioinformatics/btaf652>
- Roux S, Neri U, Bushnell B *et al*. Planetary-scale metagenomic search reveals new patterns of CRISPR targeting. bioRxiv, <https://doi.org/10.1101/2025.06.12.659409>, June 2025, preprint: not peer reviewed.
- Rumble SM, Lacroute P, Dalca AV *et al*. SHRiMP: accurate mapping of short color-space reads. *PLoS Comput Biol* 2009;**5**:e1000386. <https://doi.org/10.1371/journal.pcbi.1000386>
- Salmela L, Tarhio J, Kalsi P. Approximate Boyer–Moore string matching for small alphabets. *Algorithmica* 2010;**58**:591–609. <https://doi.org/10.1007/s00453-009-9286-3>
- Scott DA, Zhang F. Implications of human genetic variation in CRISPR-based therapeutic genome editing. *Nat Med* 2017;**23**:1095–101. <https://doi.org/10.1038/nm.4377>
- Sellers PH. Pattern recognition in genetic sequences. *Proc Natl Acad Sci USA* 1979;**76**:3041. <https://doi.org/10.1073/pnas.76.7.3041>
- Sellers PH. The theory and computation of evolutionary distances: pattern recognition. *J Algorithms* 1980;**1**:359–73. [https://doi.org/10.1016/0196-6774\(80\)90016-4](https://doi.org/10.1016/0196-6774(80)90016-4)
- Shapiro RS, Chavez A, Collins JJ. CRISPR-based genomic tools for the manipulation of genetically intractable microorganisms. *Nat Rev Microbiol* 2018;**16**:333–9.
- Šošić M, Šikić M. Edlib: a C/C++ library for fast, exact sequence alignment using edit distance. *Bioinformatics* 2017;**33**:1394–5.
- Stadick S. Ish: SIMD and GPU accelerated local and semi-global alignment as a CLI filtering tool. *Bioinform Adv* 2025;**5**:vbaf292. <https://doi.org/10.1093/bioadv/vbaf292>
- Sutinen E, Tarhio J. On using q-gram locations in approximate string matching. In: *Algorithms—ESA’95*. Berlin and Heidelberg, Germany: Springer, 1995, 327–40. [https://doi.org/10.1007/3-540-60313-1\\_153](https://doi.org/10.1007/3-540-60313-1_153)
- Suzuki H, Kasahara M. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics* 2018;**19**:45. <https://doi.org/10.1186/s12859-018-2014-8>
- Tolstoganov I, Martin M, Buchin N *et al*. Multi-context seeds enable fast and high-accuracy read mapping. *Genome Biol* 2026;**27**:118. <https://doi.org/10.1186/s13059-026-04017-x>
- Tonkin-Hill G, MacAlasdair N, Ruis C *et al*. Producing polished prokaryotic pangenomes with the Panaroo pipeline. *Genome Biol* 2020;**21**:180. <https://doi.org/10.1186/s13059-020-02090-4>
- Ukkonen E. On approximate string matching. In: *International Conference on Fundamentals of Computation Theory*. Berlin and Heidelberg, Germany: Springer, 487–95, 1983. [https://doi.org/10.1007/3-540-12689-9\\_129](https://doi.org/10.1007/3-540-12689-9_129)
- Ukkonen E. Algorithms for approximate string matching. *Inform Control* 1985a;**64**:100–18.
