A*PA2: Up to 19× faster exact global alignment

Ragnar Groot Koerkamp
ETH Zurich, Zurich, Switzerland

Abstract

Motivation Pairwise alignment is at the core of computational biology. Most commonly used exact methods are either based on $O(ns)$ *band doubling* or $O(n + s^2)$ *diagonal transition*, where n is the sequence length and s the number of errors. However, as the length of sequences has grown, these exact methods are often replaced by approximate methods based on e.g. *seed-and-extend* and heuristics to bound the computed region. We would like to develop an exact method that matches the performance of these approximate methods.

Recently, Astarix introduced the A* shortest path algorithm with the *seed heuristic* for exact sequence-to-graph alignment. A*PA adapted and improved this for pairwise sequence alignment and achieves near-linear runtime when divergence (error rate) is low, at the cost of being very slow when divergence is high.

Methods We introduce A*PA2, an exact global pairwise aligner with respect to edit distance. The goal of A*PA2 is to unify the near-linear runtime of A*PA on similar sequences with the efficiency of dynamic programming (DP) based methods. Like Edlib, A*PA2 uses Ukkonen's band doubling in combination with Myers' bitpacking. A*PA2 1) uses large block sizes inspired by Block Aligner, 2) extends this with SIMD (single instruction, multiple data), 3) introduces a new *profile* for efficient computations, 4) introduces a new optimistic technique for traceback based on diagonal transition, 5) avoids recomputation of states where possible, and 6) applies the heuristics developed in A*PA and improves them using *pre-pruning*.

Results With the first 4 engineering optimizations, A*PA2-simple has complexity $O(ns)$ and is 6× to 8× faster than Edlib for sequences $\geq$ 10 kbp. A*PA2-full also includes the heuristic and is often near-linear in practice for sequences with small divergence. The average runtime of A*PA2 is 19× faster than the exact aligners BiWFA and Edlib on >500 kbp long ONT (Oxford Nanopore Technologies) reads of a human genome having 6% divergence on average. On shorter ONT reads of 11% average divergence the speedup is 5.6× (avg. length 11 kbp) and 0.81× (avg. length 800 bp). On all tested datasets, A*PA2 is competitive with or faster than approximate methods.

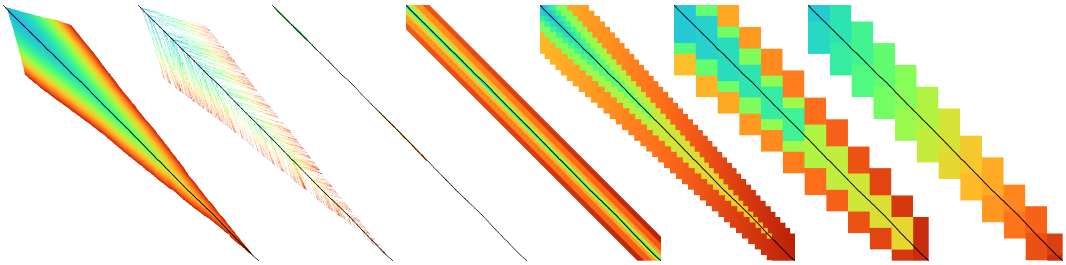
2012 ACM Subject Classification Applied computing → Bioinformatics; Software and its engineering → Software performance; Theory of computation → Shortest paths; Theory of computation → Dynamic programming

Keywords and phrases Edit distance; Pairwise alignment; A*; Shortest path; Dynamic programming

Digital Object Identifier 10.4230/LIPIcs.WABI.2024.17

Supplementary Material *Software*: github.com/RagnarGrootKoerkamp/astar-pairwise-aligner

Funding *Ragnar Groot Koerkamp*: ETH Research Grant ETH-1721-1 to Gunnar Rätsch.



(a) Dijkstra **(b)** WFA **(c)** A*PA **(d)** Ukkonen **(e)** Edlib **(f)** **A*PA2 -simple** **(g)** **A*PA2 -full**

Figure 1 Alignment of two sequences of length 3000 bp with 20% divergence using different methods. Coloured pixels correspond to visited states in the edit graph or dynamic programming matrix, and the blue to red gradient indicates the order of computation. The black path indicates an optimal alignment. (a) Dijkstra is the classical shortest path algorithm. (b) WFA uses the diagonal transition algorithm. (c) A*PA with the gap-chaining seed heuristic. (d) Ukkonen's method uses band doubling. (e) Edlib adds the gap heuristic and bitpacking. (f) A*PA2-simple additionally computes blocks of 256 columns at a time, and (g) A*PA2-full applies the heuristics of A*PA. Figure 14 in Appendix A.3 shows the same methods on a more complicated alignment.

1 Introduction

The problem of *global pairwise alignment* is to find the shortest sequence of edit operations (insertions, deletions, substitutions) to convert a string into a second string [34, 49], where the number of such operations is called the *Levenshtein distance* or *edit distance* [23, 50].

Over time, the length of genomic reads has increased from hundreds of basepairs to hundreds of thousands basepairs now. Meanwhile, the complexity of practical exact algorithms has not been improved by more than a constant factor since the introduction of the diagonal transition algorithm [48, 30].

Our recent work A*PA [13] uses the A* shortest path algorithm to speed up alignment and has near-linear runtime when divergence is low. A drawback of A* is that it uses a queue and must store all computed distances, causing large (up to 500×) overhead compared to methods based on dynamic programming (DP).

This work introduces A*PA2, a method that unifies the heuristics and near-linear runtime of A*PA with the efficiency of DP based methods.

As Fickett [10, p. 1] stated 40 years ago and still true today,

> at present one must choose between an algorithm which gives the best alignment but is expensive, and an algorithm which is fast but may not give the best alignment.

In this paper we narrow this gap and show that A*PA2 is nearly as fast as approximate methods.

1.1 Contributions

We introduce A*PA2, an exact global pairwise sequence aligner with respect to edit distance.

In A*PA2, we combine multiple existing techniques and introduce a number of new ideas to obtain up to 19× speedup over existing single-threaded exact aligners. A*PA2 is often faster and never much slower than approximate methods.

As a starting point, we take the band doubling algorithm implemented by Edlib [58] using bitpacking [31]. First, we speed up the implementation (points 1., 2., 3.). Then, we reduce the amount of work done (4., 5.). Lastly, we apply A* heuristics (6.).

1. **Block-based computation** Edlib (Figure 1e) computes one column of the DP matrix at a time, and for each column decides which range (subset of rows) of states to compute. We significantly reduce this overhead by processing blocks of 256 columns at a time (Figure 1f), taking inspiration from Block Aligner [26]. Correspondingly, we only store states of the DP-matrix at block boundaries, reducing memory usage.
2. **SIMD** We speed up the computation of each block by using 256bit SIMD, allowing the processing of 4 computer words in parallel.
3. **Novel encoding** We introduce a novel encoding of the input sequence to speed up SIMD operations by comparing characters bit-by-bit and avoiding slow `gather` instructions. This limits the current implementation to alphabets of size 4.
4. **Incremental doubling** Both the band doubling method of Ukkonen [48] and Edlib recompute states after doubling the threshold. We avoid this by using the theory behind the A* algorithm, extending the incremental doubling of Fickett [10] to blocks and arbitrary heuristics.
5. **Traceback** For the traceback, we optimistically use the diagonal transition method [48, 30, 29] within each block with a strong adaptive heuristic, only falling back to a full recomputation of the block when needed.
6. **A*** We apply the gap-chaining seed heuristic (GCSH) of A*PA [13] (Figures 1c and 1g), and improve it using *pre-pruning*. This technique discards most of the *spurious* (off-path) matches ahead of time.

1.2 Previous work

We give a brief overview of developments that this work builds on, in chronological order per approach. See also, e.g., the reviews by Kruskal [22] and Navarro [33], and the introduction of the A*PA paper [13]. Section 2 covers relevant topics more formally.

Needleman-Wunsch Pairwise alignment has classically been approached as a dynamic programming problem. For input strings of lengths n and m, this method creates a $(n+1) \times (m+1)$ table that is filled cell by cell using a recursive formula. Needleman and Wunsch [34] gave the first $O(n^2m)$ algorithm, and Sellers [41] and Wagner and Fischer [50] improved this to what is now known as the $O(nm)$ *Needleman-Wunsch algorithm*, building on the quadratic algorithm for *longest common subsequence* by Sankoff [40].

Graph algorithms It was already realized early on that an optimal alignment corresponds to a shortest path in the *edit graph* [49, 48]. Both Ukkonen and Myers [30] remarked that this can be solved using Dijkstra's algorithm [7], taking $O(ns)$ time (Figure 1a), where s is the edit distance between the two strings and is typically much smaller than the string length. (Although Ukkonen only gave a bound of $O(nm \log(nm))$.) However, Myers [30, p. 2] observed that

> the resulting algorithm involves a relatively complex discrete priority queue and this queue may contain as many as $O(ns)$ entries even in the case where just the length of the [...] shortest edit script is being computed.

Hadlock [14] realized that Dijkstra's algorithm can be improved upon by using A* [16], a more *informed* algorithm that uses a *heuristic* function h that gives a lower bound on the remaining edit distance between two suffixes. He uses two heuristics, the widely used *gap cost* heuristic [48, 14, 55, 44, 45, 35] that simply uses the difference between the lengths of the suffixes as lower bound (Figure 1e), and an improved heuristic based on character frequencies in the two suffixes. In Astarix, Ivanov et al. [19, 20] use A* for sequence-to-graph

alignment and introduce the *seed heuristic*. A*PA [13] improves this to the *gap-chaining seed heuristic* (GCSH) with *pruning* to obtain near-linear runtime when errors are uniform random (Figure 1c). Nevertheless, as Spouge [45, p. 3] states,

> algorithms exploiting the lattice structure of an alignment graph are usually faster

and further [44, p. 4]:

> This suggests a radical approach to A* search complexities: dispense with the lists [of open states] if there is a natural order for vertex expansion.

In this work we follow this advice and replace the plain A* search in A*PA with a much more efficient approach based on *computational volumes* that merges DP and A*.

Computational volumes Wilbur and Lipman [52] were, to our knowledge, the first to speed up the $O(nm)$ DP algorithm, by only considering states near diagonals with many *k-mer matches*, but at the cost of giving up the exactness of the method. Fickett [10] noted that for some chosen parameter t that is at least the edit distance s, only those DP-states with cost at most t need to be computed. This only requires $O(nt)$ time, which is fast when t is an accurate bound on the distance s. For example t can be set as a known upper bound for the data being aligned, or as the length of a suboptimal alignment. When $t = t_0$ turns out too small, a larger new bound t_1 can be chosen, and only states with distance in between t_0 and t_1 have to be computed. When t increases by 1 in each iteration, this closely mirrors Dijkstra's algorithm.

