Exact global alignment using A* with seed heuristic and match pruning

Ragnar Groot Koerkamp *,† **and Pesho Ivanov** *,†

Department of Computer Science, ETH Zurich, Switzerland

*To whom correspondence should be addressed.
†These authors contributed equally to this work.

Abstract

Motivation. Sequence alignment has been a core problem in computational biology for the last half-century. It is an open problem whether exact pairwise alignment is possible in linear time for related sequences (Medvedev, 2022b).
Methods. We solve exact global pairwise alignment with respect to edit distance by using the A* shortest path algorithm on the edit graph. In order to efficiently align long sequences with high error rate, we extend the *seed heuristic* for A* (Ivanov *et al.*, 2022) with *match chaining*, *inexact matches*, and the novel *match pruning* optimization. We prove the correctness of our algorithm and provide an efficient implementation in A*PA.
Results. We evaluate A*PA on synthetic data (random sequences of length n with uniform mutations with error rate e) and on real long ONT reads of human data. On the synthetic data with $e{=}5\%$ and $n{\leq}10^7\,\mathrm{bp}$, A*PA exhibits a near-linear empirical runtime scaling of $n^{1.08}$ and achieves $>250\times$ speedup compared to the leading exact aligners EDLIB and BIWFA. Even for a high error rate of $e{=}15\%$, the empirical scaling is $n^{1.28}$ for $n{\leq}10^7\,\mathrm{bp}$. On two real datasets, A*PA is the fastest aligner for 58% of the alignments when the reads contain only sequencing errors, and for 17% of the alignments when the reads also include biological variation.
Availability. github.com/RagnarGrootKoerkamp/astar-pairwise-aligner
Contact. ragnar.grootkoerkamp@inf.ethz.ch, pesho@inf.ethz.ch

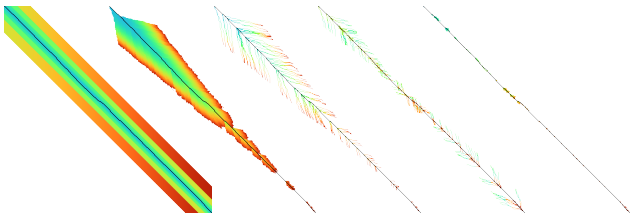
1 Introduction

The problem of aligning one biological sequence to another has been formulated over half a century ago (Needleman and Wunsch, 1970) and is known as *global pairwise alignment* (Navarro, 2001). Pairwise alignment has numerous applications in computational biology, such as genome assembly, read mapping, variant detection, multiple sequence alignment, and differential expression (Prjibelski *et al.*, 2018). Despite the centrality and age of pairwise alignment, "a major open problem is to implement an algorithm with linear-like empirical scaling on inputs where the edit distance is linear in n" (Medvedev, 2022b).

Alignment accuracy affects the subsequent analyses, so a common goal is to find a shortest sequence of edit operations (insertions, deletions, and substitutions of single letters) that transforms one sequence into the other. Finding such a sequence of operations is at least as hard as computing the *edit distance*, which has recently been proven to not be computable in strongly subquadratic time, unless SETH is false (Backurs and Indyk, 2015). Given that the number of sequencing errors is proportional to the length, existing exact aligners are limited by quadratic scaling not only in the worst case but also in practice. This is a computational bottleneck given the growing amounts of biological data and the increasing sequence lengths (Kucherov, 2019).

1.1 Related work on alignment

We outline algorithms for exact pairwise alignment and their fastest implementations for biological sequences. Refer to Kucherov (2019) for approximate, probabilistic, and non-edit distance algorithms and aligners.



(a)	(b)	(c)	(d)	(e)
Exponential band (EDLIB)	Dijkstra	DT (WFA)	DT+D&C (BIWFA)	**This work (A*PA)**

Fig. 1. Demonstration of the computed states by various optimal alignment algorithms and corresponding aligners that implement them on synthetic data (length $n{=}500\,\mathrm{bp}$, error rate $e{=}20\%$). Blue-to-red coloring indicates the order of computation. (a) Exponential banding algorithm (EDLIB), (b) Dijkstra, (c) Diagonal transition/DT (WFA), (d) Diagonal transition with divide-and-conquer/D&C (BIWFA), (e) A* with chaining seed heuristic and match pruning (seed length $k{=}5$ and exact matches).

Dynamic programming. The standard approach to sequence alignment is by successively aligning prefixes of the first sequence to prefixes of the second. Vintsyuk (1968) was the first to introduce this $O(nm)$ dynamic programming (DP) approach for a comparing a pair speech signals with n and m elements. Independently, it was applied to compute edit distance for biological sequences (Needleman and Wunsch, 1970; Sankoff, 1972; Sellers, 1974; Wagner and Fischer, 1974). This well-known algorithm is implemented in modern aligners like SEQAN (Reinert *et al.*, 2017) and PARASAIL (Daily, 2016). Improving the performance of these quadratic algorithms has been a central goal in later works.

Banding and bit-parallelization. When similar sequences are being aligned, the whole DP table may not need to be computed. One such output-sensitive algorithm is the *banded* algorithm of Ukkonen (1985) (Fig. 1a) which considers only states near the diagonal within an exponentially increasing *band*, and runs in $O(ns)$ time, where s is the edit distance between the sequences. This algorithm, combined with the *bit-parallel optimization* by Myers (1999) is implemented by the EDLIB aligner (Šošić and Šikić, 2017) that runs in $O(ns/w)$ runtime, where w is the machine word size (nowadays 32 or 64).

Diagonal transition and WFA. The $O(ns)$ runtime complexity can be improved using the algorithm independently discovered by Ukkonen (1985) and Myers (1986) that is known as *diagonal transition* (Navarro, 2001) (Fig. 1c). It has an $O(ns)$ runtime in the worst-case but only takes expected $O(n + s^2)$ time under the assumption that the input sequences are random (Myers, 1986). This algorithm has been extended to linear and affine costs in the *wavefront alignment* (WFA) algorithm (Marco-Sola *et al.*, 2021) in a way similar to Gotoh (1982), and has been improved to only require linear memory in BIWFA (Marco-Sola *et al.*, 2022) by combining it with the *divide and conquer* approach of Hirschberg (1975), similar to Myers (1986) algorithm for unit edit costs (Fig. 1d). Note that when each sequence letter has an error with a constant probability, the total number of errors s is proportional to n, so that even these algorithms have a quadratic runtime.