- Ukkonen E. Finding approximate patterns in strings. *J Algorithms* 1985b;**6**:132–7. [https://doi.org/10.1016/0196-6774\(85\)90023-9](https://doi.org/10.1016/0196-6774(85)90023-9)
- Ukkonen E. Approximate string-matching with q-grams and maximal matches. *Theor Comput Sci* 1992;**92**:191–211. [https://doi.org/10.1016/0304-3975\(92\)90143-4](https://doi.org/10.1016/0304-3975(92)90143-4)
- Ukkonen E. Approximate string-matching over suffix trees. In: *Combinatorial Pattern Matching, CPM’93*. Springer-Verlag, 1993, 228–42. <https://doi.org/10.1007/bfb0029808>
- Wagner RA, Fischer MJ. The string-to-string correction problem. *J ACM* 1974;**21**:168–73. <https://doi.org/10.1145/321796.321811>
- Walia S, Ye C, Bera A *et al*. Talco: tiling genome sequence alignment using convergence of traceback pointers. In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Portland, Oregon, USA: IEEE, 2024. <https://doi.org/10.1109/hpca57654.2024.00044>
- Weese D, Holtgrewe M, Reinert K. RazerS 3: faster, fully sensitive read mapping. *Bioinformatics* 2012;**28**:2592–9. <https://doi.org/10.1093/bioinformatics/bts505>
- Wilbur WJ, Lipman DJ. Rapid similarity searches of nucleic acid and protein data banks. *Proc Natl Acad Sci U S A* 1983;**80**:726–30. <https://doi.org/10.1073/pnas.80.3.726>
- Wozniak A. Using video-oriented instructions to speed up sequence comparison. *Comput Appl Biosci* 1997;**13**:145–50. <https://doi.org/10.1093/bioinformatics/13.2.145>
- Wright AH. Approximate string matching using withinword parallelism. *Softw Pract Exp* 1994;**24**:337–62. <https://doi.org/10.1002/spe.4380240402>
- Wu S, Manber U. Fast text searching: allowing errors. *Commun ACM* 1992;**35**:83–91. <https://doi.org/10.1145/135239.135244>
- Wu TD, Watanabe CK. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics* 2005;**21**:1859–75. <https://doi.org/10.1093/bioinformatics/bti310>

- Yaish O, Malle A, Cohen E *et al.* SWOFFinder: efficient and versatile search of CRISPR off-targets with bulges by Smith–Waterman alignment. *iScience* 2024;**27**:108557. <https://doi.org/10.1016/j.isci.2023.108557>.
- Yao B, Li F, Hadjieleftheriou M *et al.* Approximate string search in spatial databases. In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. Portland, Oregon, USA: IEEE, 2010, 545–56. <https://doi.org/10.1109/icde.2010.5447836>
- Zhang J, Lan H, Chan Y *et al.* BGSA: a bit-parallel global sequence alignment toolkit for multi-core and many-core architectures. *Bioinformatics* 2019;**35**:2306–8.

## 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:    $B \leftarrow \lceil n / (4 \cdot 64) \rceil$   $\triangleright$  Number of 64bp blocks per chunk.
3:    $j_{\leq k} \leftarrow 0$   $\triangleright$  Last row in previous block ending in a value  $\leq k$ .
4:    $j_{\max} \leftarrow 0$   $\triangleright$  Last computed row in previous block.
5:    $v^+ \leftarrow [[1, 1, 1, 1]; m]$   $\triangleright$  Initialize list of SIMD of +1 deltas.
6:    $v^- \leftarrow [[0, 0, 0, 0]; m]$   $\triangleright$  Modified when  $\alpha < 1$  for overhang.
7:    $M \leftarrow [[], [], [], []]$   $\triangleright$  Matches ( $pos, cost$ ) per lane.
8:   for  $i \in \{0, \dots, B + \lceil (m + k) / 64 \rceil - 1\}$  do
9:      $i_{\max} \leftarrow i$ 
10:     $h^+, h^- \leftarrow [0, 0, 0, 0]$   $\triangleright$  Deltas in current row.
11:     $d_s, d_e \leftarrow [0, 0, 0, 0]$   $\triangleright$  Dist to start and end of each block.
12:     $j'_{\leq k}, j'_{\max} \leftarrow 0$ 
13:    for  $j \in \{0, \dots, m - 1\}$  do
14:       $d_s \leftarrow d_s + v^+[j] - v^-[j]$ 
15:      for  $\ell \in \{0, 1, 2, 3\}$  do
16:         $\triangleright$  Profile determines a bitmask indicating equal chars.  $\triangleleft$ 
17:         $eq[\ell] \leftarrow Eq(P[j], T[B \cdot \ell + 64 \cdot i \dots B \cdot \ell + 64 \cdot i + 64])$ 
18:        ComputeBlock( $v^+[j], v^-[j], h^+, h^-, eq$ )  $\triangleright$  Myers bitpacking.
19:         $d_e \leftarrow d_e + v^+[j] - v^-[j]$ 
20:        if  $\min_{\ell} d_e[\ell] \leq k$  then
21:           $j'_{\leq k} \leftarrow j$ 
22:           $j'_{\max} \leftarrow j$ 
23:           $\triangleright$  Check if all 256 values are  $> k$ .  $\triangleleft$ 
24:          if  $j > j_{\leq k}$  and AllAboveK( $d_s, h^+, h^-, k$ ) then
25:            for  $j' \in \{j + 1, \dots, j_{\max}\}$  do
26:               $v^+[j'] \leftarrow [1, 1, 1, 1]$   $\triangleright$  Reset “dirty” skipped rows.