Ukkonen [48] introduced a very similar idea, statically bounding the computation to only those states that can be contained in a path of length at most t from the start to the end of the graph (Figure 1d). On top of this, Ukkonen introduced *band doubling*: $t_0 = 1$ can be doubled ($t_i = 2^i$) until t_k is at least the actual distance s. This finds the alignment in $O(ns)$ time.

Spouge [44] unified the methods of Fickett and Ukkonen in *computational volumes* (see Section 2), small subgraphs of the full edit graph that are guaranteed to contain all shortest paths. As Spouge noted:

> The order of computation (row major, column major or antidiagonal) is just a minor detail in most algorithms.

But this is exactly what was investigated a lot in the search for more efficient implementations.

Parallelism In the 1990s, the focus shifted from reducing the number of computed states to computing states faster through advancements in implementation and hardware. This resulted in a plethora of new methods. While there are many recent methods optimizing the computation of arbitrary scoring schemes and affine costs [43, 12, 5, 46, 42], here we focus on methods for computing edit distance.

The first technique in this direction is *microparallelism* [1], also called SWAR (SIMD within a register), where each (64 bit) computer word is divided into multiple (e.g. 16 bit) parts, and word-size operations modify all (4) parts in parallel. This was then applied with *inter-sequence parallelism* to align a given query to multiple reference sequences in parallel [1, 2, 54, 18, 38]. Hughey [17] noted that *anti-diagonals* of the DP matrix are independent and can be computed in parallel, to speed up single alignments. Wozniak [53] applied SIMD (single instruction, multiple data) for this purpose, which are special CPU instructions that operate on multiple computer words at a time. Rognes and Seeberg [39, p. 702] also use microparallelism, but use *vertical* instead of anti-diagonal vectors:

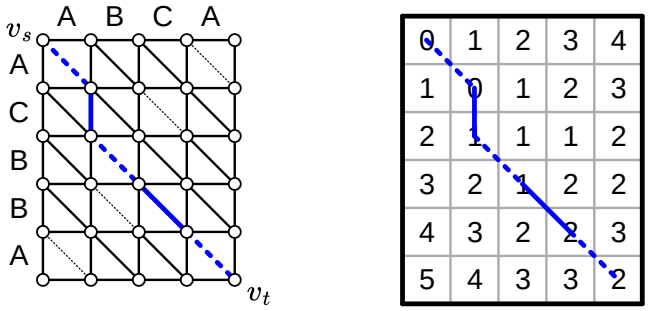


Figure 2 An example of an edit graph (left) corresponding to the alignment of strings `ABCA` and `ACBBA`, adapted from [41]. Solid edges indicate insertion/deletion/substitution edges of cost 1, while dashed edges indicate matches of cost 0. All edges are directed from the top-left to the bottom-right. The shortest path of cost 2 is shown in blue. The right shows the corresponding dynamic programming (DP) matrix containing the distance $g^*(u)$ to each state.

> The advantage of this approach is the much-simplified and faster loading of the vector of substitution scores from memory. The disadvantage is that data dependencies within the vector must be handled.

To work around these dependencies, Farrar [9] introduced an alternative *striped* SIMD scheme where lanes are interleaved with each other. A*PA2 does not use this, but for example BSAlign [42] does.

Myers [31] introduced a *bitpacking* algorithm specifically for edit distance (Figure 1e). It bit-encodes the differences between $w = 64$ states in a column into two computer words and gives an efficient algorithm to operate on them. This provides a significant speedup over previous methods. The supplement of BitPAl [27, 3] introduces an alternative scheme for edit distance based on a different bit-encoding, but as both methods end up using 20 instructions (see Appendix A.1) we did not pursue this further.

Tools There are many semi-global aligners that implement $O(nm)$ (semi)-global alignment using numerous of the aforementioned implementation techniques, such as SeqAn [8], Parasail [6], SWIPE [38], Opal [57], libssa [11], SWPS3 [47], and SSW library [56].

Other methods use *seed-chain-extend* as a proxy for edit distance by considering *chains* of matching k-mers (*seeds*). LCSk [4] and LCSk++ [37, 36] find a maximal chain of (overlapping) k-mers. DAlign [32] and Minimap [24, 25] further *extend* the best chain into a full alignment. ChainX [21] gives an efficient method to penalize the gap cost between consecutive matches.

Dedicated global alignment implementations implementing band doubling are much rarer. Edlib [58] implements $O(ns)$ band doubling and Myers' bitpacking (Figure 1e). KSW2 implements band doubling for affine costs [46, 25]. WFA and BiWFA [29, 28] implement the $O(n + s^2)$ expected time *diagonal transition* algorithm [48, 30] (Figure 1b). Block Aligner [26] is an approximate aligner that can handle position-specific scoring matrices whose main novelty is to divide the computation into larger blocks. Recently, BSAlign [42] provided a new implementation of band doubling based on Farrar's striped method that focusses on affine costs but also supports edit distance. Lastly, A*PA [13] directly implements A* on the alignment graph using the gap-chaining seed heuristic.

2 Preliminaries

Edit graph We take as input two zero-indexed sequences A and B over an alphabet of size 4 of lengths n and m. The *edit graph* (Figure 2) contains *states* $\langle i, j\rangle$ ($0 \le i \le n$, $0 \le j \le m$) as vertices. It further contains directed insertion and deletion edges $\langle i, j\rangle \to \langle i, j+1\rangle$ and $\langle i, j\rangle \to \langle i+1, j\rangle$ of cost 1, and diagonal edges $\langle i, j\rangle \to \langle i+1, j+1\rangle$ of cost 0 when $A_i = B_j$ and substitution cost 1 otherwise. A shortest path from $v_s := \langle 0, 0\rangle$ to $v_t := \langle n, m\rangle$ in the edit graph corresponds to an optimal alignment of A and B. The *distance* $d(u, v)$ from u to v is the length of the shortest (minimal cost) path from u to v, and we use *distance*, *length*, and *cost* interchangeably. We write $g^*(u) := d(v_s, u)$ for the distance from the start to u, $h^*(u) := d(u, v_t)$ for the distance from u to the end, and $f^*(u) := g^*(u) + h^*(u)$ for the minimal cost of a path from v_s to v_t through u.

A* is a shortest path algorithm based on a *heuristic* function $h(u)$ [16]. A heuristic is called *admissible* when $h(u)$ never overestimates the distance to the end, i.e., $h(u) \le h^*(u)$, and admissible h guarantee that A* finds a shortest path. A* *expands* states in order of increasing $f(u) := g(u) + h(u)$, where $g(u)$ is the best distance to u found so far. We say that u is *fixed* when the distance to u has been found, i.e., $g(u) = g^*(u)$. The gap cost heuristic $h(\langle i, j\rangle) = c_{\text{gap}}(\langle i, j\rangle, v_t) = |(n - i) - (m - j)|$ is an example of a simple admissible heuristic.

Computational volumes Spouge [44] defines a *computational volume* as a subgraph of the alignment graph that contains all shortest paths. Given a bound $t \ge s$, some examples of computational volumes are:

1. The entire $(n + 1) \times (m + 1)$ graph or DP table.
2. $\{u : g^*(u) \le t\}$, the states at distance $\le t$, introduced by Ficket [10] and similar to Dijkstra's algorithm (Figures 1a and 1b).
3. $\{u : c_{\text{gap}}(v_s, u) + c_{\text{gap}}(u, v_t) \le t\}$ the static set of states possibly on a path of cost $\le t$ (Figure 1d) [48].
4. $\{u : g^*(u) + c_{\text{gap}}(u, v_t) \le t\}$, as used by Edlib (Figures 1e and 1f) [58, 45].
5. $\{u : g^*(u) + h(u) \le t\}$ for a heuristic h, which A*PA2 uses (Figures 1c and 1g).

Band doubling is the following algorithm by Ukkonen [48], that depends on the choice of computational volume being used: For a given t, we can test whether an alignment of cost $\le t$ exists by iterating over all columns, and in each column computing the distance to the range of rows $[j_{start}, j_{end}]$ corresponding ot the computational volume being used. When the last column is reached and the distance to v_t is $\le t$, an optimal alignment is found. Otherwise, the edit distance is $> t$.

Since the actual distance s is not known, multiple *iterations* are used and $t_i = 2^i$ is doubled until $t_k = 2^k \ge s > 2^{k-1}$. Typically each iteration requires $O(n \cdot t)$ time, and hence the total time is $n \cdot 1 + \cdots + n \cdot 2^k < 4 \cdot n \cdot 2^{k-1} < 4 \cdot n \cdot s = O(ns)$. Note that without reusing values from previous iterations, on average each state is computed twice.

Myers' bitpacking exploits that the difference in distance $g^*(u)$ to adjacent states is always in $\{-1, 0, +1\}$ [31]. The method bit-encodes $w = 64$ differences between 65 adjacent states in a column in two indicator words, indicating positions where the difference is $+1$ and -1 respectively. Given also the difference along the top, the differences along the right and bottom of a $1 \times w$ rectangle can be computed in only 20 bit operations (Appendix A.1). We call each non-overlapping chunk of 64 rows a *lane*, so that there are $\lceil m/64 \rceil$ lanes, where the last lane may be padded. Note that this method originally only uses 17 instructions, but some additional instructions are needed to support multiple lanes when $m > w$.

Profile Instead of computing each substitution score $S[A_i][B_j] = [A_i \neq B_j]$ for the 64 states in a word one by one, Myers' algorithm first builds a *profile* [39]. For each character c, $Eq[c]$ stores a length m bitvector indicating which characters of B equal c. This way, adjacent scores in a column are simply found as $Eq[A_i][j \dots j']$.

Edlib implements band doubling using the $g^*(u) + c_{\text{gap}}(u, v_t) \leq t$ computational volume and bitpacking [58]. For traceback, it uses Hirschberg's *meet-in-the-middle* approach: once the distance is found, the alignment is started over from both sides towards the middle column, where a state on the shortest path is determined. This is recursively applied to the left and right halves until the sequences are short enough that $O(ns)$ memory can be used.

A*PA uses A* with the *seed heuristic* [20, 13]. In its simplest form, sequence A is split into *seeds* of length $k = 15$. For each seed, all *matches* in B are found using a hashmap containing all k-mers of B. The *seed heuristic* in a state u is then the number of upcoming seeds without matches, since each unmatched seed requires at least one edit to align. This is first extended using *chaining*, which requires matches to form a *chain* (Figure 4a, somewhat alike a dot plot), and second using *gap-chaining*, where joining two seeds incurs the gap cost between them (similar to e.g. [21]). In these last cases, the value of the heuristic is the number of remaining seeds minus the length of the longest chain. A*PA not only supports *exact* matches, but also *inexact* matches, so that a seed without matches takes at least $r = 2$ edits to cross.