Shortest paths and A*. A pairwise alignment that minimizes edit distance corresponds to a shortest path in the *edit graph* (Vintsyuk, 1968; Ukkonen, 1985). Assuming non-negative edit costs, a shortest path can be found using Dijkstra's algorithm (Ukkonen, 1985) (Fig. 1b) or A* (Spouge, 1989). A* is an informed search algorithm which uses a task-specific heuristic function to direct its search. Depending on the heuristic function, a shortest path may be found significantly faster than by an uninformed search such as Dijkstra's algorithm. In the context of semi-global sequence-to-graph alignment, A* has been used to empirically scale sublinearly with the reference size for short reads (Ivanov *et al.*, 2020).

Seeds and matches. Seed-and-extend is a commonly used paradigm for solving semi-global alignment approximately (Kucherov, 2019). Seeds are also used to define and compute LCSk (Benson *et al.*, 2014), a generalization of longest common subsequence (LCS). In contrast, the *seed heuristic* by Ivanov *et al.* (2022) speeds up finding an optimal alignment by using seed matches to speed up the A* search. The seed heuristic enables empirical near-linear scaling to long HiFi reads (up to 30 kbp with 0.3% errors) (Ivanov *et al.*, 2022). A limitation of the existing seed heuristic is the low tolerance to increasing error rates due to using only long exact matches without accounting for their order.

1.2 Contributions

Our algorithm exactly solves global pairwise alignment for edit distance costs, also known as *Levenshtein distance* (Levenshtein *et al.*, 1966). It uses the A* algorithm to find a shortest path in the edit graph.

Seed heuristic. In order to handle higher error rates, we extend the *seed heuristic* (Ivanov *et al.*, 2022) to *inexact matches*, allowing up to 1 error in each match. To handle cases with a large number of seed matches we introduce *match chaining*, constraining the order in which seed matches can be linked (Wilbur and Lipman, 1984; Benson *et al.*, 2016). We prove that our *chaining seed heuristic* with inexact matches is admissible, which guarantees that A* finds a shortest path.

Match pruning. In order to reduce the number of states expanded by A* we apply the *multiple-path pruning* observation of Poole and Mackworth (2017): once a shortest path to a vertex has been found, no other paths to this vertex can improve the global shortest path. We prove that when a state at the start of a match is expanded, a shortest path to this state has been found. Since no other path to this state can be shorter, we show that we can *prune* (remove) the match, thus improving the seed heuristic. This incremental heuristic search has some similarities to Real-time Adaptive A* (Koenig and Likhachev, 2006).

Implementation. We efficiently implement our algorithm in the A*PA aligner. In particular, we use *contours* (Hirschberg, 1977; Hunt and Szymanski, 1977; Pavetić *et al.*, 2017) to efficiently to compute the chaining seed heuristic, and update them when pruning matches.

Scaling and performance. We compare the scaling and performance of our algorithm to other exact aligners on synthetic data, consisting of random genetic sequences with up to 15% uniform errors and up to 10^7 bases. We demonstrate that inexact matches and match chaining enable scaling to higher error rates, while match pruning enables near-linear scaling with length by reducing the number of expanded states to not much more than the best path (Fig. 1e). Our empirical results show that for e=5% and n=10^7 bp, A*PA outperforms the leading aligners EDLIB (Šošić and Šikić, 2017) and BIWFA (Marco-Sola *et al.*, 2022) by more than 250 times.

We demonstrate a limited applicability of our algorithm to long Oxford Nanopore (ONT) reads from human samples. A*PA is the fastest exact aligner on 58% of the alignments on a dataset with only sequencing errors, and on 17% of the alignments on a dataset with biological variation.

2 Preliminaries

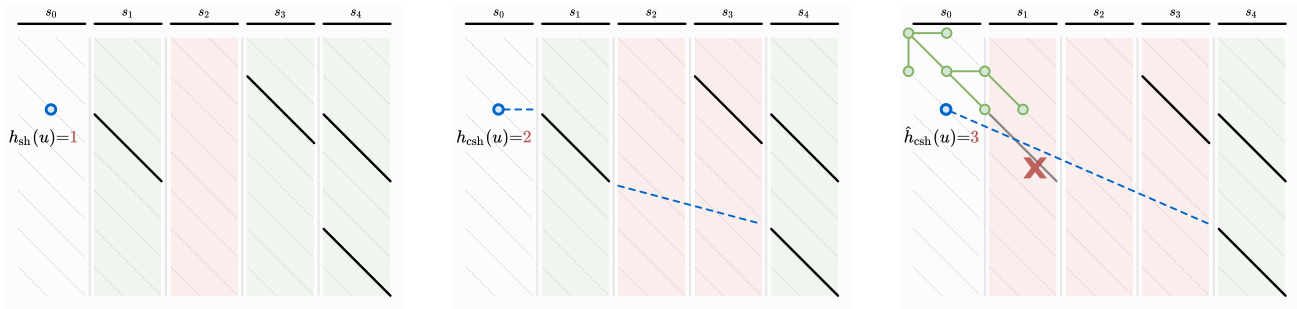
This section provides background definitions which are used throughout the paper.

Sequences. The input sequences $A = \overline{a_0 a_1 \dots a_i \dots a_{n-1}}$ and $B = \overline{b_0 b_1 \dots b_j \dots b_{m-1}}$ are over an alphabet Σ with 4 letters. We refer to substrings $\overline{a_i \dots a_{i'-1}}$ as $A_{i \dots i'}$, to prefixes $\overline{a_0 \dots a_{i-1}}$ as $A_{<i}$, and to suffixes $\overline{a_i \dots a_{n-1}}$ as $A_{\geq i}$. The *edit distance* $\mathrm{ed}(A, B)$ is the minimum number of insertions, deletions, and substitutions of single letters needed to convert A into B.

Edit graph. Let *state* $\langle i, j\rangle$ denote the subtask of aligning the prefix $A_{<i}$ to the prefix $B_{<j}$. The *edit graph* (also called *alignment graph*) $G(V, E)$ is a weighted directed graph with vertices $V = \{\langle i, j\rangle | 0 \leq i \leq n, 0 \leq j \leq m\}$ corresponding to all states, and edges connecting tasks to subtasks: edge $\langle i, j\rangle \to \langle i{+}1, j{+}1\rangle$ has cost 0 if $a_i = b_j$ (match) and 1 otherwise (substitution), and edges $\langle i, j\rangle \to \langle i{+}1, j\rangle$ (deletion) and $\langle i, j\rangle \to \langle i, j{+}1\rangle$ (insertion) have cost 1. We denote the root state $\langle 0, 0\rangle$ by v_s and the target state $\langle n, m\rangle$ by v_t. For brevity we write $f(\langle i, j\rangle)$ as $f\langle i, j\rangle$. The edit graph is a natural representation of the alignment problem that provides a base for all alignment algorithms.