27:               $v^-[j'] \leftarrow [0, 0, 0, 0]$ 
28:               $j_{\max} \leftarrow j'_{\max}$ 
29:               $j_{\leq k} \leftarrow j'_{\leq k}$ 
30:              if  $i \geq B$  and  $64 \cdot (i - B) - j > k$  then
31:                Goto 35, break out of  $i$ .  $\triangleright$  Done with chunk overlap.
32:                Goto 8, continue with next  $i$   $\triangleright$  Early break.
33:           $\triangleright$  Find (local minima) end positions with cost  $\leq k$ .  $\triangleleft$ 
34:          FindEndPositionsAtMostK( $M, d_s, h^+, h^-, k$ )
35:        for  $\ell \in \{1, 2, 3\}$  do  $\triangleright$  Prune duplicate matches in overlap.
36:           $M[\ell] \leftarrow \{x \in M[\ell] : x \geq B \cdot \ell + 64 \cdot (i_{\max} + 1)\}$ 
37:    return  $M$ 

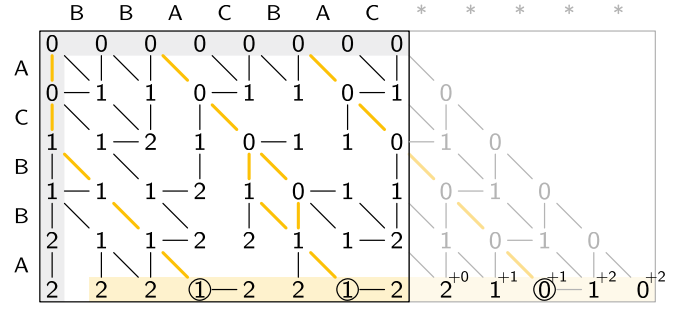
```

## 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 the 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  $P[l \dots m]$  matches a prefix of  $T$ ,
2. there is a *right-overhanging* match when a prefix  $P[0 \dots m - l]$  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  $\ell$  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. Then 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  $\text{Eq}(P[j], T[x \dots y])$  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 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 as 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 similarly 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 harbour 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 specify the maximum 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.

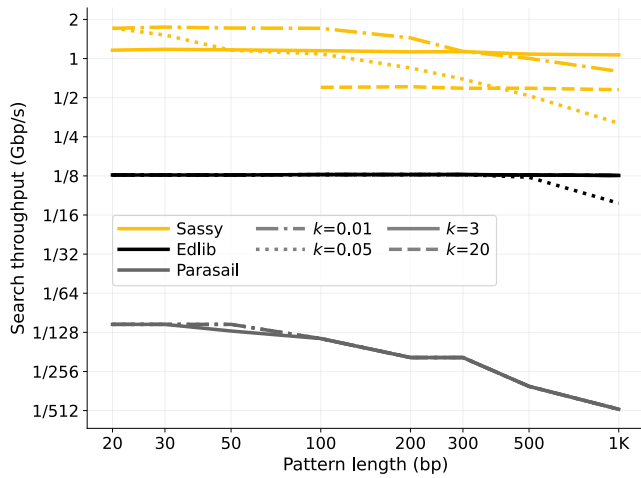
## E. Comparison with parasail

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.

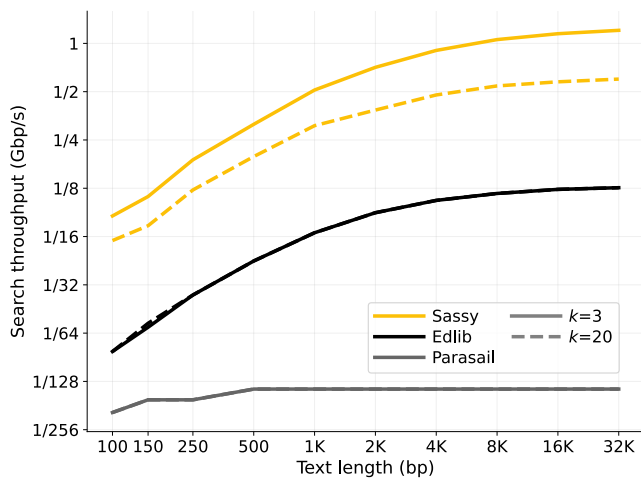
Because affine scoring is not directly comparable to edit distance, we approximated edit distance costs by setting `-gap-open=1`, `-gap-extension=1`, `match-score=0`, and `mismatch-score=-1`. Given the small range of scores under this configuration, we used 8-bit output (`solution_width=8`).

*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  $n = 100$  to  $n = 32000$  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.