During the A* search, matches are *pruned* as soon as a shortest path to their start is found, which increases the strength of the heuristic as the search progresses.

3 Methods

Conceptually, A*PA2 builds on Edlib. First we describe how we make the implementation more efficient using SIMD and blocks. Then, we modify the algorithm itself by using a new traceback method and avoiding unnecessary recomputation of states. On top of that, we apply the A*PA heuristics for further speed gains on large and complex alignments, at the cost of larger precomputation time to build the heuristic.

At the core, A*PA2 uses band doubling with the $g^*(u) + h(u) \leq t$ computational volume, in combination with bitpacking. That is, in each *iteration* of t we compute the distance to all states with $g^*(u) + h(u) \leq t$. In its simple form, A*PA2-simple, we use the gap heuristic, like Edlib does. The initial value for the tested distance t is the value of the heuristic at the start, and in the ith iteration $t_i := h(\langle 0, 0 \rangle) + B \cdot 2^i$, where B is the block size introduced below.

Section 3.1 to Section 3.6 describe our new methods, while Section 3.7 and Section 3.8 show the mathematical details of the algorithm.

3.1 Blocks

Instead of determining the range of rows to be computed for each column individually, we determine it once per *block* of $B = 256$ consecutive columns. This computes some extra states, but reduces the overhead by a lot. (From here on, B stands for the block size, and not for the sequence B to be aligned.) Within each block, we iterate over *lanes* of $w = 64$ rows at a time, and for each lane compute all B columns before moving on to the next lane.

Section 3.7 explains in detail how the range of rows to be computed is determined.

3.2 Memory

Where Edlib does not initially store intermediate values and uses meet-in-the-middle to find the alignment, A*PA2 stores the distance to all states at the end of *each* block, encoded as the distance to the top-right state of the block and the bit-encoded vertical differences along the right-most column. This simplifies the traceback method (see Section 3.5), and has sufficiently small memory usage to be practical.

3.3 SIMD

While it is tempting to use a SIMD vector as a single $W = 256$-bit word, the four $w = 64$-bit words (SIMD lanes) are dependent on each other and require manual work to shift bits between the lanes. Instead, we let each 256-bit AVX2 SIMD vector represent four 64-bit words (lanes) that are anti-diagonally staggered as in Figure 3a. This is similar to the original anti-diagonal tiling introduced by Wozniak [53], but using units of w-bit words instead of single characters. This idea was already introduced in 2014 by the author of Edlib in a GitHub issue (github.com/Martinsos/edlib/issues/5), but to our knowledge has never been implemented in either Edlib or elsewhere.

We further improve instruction-level-parallelism (ILP) by processing 8 lanes at a time using two SIMD vectors in parallel, spanning a total of 512 rows (Figure 3a).

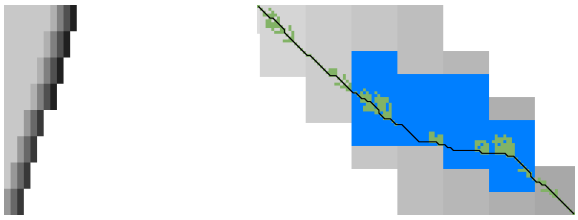
When the number of remaining lanes in a block to be computed is ℓ, we process 8 lanes in parallel as long as $\ell \geq 8$. If there are remaining lanes, we end with another 8-lane ($5 \leq \ell < 8$) or 4-lane ($1 \leq \ell \leq 4$) iteration that optionally includes some padding lanes at the bottom. In case the horizontal differences along the original bottom row are needed (as required by incremental doubling Section 3.8), we can not use padding and instead fall back to trying a 4-lane SIMD ($\ell \geq 4$), a 2-lane SIMD ($\ell \geq 2$), and lastly a scalar iteration ($\ell \geq 1$).

3.4 SIMD-friendly sequence profile

A drawback of anti-diagonal tiling is that each lane of a SIMD vector corresponds to a different column with character a_i that needs to be looked up in the profile $Eq[a_i][\ell]$. While SIMD can do multiple lookups in parallel using `gather` instructions, these instructions are not always efficient. Thus, we introduce the following alternative scheme. Let $b = \lceil \log_2(\sigma) \rceil$ be the number of bits needed to encode each character, with $b = 2$ for DNA. For each lane, the new profile Eq' stores b words as an $\lceil m/w \rceil \times b$ array $Eq'[\ell][p]$. Each word $0 \leq p < b$ stores the negation of the pth bit of each character in its lane. To check which characters in lane ℓ contain character c with bit representation $\overline{c_{b-1} \dots c_0}$, we precompute b words $C_0 = \overline{c_0 \dots c_0}$ to $C_{b-1} = \overline{c_{b-1} \dots c_{b-1}}$ and then compute $\bigwedge_{j=0}^{b-1}(C_j \oplus Eq'[\ell][j])$, where $\oplus$ denotes the xor operation. As an example take $b = 2$ and a lane with $w = 8$ characters $(0, 1, 2, 2, 3, 3, 3, 3)$. Then $Eq'[\ell][0] = \overline{00001101}$ (the negation of indicating odd positions) and $Eq'[\ell][1] = \overline{00000011}$, keeping in mind that bits are shown in reverse order in this notation. If the column now contains character $c = 2 = \overline{10}$ we initialize $C_0 = \overline{00000000}$ and $C_1 = \overline{11111111}$ and compute

$$(C_0 \oplus Eq'[\ell][0]) \wedge (C_1 \oplus Eq'[\ell][1]) = \overline{00001101} \wedge \overline{11111100} = \overline{00001100},$$

indicating that 0-based positions 2 and 3 contain character 2. This naturally extends to SIMD vectors, where each lane is initialized with its own constants.



(a) SIMD **(b)** Traceback

Figure 3 (a) SIMD processing of two times 4 lanes in parallel. This example uses lanes of 4 instead of 64 rows. First the top-left triangle is computed lane by lane, and then 8-lane diagonals are computed by using two 4-lane SIMD vectors in parallel. (b) Computed blocks are shown in grey. States expanded by the diagonal transition traceback in each block are shown in green. When the distance in a block is too large, a part of the block is fully recomputed as fallback, as shown in blue. In regions with low divergence, diagonal transition is sufficient to trace the path, and only in noisy regions the algorithm falls back to recomputing full blocks.

3.5 Traceback

The traceback stage takes as input the computed vertical differences at the end of each block of columns. We iteratively work backwards through the blocks. When tracing the block covering columns i to $i + B$, we know the distances $g(\langle i, j\rangle)$ to the states in column i at the start of the block, and a state $u = \langle i + B, j\rangle$ at distance $g^*(u)$ in column $i + B$ at the end of the block that is on an optimal path. The goal is to find an optimal path from column i to u.

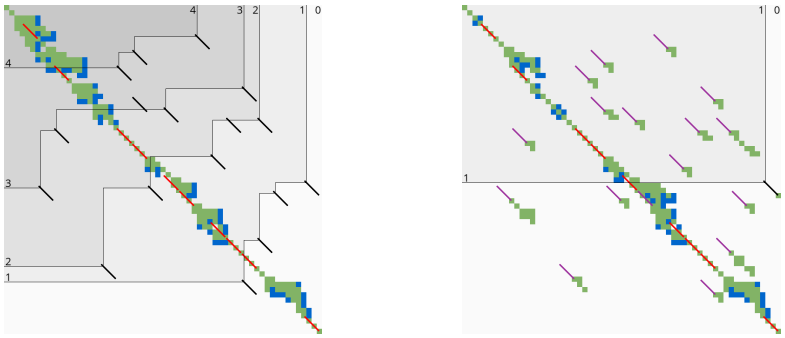
A naive approach is to simply recompute the entire block of columns while storing distances to all states. Here we consider two more efficient methods.

Optimistic block computation Instead of computing the full range of rows for this column, a first insight is that only rows up to j are needed, since an optimal path to $u = \langle i + B, j\rangle$ can never go below row j.

Secondly, the path crosses $B = 256$ columns, and so we optimistically assume that it will be contained in rows $j - 256 - 64 = j - 320$ to j. Thus, we first recompute this range of rows (rounded out to multiples of $w = 64$) from left to right while storing intermediate values, as shown in blue in Figure 3b. If the distance to u computed this way equals $g^*(u)$, there is a shortest path contained within the computed rows and we trace it one state at a time. Otherwise, we repeatedly try again with double the number of lanes until success. The exponential search ensures low overhead and good average case performance.

Optimistic diagonal transition traceback (DTT) A second improvement uses the *diagonal transition* algorithm backwards from u. We simply run the unmodified algorithm on the reverse graph covering columns i to $i + B$ and rows 0 to j. Whenever a state v in column i is reached, say at distance d from u, we check whether $g(v) + d = g^*(u)$, and continue until a v is found for which this holds. We then know that v lies on a shortest path and we can find the path from v to u via a usual traceback on the diagonal transition algorithm, as shown in green in Figure 3b.

As a further optimization, when no suitable v is found after trying all states at distance $\leq g = 40$, we abort the DTT and fall back to the block doubling described above. Another optimization is the WF-adaptive heuristic introduced by WFA [29]: all states that lag more than $x = 10$ behind the furthest reaching diagonal are dropped. Lastly, we abort early when it takes cost $> g/2$ to cross half the columns. Both g and x are experimentally determined, see Figure 12 in Appendix A.3.



(a) Without pre-pruning **(b)** With pre-pruning

Figure 4 Effect of pre-pruning on chaining seed heuristic (CSH) contours. The left shows contours and layers of the heuristic at the end of an A*PA alignment, after matches (black diagonals) on the path have been pruned (red diagonals). The right shows pre-pruned matches in purple and the states visited during pre-pruning in green. After pre-pruning, almost no off-path matches remain. This decreases the number of contours, making the heuristic stronger, and simplifies contours, making the heuristic faster to evaluate.

3.6 A* and pruning

EDLIB already uses a simple *gap cost* heuristic that gives a lower bound on the number of insertions and deletions on a path from each state to the end. We replace this by the much stronger gap-chaining seed heuristic (GCSH) introduced in A*PA, with two modifications.

Bulk pruning In A*PA, matches are *pruned* as soon as a shortest path to their start has been found. This helps to penalize states *before* (left of) the match. Each iteration of A*PA2 works left-to-right only, so that pruning of matches does not affect the current iteration. Thus, we collect all matches to be pruned at the end of each iteration, and update the contours in one right-to-left sweep. To keep the computational volume valid after pruning, we ensure that the range of computed rows in each column never shrinks.