Paths, alignments, seeds and matches. Any path from $\langle i, j\rangle$ to $\langle i', j'\rangle$ in the edit graph G represents a *pairwise alignment* (or just *alignment*) of the substrings $A_{i \dots i'}$ and $B_{j \dots j'}$. We denote with $d(u, v)$ the distance between states u and v. A shortest path π^* corresponds to an optimal alignment, thus $\mathrm{cost}(\pi^*) = d(v_s, v_t) = \mathrm{ed}(A, B)$. For a state u we write $g^*(u) := d(v_s, u)$ and $h^*(u) := d(u, v_t)$ for the distance from the start to u and from u to the target v_t, respectively. We define a *seed* as a substring $A_{i \dots i'}$ of A, and an *exact match* of a seed as an alignment of the seed of cost 0.

Dijkstra and A*. Dijkstra's algorithm (Dijkstra, 1959) finds a shortest path from v_s to v_t by *expanding* vertices in order of increasing distance $g^*(u)$ from the start. The A* algorithm (Hart *et al.*, 1968; Pearl, 1984),



(a) Seed heuristic (b) Chaining seed heuristic (c) Chaining seed heuristic + match pruning

Fig. 2. A demonstration of the seed heuristic, chaining seed heuristic, and match pruning. Sequence A is split into 5 seeds drawn as horizontal black segments (—) on top. Their exact matches in B are drawn as diagonal black segments (\). The heuristic is evaluated at the blue state (u, ○), based on the 4 remaining seeds. The dashed blue paths show maximal length chains of seed matches (green columns, ▮). The seeds that are not matched (red columns, ▮) count towards the heuristic. (a) The seed heuristic $h_{\mathrm{sh}}(u) = 1$ is the number of remaining seeds that do not have matches (s_2). (b) The chaining seed heuristic $h_{\mathrm{csh}}(u) = 2$ is the number of not matched remaining seeds (s_2 and s_3) for a path going only down and to the right containing a maximal number of matches. (c) Once the start of a match is expanded (shown as green circles, ●), the match is *pruned* (marked with red cross, ✗), and future computations of the heuristic ignore it. This reduces the maximum chain of matches starting at u by 1 (s_1) so that $\hat{h}_{\mathrm{csh}}(u)$ increases by 1.

instead, directs the search towards a target by expanding vertices in order of increasing $f(u) := g(u) + h(u)$, where $h(u)$ is a heuristic function that estimates the distance $h^*(u)$ to the end and $g(u)$ is the shortest length of a path from v_s to u found so far. A heuristic is *admissible* if it is a lower bound on the remaining distance, $h(u) \leq h^*(u)$, which guarantees that A⋆ has found a shortest path as soon as it expands v_t. Heuristic h_1 *dominates* another heuristic h_2 when $h_1(u) \geq h_2(u)$ for all vertices u. A dominant heuristic will usually, but not always (Holte, 2010), expand less vertices. Note that Dijkstra's algorithm is equivalent to A⋆ using a heuristic that is always 0, and that both algorithms require non-negative edge costs. Our variant of the A⋆ algorithm is provided in Section 4.1.

Chains. A state $u = \langle i, j\rangle \in V$ *precedes* a state $v = \langle i', j'\rangle \in V$, denoted $u \preceq v$, when $i \leq i'$ and $j \leq j'$. Similarly, a match m precedes a match m', denoted $m \preceq m'$, when the end of m precedes the start of m'. This makes the set of matches a partially ordered set. A state u precedes a match m (denoted $u \preceq m$) when it precedes the start of the match. A *chain* of matches is a (possibly empty) sequence of matches $m_1 \preceq \cdots \preceq m_l$.

Contours. To efficiently calculate maximal chains of matches, *contours* are used. Given a set of matches $\mathcal{M}$, $S(u)$ is the number of matches in the longest chain $u \preceq m_0 \preceq \ldots$, starting at u. The function $S\langle i, j\rangle$ is non-increasing in both i and j. *Contours* are the boundaries between regions of states with $S(u) = \ell$ and $S(u) < \ell$ (see Fig. 3). Note that contour ℓ is completely determined by the set of matches $m \in \mathcal{M}$ for which $S(\mathrm{start}(m)) = \ell$ (Hirschberg, 1977). Hunt and Szymanski (1977) give an algorithm to efficiently compute S when $\mathcal{M}$ is the set of single-letter matches between A and B, and Deorowicz and Grabowski (2014) give an algorithm when $\mathcal{M}$ is the set of k-mer exact matches.

3 Methods

In this section we define the *seed heuristic* with *inexact matches* and *chaining*, and *match pruning* (see Fig. 2). We motivate each of these extensions intuitively, define them formally, and prove that A⋆ is guaranteed to find a shortest path. We end with a reformulation of our heuristic definitions that allows for more efficient computation. Appendix A.9 contains a summary of the notation we use.

3.1 Overview

In order to find a minimal cost alignment of A and B, we use the A⋆ algorithm to find a shortest path from the starting state $v_s{=}\langle 0, 0\rangle$ to the target state $v_t{=}\langle n, m\rangle$ in the edit graph. We present two heuristic functions, the seed heuristic and the chaining seed heuristic, and prove that they are admissible, so that A⋆ always finds a shortest path.

To define the seed heuristic h_{sh}, we split A into short, non-overlapping substrings (*seeds*) of fixed length k. Since the whole of sequence A has to be aligned, each of the seeds also has to be aligned somewhere. If a seed cannot be *exactly* matched in B without mistakes, then at least one edit has to be made to align it. We first compute all positions in B where each seed from A matches exactly. Then, a lower bound on the edit distance between the remaining suffixes $A_{\geq i}$ and $B_{\geq j}$ is given by the number of seeds entirely contained in $A_{\geq i}$ that do not match anywhere in B. An example is shown in Fig. 2a.

We improve the seed heuristic by enforcing that the seed matches occur in the same order as their seeds occur in A, i.e., they form a *chain*. Now, the number of upcoming errors is at least the minimal number of remaining seeds that cannot be aligned on a single path to the target. The chaining seed heuristic h_{csh} equals the number of remaining seeds minus the maximum length of a chain of matches, see Fig. 2b.

To scale to larger error rates, we generalize both heuristics to use *inexact matches*. For each seed from A, our algorithm finds all its inexact matches in B with cost at most 1. Then, not using any match of a seed will require at least $r{=}2$ edits for the seed to be eventually aligned.

Finally, we use *match pruning*: once we find a shortest path to a state v, no shorter path to that state is possible. Since we are only looking for a single shortest path, we may ignore any other path going to v. In particular, when we are evaluating the heuristic h in some state u preceding v, and v is at the start of a match, all paths containing the match are excluded, and thus we may simply ignore (*prune*) this match for future computations of the heuristic. This increases the value of the heuristic, as shown in Fig. 2c, and leads to a significant reduction of the number of states being expanded by the A⋆.

3.2 Seed heuristic and chaining seed heuristic

We start with the formal definitions of seeds and matches that are the basis of our heuristic functions and algorithms.

Seeds. We split the sequence A into a set of consecutive non-overlapping substrings (*seeds*) $\mathcal{S} = \{s_0, s_1, s_2, \dots s_{\lfloor n/k \rfloor - 1}\}$, such that each seed $s_l = A_{lk \dots lk+k}$ has length k. After aligning the first i letters of A, the information for the heuristic will come from the *remaining* seeds $\mathcal{S}_{\geq i} := \{s_l \in \mathcal{S} \mid lk \geq i\}$ contained in the suffix $A_{\geq i}$.

Seed alignment. An *alignment* of a seed $s = A_{i \dots i+k}$ is a path π_s in the edit graph G from a state $u{=}\langle i, j\rangle$ to a state $v{=}\langle i{+}k, j'\rangle$, where i is the start and $i{+}k$ is the end of s. We refer to the cost of the alignment π_s as $\mathrm{cost}(\pi_s)$.

Matches. In order to only consider good alignments of seeds, we fix a threshold cost r called the *seed potential*. We define a *match* m as an alignment of a seed with cost $\mathrm{cost}(m) < r$. For each seed s, the set of all of its matches is $\mathcal{M}_s$. The inequality is strict so that $\mathcal{M}_s = \emptyset$ implies that aligning the seed will incur cost at least r. Let $\mathcal{M} = \bigcup_s \mathcal{M}_s$ denote the set of all matches. With $r{=}1$ we allow only *exact* matches, while for $r{=}2$ we allow both exact and *inexact* matches with one edit. In this paper we do not consider larger r.

Next, we define two heuristic functions:

DEFINITION 1 (Seed heuristic). *Given a set of matches $\mathcal{M}$ with costs less than r, the* seed heuristic $h^{\mathcal{M}}_{\mathrm{sh}}$ *in a state $u{=}\langle i, j\rangle$ is the minimal total cost to independently align all remaining seeds: each seed is either matched or incurs cost r if it cannot be matched in $\mathcal{M}$:*

$$h^{\mathcal{M}}_{\mathrm{sh}}(u) := \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \min_{m \in \mathcal{M}_s} \mathrm{cost}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ r & \text{otherwise.} \end{cases} \tag{1}$$

DEFINITION 2 (Chaining seed heuristic). *Given a set of matches $\mathcal{M}$ of costs less than r, the* chaining seed heuristic $h^{\mathcal{M}}_{\mathrm{csh}}$ *in a state $u{=}\langle i, j\rangle$ is the minimal cost to jointly align each remaining seed on a single path in the edit graph:*

$$h^{\mathcal{M}}_{\mathrm{csh}}(u) := \min_{\pi \in G(u \rightsquigarrow v_t)} \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \mathrm{cost}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ r & \text{otherwise.} \end{cases} \tag{2}$$

where π runs over all paths from u to v_t, and π_s is the shortest part of the path π that is an alignment of the seed s.

We abbreviate $h_{\mathrm{sh}} := h^{\mathcal{M}}_{\mathrm{sh}}$ and $h_{\mathrm{csh}} := h^{\mathcal{M}}_{\mathrm{csh}}$ when $\mathcal{M}$ is the set of all matches of cost strictly less than r. For such $\mathcal{M}$, both heuristics guarantee finding an optimal alignment.

THEOREM 1. *The seed heuristic h_{sh} and the chaining seed heuristic h_{csh} are* admissible.

The proof follows directly from the definitions and can be found in Appendix A.1. As a consequence of the proof, $h_{\mathrm{csh}}(u)$ dominates $h_{\mathrm{sh}}(u)$, that is, $h_{\mathrm{csh}}(u) \geq h_{\mathrm{sh}}(u)$ for all u. Hence $h_{\mathrm{csh}}(u)$ usually expands fewer states, at the cost of being more complex to compute.

3.3 Seed heuristic as a maximization problem

In order to efficiently evaluate the seed heuristic, Eq. (1), we rewrite it from a minimization of match costs to a maximization of match scores. We first define a few new concepts, and then rewrite the equation into form that will be simpler to compute.

Score. We define the *score of a match* m as $\mathrm{score}(m) := r - \mathrm{cost}(m)$. This is always positive since match costs are defined to be strictly less than r. The *score of a seed* s is the maximal score of a match of s:

$$\mathrm{score}(s) := \begin{cases} \max_{m \in \mathcal{M}_s} \mathrm{score}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

The *score of a state* is the sum of the scores of the remaining seeds,

$$S_{\mathrm{sh}}\langle i, j\rangle := \sum_{s \in \mathcal{S}_{\geq i}} \mathrm{score}(s). \tag{4}$$

Potential. The *potential* $P\langle i, j\rangle$ is the value of the heuristic when there are no matches, and is the maximal value the heuristic can take in a given state:

$$P\langle i, j\rangle := r \cdot |\mathcal{S}_{\geq i}|. \tag{5}$$

Objective. Eq. (1) can now be rewritten in terms of potential and score,

$$\begin{aligned} h^{\mathcal{M}}_{\mathrm{sh}}(u) &\overset{(1)}{=} \sum_{s \in \mathcal{S}_{\geq i}} \left(r - \begin{cases} \max_{m \in \mathcal{M}_s} r - \mathrm{cost}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ 0 & \text{otherwise} \end{cases} \right) \\ &= r \cdot |\mathcal{S}_{\geq i}| - \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \max_{m \in \mathcal{M}_s} \mathrm{score}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ 0 & \text{otherwise} \end{cases} \\ &\overset{(3)}{=} r \cdot |\mathcal{S}_{\geq i}| - \sum_{s \in \mathcal{S}_{\geq i}} \mathrm{score}(s) \overset{(4),(5)}{=} P(u) - S_{\mathrm{sh}}(u). \end{aligned} \tag{6}$$

The potential $P(u)$ of a state is simple to compute, and the score $S_{\mathrm{sh}}(u)$ can be computed by using *layers*.