Pre-pruning Here we introduce an independent optimization that also applies to the original A*PA method. Each of the heuristics h introduced in A*PA [13] depends on the set of *matches* $\mathcal{M}$ between seeds in A and k-mers of B.

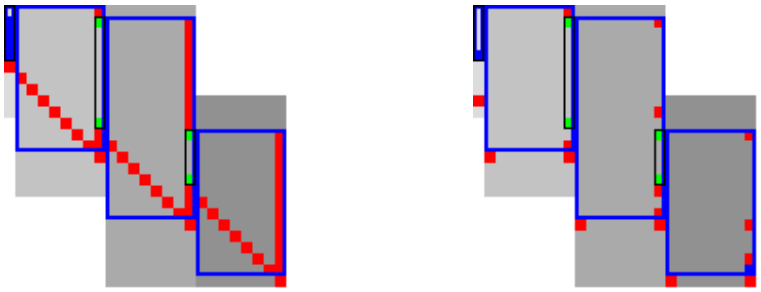
Now consider a seed s_i with an exact match m. The existence of the match is a 'promise' that seed s_i can be crossed for free. (Seeds without match require at least 1 edit.) When m can not be extended into an alignment of s_i and s_{i+1} of cost less than 2, we can amortize this cost over the two seeds and regard m as a 'false promise', since crossing the two seeds takes cost at least 2. Thus, we remove m, making the heuristic more accurate.

More generally, we try to extend each match m into an alignment of seeds s_i up to the start of s_{i+q} for all $q \leq p = 14$. If all extensions have cost $\geq q$, then m falsely promised that s_i to s_{i+q} can be crossed for cost $< q$ and we *pre-prune* (remove) m.

The extension of each match is done by running the diagonal transition algorithm from its end. Any furthest reaching states that are at distance $\geq q$ while at most q seeds have been covered are dropped, and the match is pre-pruned when no active states are left.

As shown in Figure 4b, the effect is that the number of off-path matches is significantly reduced. This makes contours simpler and hence faster to initialize, update, and query, and it increases the value of the heuristic.

Pre-pruning lowers number of remaining matches and allows using $k = 12$ instead of $k = 15$, further improving the heuristic. For $k = 12$, $p = 14$ was experimentally determined to remove nearly all off-path matches, while not taking significant time (Figure 12).



(a) Simple **(b)** Sparse

Figure 5 Detail of computed ranges (a) Starting with a fixed range between two green states (black rectangle), first, the bottom of the to be computed region (blue rectangle) is computed (red diagonal, with $f_l(u) > t$). Then, the region is rounded out to multiples of w and computed (grey background). Lastly, the last column is shrunk (red, $f(u) > t$) until states with $f(u) \leq t$ are found (green), that determine the fixed range (black rectangle). The last block has no fixed states in its right column, so t must be increased. (b) The sparse variant uses much fewer heuristic invocations.

3.7 Determining the rows to compute

For each block, from column i to $i+B$, we only compute those rows that can possibly contain states on a path/alignment of cost at most t. Intuitively, we try to 'trap' the alignment inside a wall of states that can not lie on a path of length at most t, i.e., that must have $f^*(u) > t$, as can be seen in red in Figure 5a. We determine this range of rows in two steps.

Step 1: Fixed range First, we determine the *fixed range* at the end of the preceding block. I.e., we find the topmost and bottommost states $\langle i, j_{start}\rangle$ and $\langle i, j_{end}\rangle$ with $f(u) = g(u) + h(u) \leq t$, as shown in green in Figure 5. All in-between states $u = \langle i, j\rangle$ with $j_{start} \leq j \leq j_{end}$ are then *fixed*, meaning that the correct distance has been found and $g(u) = g^*(u)$. One way to find j_{start} and j_{end} is by simply iterating inward from the start/end of the computed range and skipping all states with $f(u) = g(u) + h(u) > t$, as indicated by the red columns in Figure 5a.

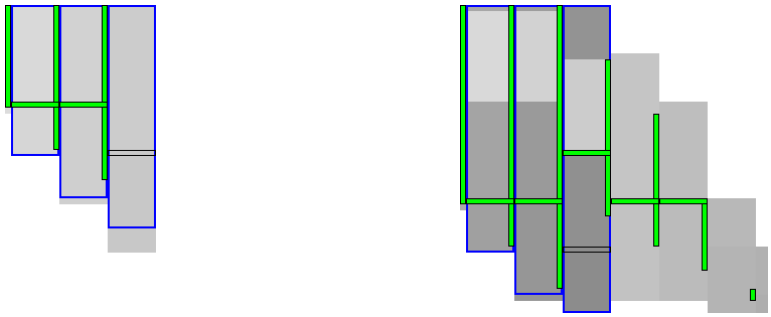
Step 2: End of computed range Then, we find the bottommost state $v = \langle i + B, j'_{end}\rangle$ at the end of the to-be-computed block that can possibly lie on a path of length $\leq t$. We then compute rows j_{start} to j'_{end} in columns i to $i + B$, rounding $[j_{start}, j'_{end}]$ *out* to the previous/next multiple of the word size $w = 64$.

To determine j'_{end}, let $u = \langle i, j_{end}\rangle$ be the bottommost fixed state in column i with $f(u) \leq t$. Let $v = \langle i', j'\rangle$ be a state in the current block ($i \leq i' \leq i + B$) that is strictly below the diagonal of u. Suppose v lies on a path of length $\leq t$. This path must cross column i in or above u, since states u' below u have $f^*(u') > t$. The distance to v is now at least $\min_{j \leq j_{end}} \left(g^*(\langle i, j\rangle) + c_{\text{gap}}(\langle i, j\rangle, v)\right) \geq g^*(u) + c_{\text{gap}}(u, v)$, and thus we define

$$f_l(v) := g^*(u) + c_{\text{gap}}(u, v) + h(v) \leq f^*(v)$$

as a lower bound on the length of the shortest path through v, assuming v is strictly below the diagonal of u and $f^*(v) \leq t$. When $f_l(v) > t$, this implies $f^*(v) > t$ and also $f^*(v') > t$ for all v' below v. The end of the range is now computed by finding the bottommost state v in each column for which f_l is at most t, using the following algorithm:

1. Start with $v = \langle i', j'\rangle = u = \langle i, j_{end}\rangle$.
2. While the below-neighbour $v' = \langle i', j' + 1\rangle$ of v has $f_l(v) \leq t$, increment j'.
3. Go to the next column by incrementing i' and j' by 1 and repeat step 2, until $i' = i + B$.



(a) First iteration **(b)** Second iteration

Figure 6 Incremental doubling detail Blue rectangles show the ranges required to be computed, and grey the computed blocks. Vertical green rectangles show the fixed range at the end of each block, and green horizontal rectangles a fixed row j_f of states inside some blocks. In both figures the third block was just computed, in the (a) first and (b) second iteration. The black empty rectangle indicates the new candidate j'_f for the fixed horizontal region. In (a), the computation is split into two parts, above and below j'_f. In (b), the computation is split into three parts (dark grey): above the reusable region, between the old j_f and the new j'_f, and below the new j'_f.

The row j'_{end} of the last v we find in this way is the bottommost state in column $i + B$ that can possibly have $f(v) \leq t$, and hence this is end of the range we compute.

In Figure 5a, we see that $f(v)$ is evaluated at a diagonal of states just below the bottommost fixed (green) state u at the end of the preceding black, and that the to-be-computed range (indicated in blue) includes exactly all states above this diagonal.

Sparse heuristic invocation A drawback of the previous method is that it requires a large number of calls to f and hence to the heuristic h: roughly one per column and one per row. In Appendix A.2 we present a *sparse* version that uses fewer calls to f, as shown in Figure 5b.

3.8 Incremental doubling

When band doubling doubles the threshold from t to $2t$, it simply recomputes the distance to all states. On the other hand, Dijkstra visits states in increasing order of distance, and the distance to a state is correct (*fixed*) as soon as a state is expanded.

Indeed, band doubling algorithm can also avoid recomputations. After completing the iteration for t, it is guaranteed that the distance is fixed to all states that satisfy $f(u) \leq t$. In fact, a stronger result holds: in any column, the distance is fixed for all states *between* the topmost and bottommost state in that column with $f(u) \leq t$.

In each block, we would like to skip some rows preceding j_{end}, the end of the fixed range in its first column. To be able to do this, we must store the horizontal differences along row j_{end} so that we can continue from there in the next iteration. In practice, we choose row j_f (for *fixed*) as the last row at a lane boundary before j_{end}, as indicated in Figure 6 by a horizontal black rectangle. In the first iteration (Figure 6a), reusing values is not yet possible, so the computation of the block is split into two parts: one above j_f, to extract the horizontal differences along row j_f, and the remainder below j_f.

In the second and further iterations (Figure 6b), the values at j_f are reused and each block is split into three parts. The first part computes all lanes covering states before the start of the fixed range (green column) at the end of the block. We then skip the lanes up to the previous j_f, since the values at both the bottom and right of this region are already fixed. Then, we compute the lanes between the old j_f and its new value j'_f. Lastly we compute the lanes from j'_f to the end.

4 Results

A*PA2 is available at github.com/RagnarGrootKoerkamp/astar-pairwise-aligner and written in Rust. We compare it against other aligners on real datasets, report the impact of the individual techniques we introduced, and measure time and memory usage.

4.1 Setup

Datasets We benchmark on five datasets containing real sequences of varying length and divergence, as listed in detail in Table 1 in Appendix A.3. They can be downloaded from github.com/pairwise-alignment/pa-bench/releases/tag/datasets.

Four datasets containing Oxford Nanopore Technologies (ONT) reads are reused from the WFA, BiWFA, and A*PA evaluations [29, 28, 13]. Of these, two '>500 kbp' and '>500 kbp with genetic variation' datasets have divergence (error rate, or more precisely, edit distance divided by length) $6-7\%$, while two '1 kbp' and '10 kbp' datasets are filtered for sequences of length <1 kbp and <50 kbp and have average divergence 11% and average sequence length 800 bp and 11 kbp.

A SARS-CoV-2 dataset was newly generated by downloading 500 MB of viral sequences from the COVID-19 Data Portal, covid19dataportal.org [15], filtering out non-ACTG characters, and selecting 10000 random pairs. This dataset has average divergence 1.5% and average length 30 kbp.

For each set, we sorted all sequence pairs by edit distance and split them into 50 files each containing multiple pairs, so that the first file contains the 2% of pairs with the lowest divergence. Reported runtimes are averaged over the sequences in each file.

Algorithms and aligners We benchmark A*PA2 against state-of-the-art exact aligners Edlib, BiWFA, and A*PA. We further compare against the approximate aligners WFA-Adaptive [29] and Block Aligner. For WFA-Adaptive we use default parameters $(10, 50, 10)$, dropping states that lag behind by more than 50. For Block Aligner we use block sizes from 0.1% to 1% of the input size. Block Aligner only supports affine costs so we use gap-open cost 1 instead of 0.

We compare two versions of A*PA2. *A*PA2-simple* uses all engineering optimizations (bitpacking, SIMD, blocks, new traceback) and uses the simple gap-heuristic. *A*PA2-full* additionally uses more complicated techniques: incremental-doubling, and the gap-chaining seed heuristic introduced by A*PA with pre-pruning.

Parameters For A*PA2, we fix block size $B = 256$. For A*PA2-full, we use the gap-chaining seed heuristic (GCSH) of A*PA with exact matches and seed length $k = 12$. We pre-prune matches by looking ahead up to $p = 14$ seeds. Parameters were determined experimentally, see Figure 12 in Appendix A.3 for a comparison. For most parameters, the runtime is not very sensitive to the exact value. For A*PA, we use the original inexact matches with seed length $k = 15$ by default, and only change this for the low-divergence SARS-CoV-2 dataset and 4% divergence synthetic data, where we use exact matches ($r = 1$).

Execution We ran all benchmarks using PABench (github.com/pairwise-alignment/pa-bench) on Arch Linux on an `Intel Core i7-10750H` with 64GB of memory and 6 cores, with hyper-threading disabled, frequency boost disabled, and CPU power saving features disabled. The CPU frequency is fixed to 3.6GHz and we run 1 single-threaded job at a time with niceness −20. Reported running times are the average wall-clock time per alignment and exclude the time to read data from disk for more stable measurements. For A*PA and A*PA2-full, reported times do include the time to find matches and initialize the heuristic.

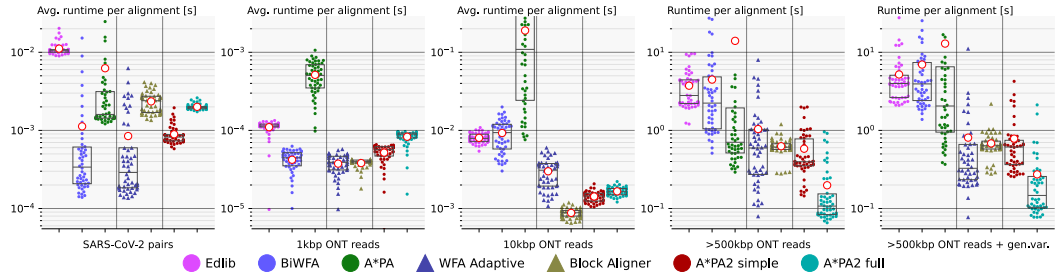


Figure 7 Runtime comparison (log) Each dot shows the running time of a single alignment (right two plots) or the average runtime over 2% of the input pairs (left three plots). Box plots show the three quartiles, and the red circled dot shows the average running time over all alignments. For A*PA, exact matches are used for the SARS-CoV-2 dataset. Some A*PA alignments ≥ 10 kbp time out, and the shown average is a lower bound on the true average. Approximate aligners WFA Adaptive and Block Aligner are indicated with triangles. On the >500 kbp reads, A*PA2-full is 19× faster than other exact methods.

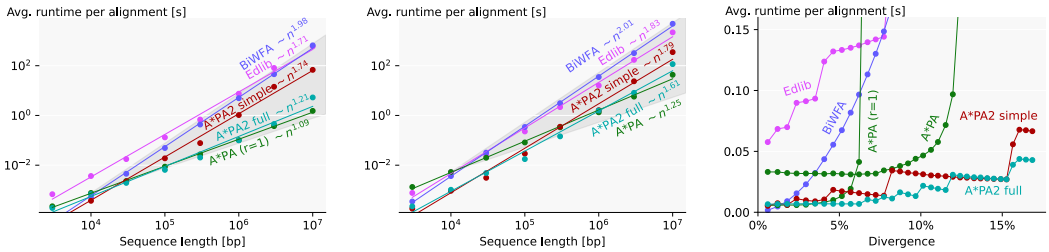
4.2 Comparison with other aligners

Speedup on real data Figure 7 compares the running time of aligners on real datasets. Table 2 in Appendix A.3 contains a corresponding table. On the >500 kbp ONT reads, A*PA2-full is 19× faster than Edlib, BiWFA, and A*PA in average running time, and using the gap-chaining seed heuristic in A*PA2-full provides 3× speedup over A*PA2-simple.

On shorter sequences, the overhead of initializing the heuristic in A*PA2-full is large, and A*PA2-simple is faster. For the 10 kbp dataset, A*PA2-simple is 5.6× faster than other exact methods. For the shortest (1 kbp ONT reads) and most similar sequences (SARS-CoV-2 with 1% divergence), BiWFA is usually faster than Edlib and A*PA2-simple. In these cases, the overhead of using 256 wide blocks is relatively large compared to the edit distance $s \leq 500$ in combination with BiWFA's $O(s^2 + n)$ expected running time.

Comparison with approximate aligners For the smallest datasets, BiWFA is about as fast as the approximate methods WFA Adaptive and Block Aligner, while for the largest datasets A*PA2-full is significantly faster. Only on the dataset of 10 kbp ONT reads is Block Aligner 1.6× faster than A*PA2, but it only finds the correct edit distance for 53% of the alignments. All accuracy numbers can be found in Table 3 in Appendix A.3.

Scaling with length Figures 8a and 8b compare the runtime of aligners on synthetic random sequences of increasing length and constant uniform divergence. BiWFA's runtime is quadratic and is fast for sequences up to 3000 bp. As expected, A*PA2-simple has very similar scaling to Edlib but is faster by a constant factor around 7.5×. A*PA2-full includes the gap-chaining seed heuristic used by A*PA, resulting in comparable speed and near-linear scaling for both of them when $d = 4\%$. For more divergent sequences, A*PA2-full is faster than A*PA since initializing the A*PA heuristic with inexact matches is relatively slow. The reason that A*PA2-full is slower than A*PA for sequences of length 10 Mbp is that A*PA2-full uses shorter seed length $k = 12$ instead of $k = 15$. In most cases, pre-pruning is fast enough to handle the extra matches this causes, but when n approaches $4^{12} \approx 16 \cdot 10^6$, this becomes a bottleneck.



(a) Scaling with length ($d = 4\%$) **(b)** Scaling with length ($d = 12\%$) **(c)** Scaling with divergence

Figure 8 Runtime comparison on synthetic data (a)(b) Log-log plot of average running time of aligners on synthetic sequences of increasing length with 4% divergence and 12% divergence. A*PA uses exact matches (indicated by $r = 1$) for $d = 4\%$ and inexact matches for $d = 12\%$. For sequences of length n, averages are over $10^7/n$ pairs. Lines are fitted in the log-log domain. The region between linear and quadratic growth is shaded in grey. (c) Average running time of aligners over 10 sequences of length 100 kbp with varying uniform divergence.

Scaling with divergence Figure 8c compares the runtime of aligners on synthetic sequences of increasing divergence. BiWFA's runtime grows quadratically, while Edlib grows linearly and jumps up each time another doubling of the threshold is required. A*PA is fast until the maximum potential is reached at 6% resp. 12% and then becomes very slow. A*PA2-simple behaves similar to Edlib and jumps up each time another doubling of the threshold is needed, but is around 8× faster. A*PA2-full outperforms BiWFA for divergence $\geq 2\%$ and A*PA for divergence $\geq 4\%$. The runtime of A*PA2-full is near-constant up to divergence 7% due to the gap-chaining seed heuristic which can correct for up to $1/k = 1/12 = 8.3\%$ of divergence, while A*PA2-simple starts to slow down at lower divergence. The graph of A*PA2 has a negative slope when the number of doublings is fixed, because too low thresholds are rejected more quickly when divergence is higher.

Memory usage of A*PA2 on >500 kbp sequences is at most 200 MB and only 30 MB in median, down from 6868 MB and 158 MB for A*PA. On other datasets and for Edlib and BiWFA, memory usage is at most 11 MB, and usually < 1 MB (Table 4 in Appendix A.3).

4.3 Effects of methods

Figure 9 shows the effect of one-by-one adding improvements to A*PA2 on >500 kbp long sequences, roughly in order of importance, starting with Ukkonen's band doubling method using Myers' bitpacking. Figure 11 in Appendix A.3 instead shows the effect of removing each improvement from the final method. We first change to the $g^*(u) + c_{\text{gap}}(u, v_t)$ computational volume (+A*), making it comparable to Edlib. Then we process blocks of 256 columns at a time and only store differences at block boundaries, giving 2.5× speedup. Adding SIMD gives another 2.7× speedup, and instruction level parallelism (ILP) provides a further small improvement. The diagonal transition traceback (DTT) and sparse heuristic computation do not improve performance of A*PA2-simple much on long sequences, but their removal can be seen to slow it down for shorter sequences in the ablation (Figure 11). Incremental doubling (ID), the gap-chaining seed heuristic (GCSH), pre-pruning (PP), and the pruning of A*PA give another 3× speedup on average and 4× speedup in the first quantile.

Figure 13 in Appendix A.3 shows that A*PA2 typically spends most of its time computing blocks. For short 1 kbp long sequences, half the time is spent on traceback, and for the >500 kbp sequences, A*PA2-full spends around a quarter of time on initializing the heuristic.

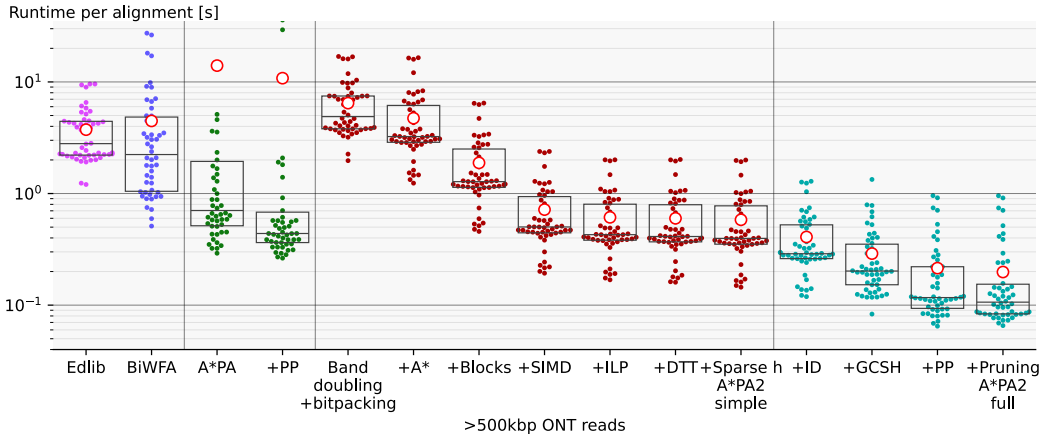


Figure 9 Effect of adding features Box plots showing the performance improvements of A*PA2 on long sequences when incrementally adding new methods one-by-one. A*PA2-simple corresponds to the middle red columns, and A*PA2-full corresponds to the rightmost blue columns.