Layer. Let *layer* $\mathcal{L}_\ell$ be the set of states u with score $S_{\mathrm{sh}}(u) \geq \ell$. Since S_{sh} is non-increasing in i and independent of j, and only changes value at the start of seeds, $\mathcal{L}_\ell$ is fully determined by largest i_ℓ such that $S_{\mathrm{sh}}\langle i_\ell, \cdot\rangle \geq \ell$. The score $S_{\mathrm{sh}}\langle i, j\rangle$ is then the largest ℓ such that $i_\ell \geq i$.

3.4 Chaining seed heuristic as a maximization problem

Similar to the seed heuristic, we rewrite Eq. (2) into a maximization form that will be simpler to compute.

Chain score. The *score of a chain* is the sum of the scores of the matches in the chain. Let $S_{\mathrm{chain}}(m)$ be the maximum score of a chain starting with match m. This satisfies the recursion

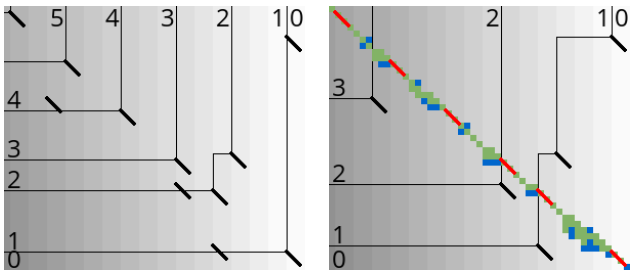
$$S_{\mathrm{chain}}(m) := \mathrm{score}(m) + \max_{m \preceq m'} S_{\mathrm{chain}}(m'). \tag{7}$$

where this and the following maxima are taken as 0 when they are over an empty set. For the chaining seed heuristic, the score at a state is

$$S_{\mathrm{csh}}(u) := \max_{u \preceq m} S_{\mathrm{chain}}(m). \tag{8}$$

Objective. Similar to the seed heuristic, the chaining seed heuristic is computed via the following equality,

$$\begin{aligned} h^{\mathcal{M}}_{\mathrm{csh}}(u) &\overset{(2)}{=} \min_{\pi \in G(u \rightsquigarrow v_t)} \sum_{s \in \mathcal{S}_{\geq i}} \left(r - \begin{cases} \mathrm{score}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ 0 & \text{otherwise} \end{cases} \right) \\ &= r \cdot |\mathcal{S}_{\geq i}| - \max_{\pi \in G(u \rightsquigarrow v_t)} \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \mathrm{score}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ 0 & \text{otherwise} \end{cases} \\ &= r \cdot |\mathcal{S}_{\geq i}| - \max_{u \preceq m_1 \preceq \dots \preceq m_l} \left\{ \mathrm{score}(m_1) + \dots + \mathrm{score}(m_l) \right\} \\ &\overset{(5),(8)}{=} P(u) - S_{\mathrm{csh}}(u). \end{aligned} \tag{9}$$



(a) Before (b) After

Fig. 3. An example of the layers and contours used by the chaining seed heuristic before and after the A⋆ execution with match pruning for $r{=}1$ and seed length $k{=}3$. Exact matches with $\mathrm{score}(m){=}1$ are shown as black diagonal segments (⬊). *Contours* are shown as horizontal and vertical black lines and indicate the bottom-right boundaries of *layers* $\mathcal{L}_\ell$ consisting of the states above/left of the contour marked with ℓ. States u between two contours have the same maximal number of matches $S_{\mathrm{csh}}(u)$ on a chain to the end. The value of the heuristic is shown in the background, from white ($h{=}0$) to darker grey. Expanded states are shown in green (■), open states in blue (■), and pruned matches in red (⬊). Note that pruning matches during the A⋆ execution shifts the contours and changes the layers.

Contours. Like fore the seed heuristic, we define layer $\mathcal{L}_\ell$ as those states u with score $S_{\mathrm{csh}}(u)$ *at least* ℓ. The ℓth *contour* is the bottom-right boundary of $\mathcal{L}_\ell$ (see Fig. 3). A state $u \in \mathcal{L}_\ell$ is *dominant* when there is no state $v \in \mathcal{L}_\ell$ with $v \neq u$ and $u \preceq v$. Both $\mathcal{L}_\ell$ and the corresponding contour are completely determined by the matches starting in the dominant states. The following lemma makes this precise:

LEMMA 1. *For $\ell > 0$ the matches m with $\ell \leq S_{\mathrm{chain}}(m) < \ell + r$ fully determine layer $\mathcal{L}_\ell$:*

$$\mathcal{L}_\ell = \{u \mid \exists m \in \mathcal{M} : u \preceq m \text{ and } \ell \leq S_{\mathrm{chain}}(m) < \ell + r\}. \quad (10)$$

PROOF. It follows directly from Eq. (8) that $\mathcal{L}_\ell = \{u \mid \exists m \in \mathcal{M} : u \preceq m \text{ and } S_{\mathrm{chain}}(m) \geq \ell\}$. Suppose that m has score $S_{\mathrm{chain}}(m) \geq \ell + r$. By definition, $\mathrm{score}(m) \leq r$, so by Eq. (7) there must be a match $m \preceq m'$ with $S_{\mathrm{chain}}(m') = S_{\mathrm{chain}}(m) - \mathrm{score}(m) \geq (\ell{+}r){-}r = \ell$. This implies that $u \preceq m \preceq m'$, which allows m, and hence any match with $S_{\mathrm{chain}}(m) \geq \ell + r$, to be omitted from consideration. □

We will use this lemma to efficiently find the largest ℓ such that layer $\mathcal{L}_\ell$ contains a state u, which gives $S_{\mathrm{csh}}(u)$.

Note that the formulas for the chaining seed heuristic become equivalent to those for the seed heuristic when the partial order $\langle i, j\rangle \preceq \langle i', j'\rangle$ is redefined to mean $i \leq i'$, ignoring the j-coordinate.

3.5 Match pruning

In order to reduce the number of states expanded by the A⋆ algorithm, we apply the *multiple-path pruning* observation: once a shortest path to a vertex has been found, no other path to this vertex could possibly improve the global shortest path (Poole and Mackworth, 2017). When A⋆ expands the start of a match we *prune* this match, so that the heuristic values of preceding states do no longer benefit from it, thus getting deprioritized by the A⋆. We define *pruned* variants of the seed heuristic and the chaining seed heuristic that ignore pruned matches:

DEFINITION 3 (Match pruning). *Let E be the set of expanded states during the A⋆ search, and let $\mathcal{M}\backslash E$ be the set of* unpruned matches*, i.e. those matches not starting in an expanded vertex. Then the pruning seed heuristic is $\hat{h}_{\mathrm{sh}} := h_{\mathrm{sh}}^{\mathcal{M}\backslash E}$ and the pruning chaining seed heuristic is $\hat{h}_{\mathrm{csh}} := h_{\mathrm{csh}}^{\mathcal{M}\backslash E}$.*

The hat ($\hat{h}$) denotes the implicit dependency on the progress of the A⋆. Even though match pruning breaks the admissibility of our heuristics for some vertices, we prove that A⋆ is sill guaranteed to find a shortest path:

THEOREM 2. *A⋆ with match pruning heuristic $\hat{h}_{\mathrm{sh}}$ or $\hat{h}_{\mathrm{csh}}$ finds a shortest path.*

The proof can be found in Appendix A.2. Since the heuristic may increase over time, our algorithm ensures that f is up-to-date when expanding a state by *reordering* nodes with outdated f values (see Remark 1 in Appendix A.2).