5 Discussion

We have shown that by incorporating many existing techniques and by writing highly optimized code, A*PA2 achieves 19× speedup over other methods when aligning >500 kbp ONT reads with 6% divergence, 5.6× speedup for sequences of average length 11 kbp, and only a slight slowdown over BiWFA for very short (<1 kbp) and very similar (<2% divergence) sequences. A*PA2's speed is also comparable to approximate aligners, and is faster for long sequences, thereby nearly closing the gap between approximate and exact methods. A*PA2-simple has similar $O(ns)$ scaling behaviour as Edlib in both length and divergence, but with a 6× to 8× better constant. A*PA2-full achieves the best of both: the near-linear scaling with length of A*PA when divergence is small, and the efficiency of Edlib.

Limitations

1. The main limitation of A*PA2-full is that the heuristic requires finding all matches between the two input sequences, which can take long compared to the alignment itself.
2. For sequences with divergence <2%, the diagonal transition algorithm (BiWFA) is very sparse, and computing full blocks in A*PA2 has considerable overhead.
3. The new sequence profile only supports sequences over alphabet size 4, so DNA sequences containing e.g. `N` characters must either be cleaned or fall back to a slower profile.

Future work

1. When divergence is very low (< 1%), the block-based DP is relatively slow, and performance could be improved by using A* with diagonal transition, possibly per block.
2. Currently A*PA2 is completely unaware of the type of sequences it aligns. Using an upper bound on the edit distance, either known or found using a non-exact method, could avoid trying overly large thresholds and smoothen the curve in Figure 8c.
3. It should be possible to extend A*PA2 to semi-global alignment, and to incorporate the ideas of A*PA and A*PA2 back into the Astarix' sequence-to-graph alignment.
4. Extending A*PA2 to affine cost models should also be possible. This will require adjusting the gap-chaining seed heuristic, and changing the computation of the blocks from a bitpacking approach to one of the SIMD-based methods for affine costs.
5. Lastly, TALCO [51] provides an interesting idea: it may be possible start traceback while still computing blocks, thereby saving memory.

Acknowledgements

I am grateful to Pesho Ivanov for many discussions on A*PA. Furthermore, I thank Daniel Liu for discussions, feedback, and suggesting additional related papers, André Kahles, Harun Mustafa, and Gunnar Rätsch for feedback on the manuscript, Andrea Guarracino and Santiago Marco-Sola for sharing the WFA and BiWFA benchmark datasets, and Gary Benson for help with debugging the BitPAl bitpacking code.

References

1 Bowen Alpern, Larry Carter, and Kang Su Gatlin. Microparallelism and high-performance protein matching. In Sidney Karin, editor, *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 24. ACM, 1995. doi:10.1145/224170.224222.

2 Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, Oct 1992. doi:10.1145/135239.135243.

3 Gary Benson, Yözen Hernández, and Joshua Loving. A bit-parallel, general integer-scoring sequence alignment algorithm. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 2013. doi:10.1007/978-3-642-38905-4_7.

4 Gary Benson, Avivit Levy, S. Maimoni, D. Noifeld, and B. Riva Shalom. Lcsk: A refined similarity measure. *Theoretical Computer Science*, 638:11–26, jul 2016. doi:10.1016/j.tcs.2015.11.026.

5 Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–66, Feb 2002. doi:10.1142/s0129054102000947.

6 Jeff Daily. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1), Feb 2016. doi:10.1186/s12859-016-0930-z.

7 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959. doi:10.1007/bf01386390.

8 Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic C++ library for sequence analysis. *BMC Bioinform.*, 9(1), Jan 2008. doi:10.1186/1471-2105-9-11.

9 Michael Farrar. Striped smith-waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, Nov 2007. doi:10.1093/bioinformatics/btl582.

10 James W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179, 1984. doi:10.1093/nar/12.1part1.175.

11 J.T. Frielingsdorf. Improving optimal sequence alignments through a simd-accelerated library. Master's thesis, University of Oslo, 2015. URL: https://bib.irb.hr/datoteka/758607.diplomski_Martin_Sosic.pdf.

12 Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705—-708, Dec 1982. doi:10.1016/0022-2836(82)90398-9.

13 Ragnar Groot Koerkamp and Pesho Ivanov. Exact global alignment using A* with chaining seed heuristic and match pruning. *Bioinformatics*, 40(3), jan 2024. doi:10.1093/bioinformatics/btae032.

14 Frank O. Hadlock. Minimum detour methods for string or sequence comparison. *Congressus Numerantium*, 61:263–274, 1988.

15 Peter W. Harrison, Rodrigo Lopez, Nadim Rahman, Stefan Gutnick Allen, Raheela Aslam, Nicola Buso, Carla A. Cummins, Yasmin Fathy, Eloy Felix, Mihai Glont, Suran Jayathilaka, Sandeep Kadam, Manish Kumar, Katharina B. Lauer, Geetika Malhotra, Abayomi Mosaku, Ossama Edbali, Young Mi Park, Andrew Parton, Matt Pearce, Jose Francisco Estrada pena,

Joseph Rossetto, Craig Russell, Sandeep Selvakumar, Xènia Pérez Sitjà, Alexey Sokolov, Ross Thorne, Marianna Ventouratou, Peter Walter, Galabina Yordanova, Amonida Zadissa, Guy Cochrane, Niklas Blomberg, and Rolf Apweiler. The COVID-19 data portal: accelerating sars-cov-2 and COVID-19 research through rapid open access data sharing. *Nucleic Acids Research*, 49(W1):619–623, may 2021. doi:10.1093/nar/gkab417.

16 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi:10.1109/tssc.1968.300136.

17 Richard Hughey. Parallel hardware for sequence comparison and alignment. *CABIOS*, 12(6):473–479, 1996. doi:10.1093/bioinformatics/12.6.473.

18 Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM Journal of Experimental Algorithmics*, 10, Dec 2005. doi:10.1145/1064546.1180617.

19 Pesho Ivanov, Benjamin Bichsel, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin T. Vechev. Astarix: Fast and optimal sequence-to-graph alignment. In Russell Schwartz, editor, *Research in Computational Molecular Biology - 24th Annual International Conference, RECOMB 2020, Padua, Italy, May 10-13, 2020, Proceedings*, volume 12074 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2020. doi:10.1007/978-3-030-45257-5_7.

20 Pesho Ivanov, Benjamin Bichsel, and Martin T. Vechev. Fast and optimal sequence-to-graph alignment guided by seeds. In Itsik Pe'er, editor, *Research in Computational Molecular Biology - 26th Annual International Conference, RECOMB 2022, San Diego, CA, USA, May 22-25, 2022, Proceedings*, volume 13278 of *Lecture Notes in Computer Science*, pages 306–325. Springer, 2022. doi:10.1007/978-3-031-04749-7_22.

21 Chirag Jain, Daniel Gibney, and Sharma V. Thankachan. Algorithms for colinear chaining with overlaps and gap costs. *Journal of Computational Biology*, 29(11):1237–1251, Nov 2022. doi:10.1089/cmb.2022.0266.

22 Joseph B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201—237, Apr 1983. doi:10.1137/1025045.

23 Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965. URL: https://api.semanticscholar.org/CorpusID:60827152.

24 Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, Mar 2016. doi:10.1093/bioinformatics/btw152.

25 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, May 2018. doi:10.1093/bioinformatics/bty191.

26 Daniel Liu and Martin Steinegger. Block aligner: an adaptive SIMD-accelerated aligner for sequences and position-specific scoring matrices. *Bioinformatics*, 39(8), 08 2023. doi:10.1093/bioinformatics/btad487.

27 Joshua Loving, Yözen Hernández, and Gary Benson. Bitpal: a bit-parallel, general integer-scoring sequence alignment algorithm. *Bioinformatics*, 30(22):3166–3173, Jul 2014. doi:10.1093/bioinformatics/btu507.

28 Santiago Marco-Sola, Jordan M. Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moretó. Optimal gap-affine alignment in $O(s)$ space. *Bioinformatics*, 39(2), feb 2023. doi:10.1093/bioinformatics/btad074.

29 Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa. Fast gap-affine pairwise alignment using the wavefront algorithm. *Bioinformatics*, 37(4):456–463, 09 2020. doi:10.1093/bioinformatics/btaa777.

30 Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, Nov 1986. doi:10.1007/bf01840446.

31 Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, May 1999. doi:10.1145/316542.316550.

32 Gene Myers. Efficient local alignment discovery amongst noisy long reads. In Daniel G. Brown and Burkhard Morgenstern, editors, *Algorithms in Bioinformatics - 14th International Workshop, WABI 2014, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8701 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2014. doi:10.1007/978-3-662-44753-6_5.

33 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, Mar 2001. doi:10.1145/375360.375365.

34 Saul B. Needleman and Christian 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, Mar 1970. doi:10.1016/0022-2836(70)90057-4.

35 Dimitrios P. Papamichail and Georgios P. Papamichail. Improved algorithms for approximate string matching (extended abstract). *BMC Bioinformatics*, 10(S-1), Jan 2009. doi:10.1186/1471-2105-10-s1-s10.

36 Filip Pavetić, Ivan Katanić, Gustav Matula, Goran Žužić, and Mile Šikić. Fast and simple algorithms for computing both LCS_k and LCS_{k+}. 2017. doi:10.48550/arXiv.1705.07279.

37 Filip Pavetić, Goran Žužić, and Mile Šikić. *LCSk++*: practical similarity metric for long strings. 2014. doi:10.48550/arXiv.1407.2407.

38 Torbjørn Rognes. Faster smith-waterman database searches with inter-sequence SIMD parallelisation. *BMC Bioinformatics*, 12(1), Jun 2011. doi:10.1186/1471-2105-12-221.

39 Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16(8):699–706, Aug 2000. doi:10.1093/bioinformatics/16.8.699.

40 David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, Jan 1972. doi:10.1073/pnas.69.1.4.

41 Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, Jun 1974. doi:10.1137/0126070.

42 Haojing Shao and Jue Ruan. Bsalign: A library for nucleotide sequence alignment. *Genomics, Proteomics & Bioinformatics*, mar 2024. doi:10.1093/gpbjnl/qzae025.

43 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195—197, Mar 1981. doi:10.1016/0022-2836(81)90087-5.