4 Algorithm

Our algorithm computes the shortest path in the edit graph using a variant of A⋆ that handles pruning (Section 4.1). This depends on the efficient computation of the heuristics (Sections 4.2 and 4.3). Section 4.4 contains further implementation notes.

At a high level, we first initialize the heuristic by finding all seeds and matches and precomputing the potential $P[i]$ and layers $\mathcal{L}_\ell$. Then we run the A⋆ search that evaluates the heuristic in many states, and updates the heuristic whenever a match is pruned.

4.1 A⋆ algorithm

We give our variant of the A⋆ algorithm (Hart *et al.*, 1968) that adds steps A4 and A6a (marked as **bold**) to handle the pruning of matches. All computed values of g are stored in a map, and all states in the front are stored in a priority queue of tuples $(v, g(v), f(v))$ ordered by increasing f.

- A1. Set $g(v_s) = 0$ in the map and initialize the priority queue with $(v_s, g(v_s){=}0, f(v_s))$.
- A2. Pop the tuple (u, g, f) with minimal f.
- A3. If $g > g(u)$, i.e. the value of $g(u)$ in the map has decreased since pushing the tuple, go to step 2. This is impossible for a consistent heuristic.
- **A4.** If $f \neq f(u)$, *reorder* u: push $(u, g(u), f(u))$ and go to step 2. This can only happen when f changes value, i.e. after pruning.
- A5. If $u = v_t$, terminate and report a best path.
- A6. Otherwise, *expand* u:
  - **a.** Prune matches starting at u.
  - b. For each successor v of u, *open* v when either $g(v)$ has not yet been computed or when $g(u) + d(u, v) < g(v)$. In this case, update the value of $g(v)$ in the map and insert $(v, g(v), f(v))$ in the priority queue. Go to step 2. This makes u the *parent* of v.

Note that the heuristic is evaluated in steps A4 and A6b for the computation of $f(u)$ and $f(v)$ respectively.

A vertex is called *closed* after it has been expanded for the first time, and called *open* when it is present in the priority queue.

4.2 Computing the seed heuristic

We compute the seed heuristic using an array LS that contains for each layer $\mathcal{L}_\ell$ the *layer start* $LS[\ell] := i_\ell = \max\{i : S_{\mathrm{sh}}\langle i, \cdot\rangle \geq \ell\}$.

We give the algorithms to precompute LS, to evaluate the heuristic using it, and to update it after pruning.

Precomputation.

- R1. Compute the set of seeds $\mathcal{S}$ by splitting A into consecutive kmers.
- R2. Build a hashmap containing all kmers of B.
- R3. For each seed, find all matches via a lookup in the hashmap. In case of inexact matches ($r = 2$), all sequences at distance 1 from each seed are looked up independently. The matches $\mathcal{M}$ are stored in a hashmap keyed by the start of each match.
- R4. Initialize the array of potentials P by iterating over the seeds backwards.
- R5. Initialize the array of layer starts LS by setting $LS[0] = n$ and iterating over the seeds backwards. For each seed $s = A_{i\ldots i'}$ that has matches, push $\mathrm{score}(s)$ copies of i to the end of LS.

Evaluating the heuristic in $u = \langle i, j\rangle$.

- E1. Look up the potential $P[i]$.
- E2. $S_{\mathrm{sh}}(u) = \max\{\ell : LS[\ell] \geq i\}$ is found by a binary search over ℓ.
- E3. Return $h(u) = P[i] - S_{\mathrm{sh}}(u)$.

Pruning when expanding match start $u = \langle i, j\rangle$.

- P1. Compute $seedscore_{old} := \mathrm{score}(s)$ and $\ell_{old} := S_{\mathrm{sh}}(u)$.
- P2. Remove all matches from $\mathcal{M}$ that start at u.
- P3. Compute $seedscore_{new} := \mathrm{score}(s)$ and set $\ell_{new} := \ell_{old} - seedscore_{old} + seedscore_{new}$. If $\ell_{new} < \ell_{old}$, remove layers $LS[\ell_{new}+1]$ to $LS[\ell_{old}]$ from LS and shift down larger elements $LS[\ell]$ with $\ell > \ell_{old}$ correspondingly.

4.3 Computing the chaining seed heuristic

The computation of the chaining seed heuristic is similar to that of the seed heuristic. It involves a slightly more complicated array LM of *layer matches* that associates to each layer $\mathcal{L}_\ell$ a list of matches with score ℓ: $LM[\ell] = \{m \in \mathcal{M} \mid S_{\mathrm{chain}}(m) = \ell\}$. The score $S_{\mathrm{csh}}(u)$ is then the largest ℓ such that $LM[\ell]$ contains a match m preceded by u.

Our algorithm for computing the chaining seed heuristic is similar to the one for the seed heuristic (Section 4.2). Hence we highlight only the steps which differ (e.g. step 2′ is used instead of 2 for the seed heuristic).

Precomputation.

- R5′. Initialize LM by setting $LM[0] = \emptyset$. Iterate over all matches in order of decreasing i of the match start, compute $\ell = S_{\mathrm{chain}}(m)$ (see point 2′ below), and add m to $LM[\ell]$.

Evaluating the heuristic in $u = \langle i, j\rangle$.

- E2′. Because of Lemma 1, the score $S_{\mathrm{csh}}(u)$ can be computed using binary search: $S_{\mathrm{csh}}(u) \geq \ell$ holds if and only if one of the layers $LM[\ell']$ with $\ell \leq \ell' < \ell + r$ contains a match m with $u \preceq m$. This is checked by simply iterating over all the matches in these layers.

Pruning when expanding match start $u = \langle i, j\rangle$.