44 John L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal on Applied Mathematics*, 49(5):1552—1566, Oct 1989. doi:10.1137/0149094.

45 John L. Spouge. Fast optimal alignment. *CABIOS*, 7(1):1–7, 1991. doi:10.1093/bioinformatics/7.1.1.

46 Hajime Suzuki and Masahiro Kasahara. Introducing difference recurrence relations for faster semi-global alignment of long sequences. *BMC Bioinformatics*, 19-S(1):33–47, feb 2018. doi:10.1186/s12859-018-2014-8.

47 Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 – fast multi-threaded vectorized smith-waterman for ibm cell/b.e. and ×86/sse2. *BMC Research Notes*, 1(1):107, 2008. doi:10.1186/1756-0500-1-107.

48 Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, Jan 1985. doi:10.1016/s0019-9958(85)80046-2.

49 T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52—57, 1968. doi:10.1007/bf01074755.

50 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan 1974. doi:10.1145/321796.321811.

51 Sumit Walia, Cheng Ye, Arkid Bera, Dhruvi Lodhavia, and Yatish Turakhia. TALCO: tiling genome sequence alignment using convergence of traceback pointers. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*, pages 91–107. IEEE, mar 2024. doi:10.1109/hpca57654.2024.00044.

52 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. doi:10.1073/pnas.80.3.726.

53 Andrzej Wozniak. Using video-oriented instructions to speed up sequence comparison. *CABIOS*, 13(2):145–150, 1997. doi:10.1093/bioinformatics/13.2.145.

54 Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, Oct 1992. doi:10.1145/135239.135244.

55 Sun Wu, Udi Manber, Gene Myers, and Webb Miller. An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, Sep 1990. doi:10.1016/0020-0190(90)90035-v.

56 Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth. SSW library: An SIMD smith-waterman C/C++ library for use in genomic applications. *PLoS ONE*, 8(12), Dec 2013. doi:10.1371/journal.pone.0082138.

57 M. Šošic. An SIMD dynamic programming C/C++ library. Master's thesis, University of Zagreb, 2015. URL: https://bib.irb.hr/datoteka/758607.diplomski_Martin_Sosic.pdf.

58 Martin Šošić and Mile Šikić. Edlib: a C/C ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, Jan 2017. doi:10.1093/bioinformatics/btw753.

A Appendix

A.1 Bitpacking

Figure 10a shows a SIMD version of Myers' bitpacking and Figure 10b shows the method for edit distance described in the supplement of BitPAl [27]. Both methods require 20 instructions.

Both methods are usually reported to use fewer than 20 instructions, but this excludes the shifting out of the bottom horizontal difference (four instructions) and the initialization of the carry for BitPAl (one operation). We require these additional outputs/inputs since we want to align multiple 64bit lanes below each other, and the horizontal difference in between must be carried through.

```rust
pub fn compute_block_simd_myers(
    // 0 or 1. Indicates -1 difference on top.
    hp0: &mut Simd<u64, 4>,
    // 0 or 1. Indicates -1 difference on top.
    hm0: &mut Simd<u64, 4>,
    // 64-bit indicator of +1 differences on left.
    vp: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 differences on left.
    vm: &mut Simd<u64, 4>,
    // 64-bit indicator of chars equal to top char.
    eq: Simd<u64, 4>,
) {
    let vx = eq | *vm;
    let eq = eq | *hm0;
    // The addition carries information between rows.
    let hx = (((eq & *vp) + *vp) ^ *vp) | eq;
    let hp = *vm | !(hx | *vp);
    let hm = *vp & hx;
    // Extract the high bit as bottom difference.
    let right_shift = Simd::<u64, 4>::splat(63);
    let hpw = hp >> right_shift;
    let hmw = hm >> right_shift;
    // Insert the top horizontal difference.
    let left_shift = Simd::<u64, 4>::splat(1);
    let hp = (hp << left_shift) | *hp0;
    let hm = (hm << left_shift) | *hm0;
    // Update the input-output parameters.
    *hp0 = hpw;
    *hm0 = hmw;
    *vp = hm | !(vx | hp);
    *vm = hp & vx;
}
```

(a) Myers' bitpacking

```rust
pub fn compute_block_simd_bitpal(
    // 0 or 1. Indicates 0 difference on top.
    hz0: &mut Simd<u64, 4>,
    // 0 or 1. Indicates -1 difference on top.
    hp0: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 differences on left.
    vm: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 and 0 differences on left.
    vmz: &mut Simd<u64, 4>,
    // 64-bit indicator of chars equal to top char.
    eq: Simd<u64, 4>,
) {
    let eq = eq | *vm;
    let ris = !eq;
    let notmi = ris | *vmz;
    let carry = *hp0 | *hz0;
    // The addition carries info between rows.
    let masksum = (notmi + *vmz + carry) & ris;
    let hz = masksum ^ notmi ^ *vm;
    let hp = *vm | (masksum & *vmz);
    // Extract the high bit as bottom difference.
    let right_shift = Simd::<u64, 4>::splat(63);
    let hzw = hz >> right_shift;
    let hpw = hp >> right_shift;
    // Insert the top horizontal difference.
    let left_shift = Simd::<u64, 4>::splat(1);
    let hz = (hz << left_shift) | *hz0;
    let hp = (hp << left_shift) | *hp0;
    // Update the input-output parameters.
    *hz0 = hzw;
    *hp0 = hpw;
    *vm = eq & hp;
    *vmz = hp | (eq & hz);
}
```

(b) Bitpal's bitpacking

Figure 10 Bitpacking Rust code for SIMD version of Myers' and Bitpal's bitpacking algorithms that both take 20 instructions.

A.2 Sparse heuristic invocation

Here we explain how to compute the fixed range and the range of rows to compute with significantly fewer evaluations of the heuristic than the simpler method of Section 3.7. We first state two very similar lemmas, and then give the updated steps.

▶ **Lemma 1.** *When h is admissible and $f(u) > t + 2D$, then $f^*(u') > t$ for all u' within distance $d(u, u') \leq D$ from u.*

Proof. Since adjacent states differ in distance by $\{-1, 0, +1\}$, we have $g(u') \geq g(u) - d(u, u') \geq g(u) - D$ and $h^*(u') \geq h^*(u) - d(u, u') \geq h^*(u) - D$. Now suppose that $f^*(u') \leq t$. Then u' is fixed and we have $g(u') = g^*(u')$, and since h is admissible $h(u) \leq h^*(u)$. Thus:

$$\begin{aligned} t + 2D < f(u) &= g(u) + h(u) \\ &\leq g(u) + h^*(u) \leq g(u') + h^*(u') + 2D \\ &= g^*(u') + h^*(u') + 2D = f^*(u') + 2D \leq t + 2D. \end{aligned}$$

This is a contradiction, so we must have $f^*(u') > t$, as required. ◀

▶ **Lemma 2.** *When h is admissible, v is below the diagonal of a computed state u, and $f_l(v) = g^*(u) + c_{\text{gap}}(u, v) + h(v) > t + 2D$, then $f^*(v') > t$ when v has distance $d(v, v') \leq D$ from u.*

Proof. We have $c_{\text{gap}}(u, v') \geq c_{\text{gap}}(u, v) - d(v, v') \geq c_{\text{gap}}(u, v) - D$, and $h^*(v') \geq h^*(v) - D$. Now suppose that $f^*(v') \leq t$. Then we know that $g^*(v) \geq g^*(u) + c_{\text{gap}}(u, v)$, and we still have $h(v) \leq h^*(v)$, $g^*(v') \geq g^*(v) - D$, and $h^*(v') \geq h^*(v) - D$. It follows that

$$\begin{aligned} t + 2D < f_l(v) &= g^*(u) + c_{\text{gap}}(u, v) + h(v) \\ &\leq g^*(v) + h^*(v) \\ &\leq g^*(v') + h^*(v') + 2D = f^*(v') \leq t + 2D, \end{aligned}$$

which is a contradiction, so we conclude that $f^*(v') > t$, as required. ◀

Step 1': Sparse fixed range To find the first row j_{start} with $f(\langle i, j_{start}\rangle) \leq t$, start with $j = r_{start}$, and increment j by $\lceil (f(v) - t)/2 \rceil$ as long as $f(v) > t$, since none of the intermediate states can lie on a path of length $\leq t$ by Lemma 1. The last row is found in the same way by going up from r_{end}. As seen in Figure 5b, this sparse variant significantly reduces the number of evaluations of the heuristic in the right-most columns of each block.

Step 2': Sparse end of computed range Instead of considering one column at a time, we now first make a big jump down and then jump to the right.

1. Start with $v = \langle i', j'\rangle = u + \langle 1, B + 1\rangle = \langle i + 1, j_{end} + B + 1\rangle$.
2. If $f_l(v) \leq t$, increase j' (go down) by 8.
3. If $f_l(v) > t$, increase i' (go right) by $\lceil (f_l(v) - t)/2 \rceil$, but do not exceed column $i + B$.
4. Repeat from step 2, until $i' = i + B$.
5. While $f_l(v) > t$, decrease j' (go up) by $\lceil (f_l(v) - t)/2 \rceil$, but do not go above the diagonal of u.

The resulting v is again the bottommost state in column $i + B$ that can potentially have $f(t) \leq t$, and its row is the last row that will be computed.

A.3 Further results

See Figures 11–14 and Tables 1–4.

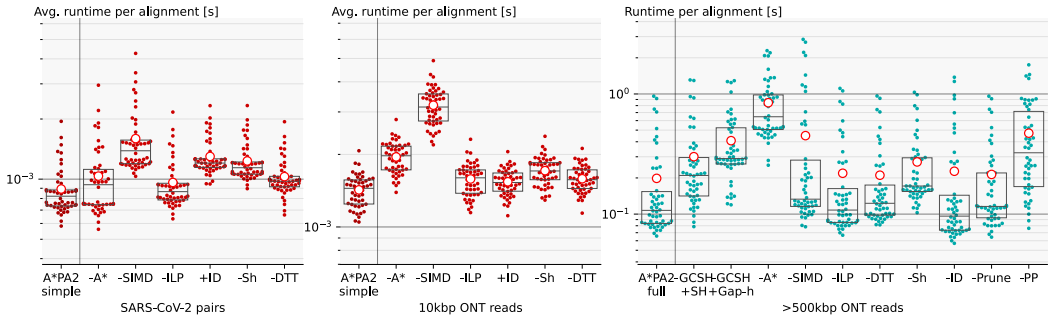


Figure 11 Ablation Box plots showing how the performance of A*PA2-simple (left, middle) and A*PA2-full (right) changes when removing (-) or adding (+) features. ILP: instruction level parallelims, ID: incremental doubling, Sh: sparse heuristic evaluation, DTT: diagonal transition traceback, PP: pre-pruning. Note that adding incremental doubling to A*PA2-simple slows down simple alignments, while A*PA2-full benefits from it for larger alignments.