- P2′. Compute $\ell_u = S_{\mathrm{csh}}(u)$, and remove all matches that start at u from layers $LM[\ell_u - r + 1]$ to $LM[\ell_u]$.
- P3′. Iterate over increasing ℓ starting at $\ell = \ell_u + 1$ and recompute $\ell' := S_{\mathrm{chain}}(m) \leq \ell$ for all matches m in $LM[\ell]$. Move m from $LM[\ell]$ to layer $LM[\ell']$ whenever $\ell' \neq \ell$. Stop when either r consecutive layers are unchanged, in which case no further changes of $S_{\mathrm{chain}}(m)$ can happen because of Eq. (7) and $\mathrm{score}(m) \leq r$, or when all matches in r consecutive layers have shifted down by the same amount, say $\Delta := \ell - \ell'$. In the latter case, $S_{\mathrm{chain}}(m)$ decreases by Δ for all matches with score at least ℓ. We remove the emptied layers $LM[\ell - \Delta + 1]$ to $LM[\ell]$ so that all higher layers shift down by Δ.

4.4 Implementation

This section covers some implementation details which are necessary for a good performance.

Bucket queue. We use a hashmap to store all computed values of g in the A* algorithm. Since the edit costs are bounded integers, we implement the priority queue using a *bucket queue* (Bertsekas, 1991). Unlike heaps, this data structure has amortized constant time push and pop operations since the difference between the value of consecutive pop operations is bounded.

Greedy matching of letters. From a state $\langle i, j\rangle$ where $a_i = b_j$, it is sufficient to only consider the matching edge to $\langle i+1, j+1\rangle$ (Allison, 1992; Ivanov *et al.*, 2020), and ignore the insertion and deletion edges to $\langle i, j+1\rangle$ and $\langle i+1, j\rangle$. During alignment, we greedily match as many letters as possible within the current seed before inserting only the last opened state in the priority queue. We do not cross seed boundaries in order to not interfere with match pruning. We include greedily matched states in the reported number of expanded states.

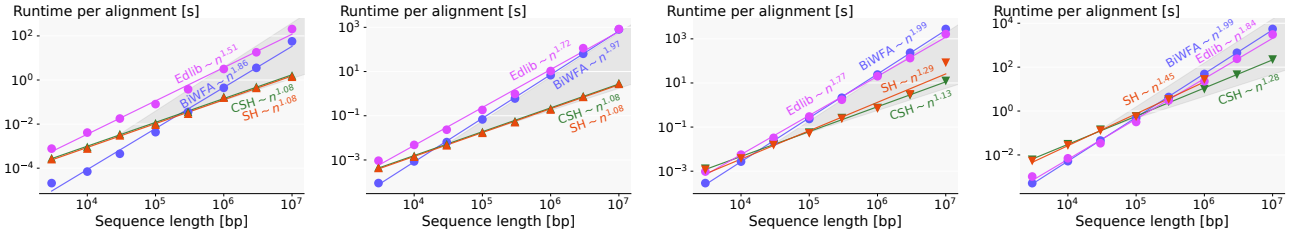
Priority queue offset. Pruning the last match in a layer may cause an increase of the heuristic in all states preceding the start u of the match. This invalidates f values in the priority queue and causes reordering (step A4). We skip most of the update operations by keeping a global offset to the f-values in the priority queue, which is updated it when all states in the priority queue precede u.

Split vector for layers. Pruning a match may lead to the removal of one or more layers of the array of layer starts LS or the array of layer matches LM, after which all higher layers are shifted down. To support this efficiently, we use a *split vector*: a data structure that internally consists of two stacks, one containing the layers before the last deletion, and one containing the layers after the last deletion in reverse order. To delete a layer, we move layers from the top of one of the stacks to the top of the other, until the layer to be deleted is at the top of one of the stacks, and then remove it. When layers are removed in decreasing order of ℓ, this takes linear total time.

Binary search hints. When we open v from its parent u in step A6b of the A* algorithm, the value of $h(v)$ is close to the value of $h(u)$. In particular, $S_{\mathrm{sh}}(v)$ (resp. $S_{\mathrm{csh}}(v)$) is close to and at most the score in u. Instead of a binary search (step E2), we do a linear search starting at the value of $\ell = S_{\mathrm{sh}}(u)$ (resp. $\ell = S_{\mathrm{csh}}(u)$).

We also speed up the binary search in the recomputation of $f(u)$ in the reordering check (step A4) by storing the last computed value of $hint(u) := S_{\mathrm{sh}}(v_s) - S_{\mathrm{sh}}(u)$ (resp. $S_{\mathrm{csh}}(v_s) - S_{\mathrm{csh}}(u)$) in the hashmap. When evaluating $S_{\mathrm{sh}}(u)$, we start the linear search at $S_{\mathrm{sh}}(v_s) - hint(u)$ with a fallback to binary search in case the linear search needs more than 5 iterations. Since $hint(u)$ remains constant when matches starting after u are pruned, it is a good starting point for the search when only matches near the tip of the A* search are pruned.

Code correctness. Our implementation A*PA is written in Rust, contains many assertions testing e.g. the correctness of our A* and layers data structure implementation, and is tested for correctness and performance on randomly-generated sequences. Correctness is checked against simpler algorithms (Needleman-Wunsch) and other aligners (EDLIB, BIWFA).



(a) e=1%, exact matching (b) e=5%, exact matching (c) e=10%, inexact matching (d) e=15%, inexact matching

Fig. 4. Log-log plots of the runtime for aligning **synthetic sequences** of increasing length for A*PA seed heuristic (SH), A*PA chaining seed heuristic (CSH), EDLIB (●) and BIWFA (●). The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. For $e \leq 5\%$, SH (▼) and CSH (▼) use k=15, r=1. For $e \geq 10\%$, SH (▲) and CSH (▲) use k=15, r=2. The missing data points for SH at e=15% are due to exceeding the memory limit (30 GB). Each runtime is the average over $\lfloor 10^7/n \rfloor$ alignments.

5 Results

We implemented the algorithms from Section 2 in the aligner A*PA and refer to the two of our algorithms as SH for seed heuristic, and CSH for chaining seed heuristic. Here we empirically compare the runtime and memory usage to the exact optimal aligners EDLIB and BIWFA on both synthetic and real data. We demonstrate the benefit of match pruning on the runtime scaling with sequence length, and the benefits from match chaining and inexact matches on scaling to high error rates. All code, evaluation scripts and data are available in the A*PA repository.

5.1 Setup

Synthetic data. We measure the performance of aligners on a set of randomly-generated sequences. Each set is parametrized by the number of sequence pairs, the sequence length n, and the error rate e. The first sequence in a pair is generated by concatenating n i.i.d. letters from Σ. The second sequence is generated by sequentially applying $\lfloor e \cdot n \rfloor$ edit operations (insertions, deletions, and substitutions with equal probability) to the first sequence. Note that errors can cancel each other, so the final distance between the sequences is usually less than $\lfloor e \cdot n \rfloor$. In order to minimize the measurement errors, each test consists of a number of sequence pairs.