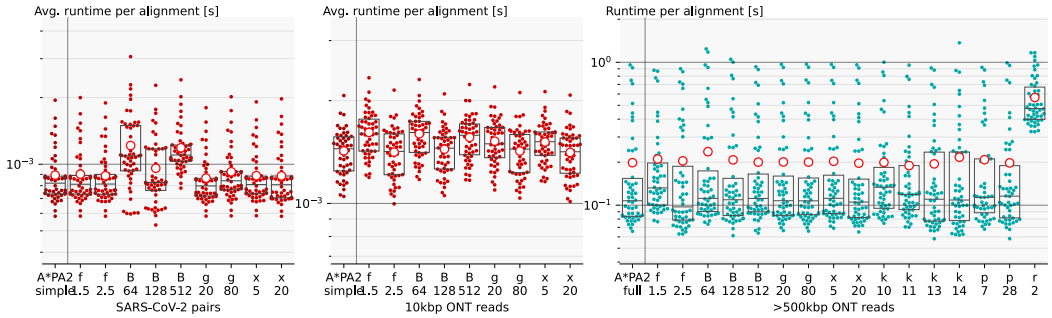


Figure 12 Changing parameters Runtime of A*PA2-simple (left, middle) and A*PA2-full (right) with one parameter modified. Default parameters are seed length $k = 12$, pre-pruning look-ahead $p = 14$, growth factor $f = 2$, block size $B = 256$, max DTT traceback cost $g = 40$, and dropping diagonals that lag $x = 10$ behind during traceback. Running time is not very sensitive with regards to most parameters. Of note are using inexact matches ($r = 2$) for the heuristic, which take significantly longer to find, larger seed length k, which decreases the strength of the heuristic, and smaller block sizes ($B = 128$ and $B = 64$), which induce more overhead.

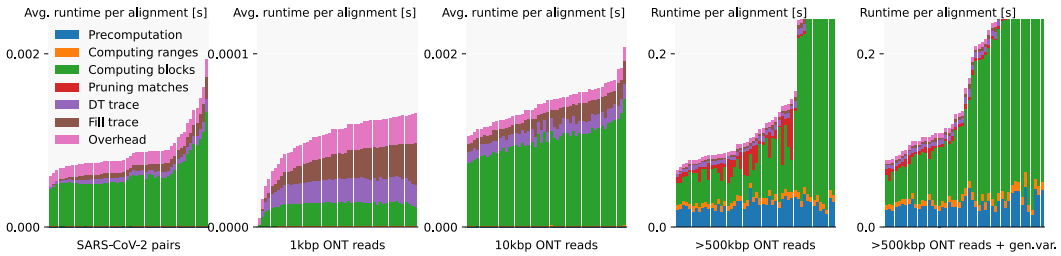


Figure 13 Runtime profile of A*PA2 using A*PA2-simple for short sequences and A*PA2-full for the two rightmost >500 kbp datasets. Each column corresponds to a (set of) alignment(s), which are sorted by total runtime. *Overhead* is the part of the runtime not measured in one of the other parts and includes the time to build the profile. For >500 kbp long sequences, A*PA2-full spends most of its time computing blocks, followed by the initialization of the heuristic. For very short sequences of 1 kbp, up to half the time is spent on tracing an optimal alignment.

			Length [kbp]			Divergence [%]			Max gap [kbp]	
Dataset	**Source**	**Cnt**	min	mean	max	min	mean	max	mean	max
SARS-CoV-2	A*PA2	10 000	27	30	30	0.0	**1.5**	12.8	0.1	1.0
ONT 1 kbp	WFA	12 500	0.04	0.8	1.1	0.0	10.4	22.5	0.01	0.1
ONT 10 kbp	BiWFA	10 000	0.2	11	50	3.0	11.6	19.2	0.07	3.4
ONT >500 kbp	A*PA	50	500	594	849	2.7	6.1	16.7	0.1	1.3
ONT >500 kbp + gv	BiWFA	48	502	632	1053	4.3	7.2	18.2	**1.9**	42

Table 1 Dataset statistics All but the first dataset are ONT reads. The dataset with genetic variation (gv) also includes long gaps, while the SARS-CoV-2 dataset stands out for having only 1.5% divergence on average. **Cnt**: number of sequence pairs. **Max gap**: longest gap in the reconstructed alignment.

	SARS-CoV-2	ONT			
Aligner	[ms]	1 kbp [ms]	10 kbp [ms]	>500 kbp [s]	>500 kbp + gv [s]
Edlib	11.14	0.110	8.0	3.74	5.20
BiWFA	1.13	0.042	9.3	4.47	6.96
A*PA	6.25	0.514	>190.1	>14.01	>12.92
WFA-Adaptive	0.85	0.038	3.0	1.04	0.81
Block Aligner	2.35	0.038	0.9	0.63	0.68
A*PA2-simple	0.89	0.052	1.4	0.58	0.78
A*PA2-full	2.00	0.083	1.7	0.20	0.27
Speedup [×]	1.3	0.81	5.6	18.8	19.0

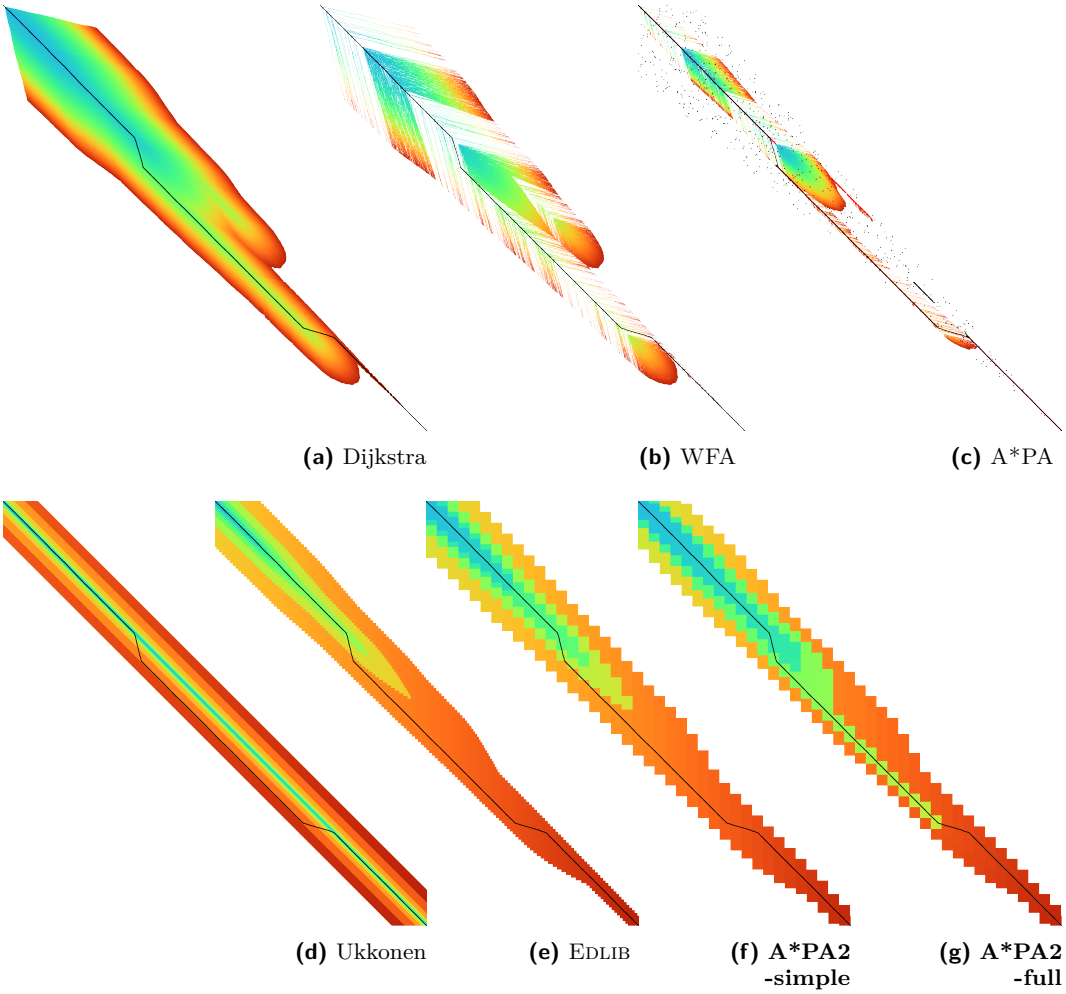
Table 2 Average runtime per sequence of each aligner on each dataset. Cells marked with > are a lower bound due to timeouts. Speedup is reported as the fastest A*PA2 variant compared to the fastest of Edlib, BiWFA, and A*PA. WFA-Adaptive and Block Aligner are approximate aligners.

	SARS-CoV-2	ONT			
Aligner		1 kbp	10 kbp	>500 kbp	>500 kbp + gv
WFA-Adaptive	92%	93%	49%	60%	4%
Block Aligner	34%	85%	53%	96%	50%

Table 3 Percentage of correctly aligned reads by approximate aligners. The accuracy of WFA-Adaptive drops a lot for the >500 kbp dataset with genetic variation, since these alignments contain gaps of thousands of basepairs, much larger than the 50 bp cutoff after which trailing diagonals are dropped.

Memory [MB]	SARS-CoV-2		ONT							
			1 kbp		10 kbp		>500 kbp		>500 kbp + gv	
Aligner	median	max	median	max	median	max	median	max	median	max
Edlib	0	0	0	0	0	0	0	0	0	0
BiWFA	0	0	0	0	0	0	4	11	0	2
A*PA	0	236	0	0	228	873	84	3453	158	6868
WFA-Adaptive	0	11	0	0	0	0	0	0	0	0
Block Aligner	0	16	0	0	0	3	583	1189	610	2171
A*PA2 simple	2	5	0	0	4	6	0	55	2	164
A*PA2 full	0	0	0	0	0	0	30	82	6	141

Table 4 Memory usage of aligners, measured as the increase in `max_rss` before and after aligning a pair of sequences.



(a) Dijkstra **(b)** WFA **(c)** A*PA

(d) Ukkonen **(e)** Edlib **(f) A*PA2 -simple** **(g) A*PA2 -full**

Figure 14 Alignment of two sequences of length 10 kbp with 17% divergence using the same methods as in Figure 1. The optimal alignment contains similar regions, noisy regions, indels, and repeats (shown by the black matches and red pruned matches in (c)). A*PA computes a subset of the states of WFA. A*PA2-full computes more states than A*PA, but is more efficient.