Human data. We consider two datasets[1] of Oxford Nanopore Technologies (ONT) reads that are aligned to regions of the telomere-to-telomere assembly of a human genome CHM13 (v1.1) (Nurk *et al.*, 2022). Statistics are shown in Table 1.

- *CHM13: Human reads without biological variation.* We randomly sampled 50 reads of length at least 500 kbp from the first 12 GB of the ultra-Long ONT reads that were used to assemble CHM13. We removed soft clipped regions and paired each read to its corresponding reference region in CHM13[2].
- *NA12878: Human reads with biological variation.* We consider the long ONT MinION reads from a different reference sample NA12878 (Bowden *et al.*, 2019) which was used to evaluate BIWFA (Marco-Sola *et al.*, 2022). This dataset contains 48 reads longer than 500 kbp that were mapped to CHM13.

Compared algorithms and aligners. We compare the seed heuristic and chaining seed heuristic, implemented in A*PA, to the state-of-the-art exact pairwise aligners BIWFA and EDLIB. In order to study the performance of the A* heuristic functions and the pruning optimization, we also compare to Dijkstra's algorithm (which is equivalent to A* with a zero heuristic) and to a no-pruning variant of A*, all implemented in A*PA. The exact aligners SEQAN and PARASAIL are not included in this evaluation since they have been outperformed by BIWFA and EDLIB (Marco-Sola *et al.*, 2021). We execute all aligners with unit edit costs and compute not only the edit distance but also an optimal alignment. See Appendix A.4 for aligner versions and parameters.

Dataset	Biological mutations	Length [kbp]			Error rate [%]		
		min	mean	max	min	mean	max
CHM13	No	500	590	840	2.7	6.3	18.0
NA12878	Yes	502	624	1053	4.4	7.4	19.8

Table 1. Statistics on the **real data**: ONT reads from human samples.

Execution. All evaluations are executed on Arch Linux on a single thread on an `Intel Core i7-10750H CPU @ 2.6GHz` processor with 64 GB of memory and 6 cores, with hyper-threading and performance mode disabled. We fix the CPU frequency to `2.6GHz` and limit the available memory per execution to 30 GB using `ulimit -v 30000000`. To speed up the evaluations, we run 3 jobs in parallel, pinned to cores 0, 2, and 4.

Measurements. The runtime (wall-clock time) and memory usage (resident set size) of each run are measured using `time`. The runtime in all plots and tables refers to the average alignment time per pair of sequences. To reduce startup overhead, we average faster alignments over more pairs of sequences, as specified in the figure captions. Memory usage is measured as the maximum used memory during the processing of the whole input file. To estimate how algorithms scale with sequence length, we calculate best fit polynomials via a linear fit in the log-log domain using the least squares method.

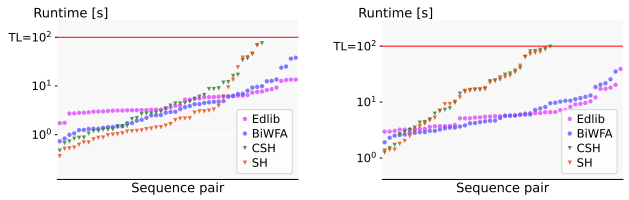
Parameters for the A* heuristics. Longer sequences contain more potential locations for off-path seed matches (i.e. lying outside of the resulting optimal alignment). Each match takes time to be located and stored, while also potentially worsening the heuristic. To keep the number of matches low, longer seeds are to be preferred. On the other hand, to handle higher error rates, a higher number of shorter reads is preferable. For simplicity, we fix k=15 throughout our evaluations as a reasonable trade-off between scaling to large n and scaling to high e. We use exact matches (r=1) for low error rates ($e \leq 5\%$), and inexact matches (r=2) for high error rates ($e \geq 10\%$). For human datasets we use r=2 since they include sequences with error rates higher than 10%.

[1] `https://github.com/RagnarGrootKoerkamp/astar-pairwise-aligner/releases/tag/datasets`

[2] `https://github.com/RagnarGrootKoerkamp/bam2seq`

	Aligner	Algorithm	Runtime [s]				Memory [MB]			
$e =$			1%	5%	10%	15%	1%	5%	10%	15%
●	EDLIB	Banding, exponential search, bit-parallel	206	839	1653	3073	**102**	**106**	**105**	**106**
●	BIWFA	Diagonal transition, divide & conquer	56.9	806	2799	5492	148	168	181	190
▲▼	A*PA	A*, match-pruning, seed heuristic	**1.41**	**2.79**	82.5	ML	434	548	4759	ML
▲▼	A*PA	A*, match-pruning, chaining seed heuristic	1.60	2.98	**12.3**	**220**	436	543	2369	4696

Table 2. Runtime and memory comparison on **synthetic sequences** of length $n{=}10^7$ bp for various error rates. ML stands for exceeding the memory limit of 30 GB. Note that we run our A* heuristics with exact matches (▲▲) when $e{\leq}5\%$, and with inexact matches (▼▼) when $e{\geq}10\%$.



(a) CHM13: ONT read errors

(b) NA12878: ONT read errors + biological mutations

Fig. 5. Log plot comparison of the aligners' runtime on **real data**. The data points for each individual aligner are sorted by alignment time. Alignments that timed out after 100 seconds are not shown.

5.2 Comparison on synthetic data

Here we compare the runtime scaling and performance of the exact optimal aligners on synthetic data. Appendix A.5 compares the effects of pruning and inexact matches, whereas Appendix A.6 compares SH and CSH in terms of the number of expanded states.

Scaling with sequence length. First, we compare the aligners by runtime scaling with sequence length n for various error rates e (Fig. 4). As expected from the theoretical analysis, EDLIB and BIWFA scale approximately quadratically regardless of the error rate. Unlike them, SH and CSH scale subquadratically which explains why they are faster for long enough sequences.

More specifically, for low error rates ($e{=}1\%$ and $e{=}5\%$), both SH and CSH with exact matches scale near-linearly with sequence length (best fit $n^{1.08}$ for $n{\leq}10^7$), and the benefit from chaining is negligible.

For high error rates ($e{\geq}10\%$) and large n, the need to match the seeds inexactly causes a significant number of off-path matches. These off-path matches lower the value of the heuristics and prevent the punishment of suboptimal paths. This causes SH to expand a super-linear number of states (Appendix A.6). Chaining the matches enforces them to follow the order of the seeds, which greatly reduces the negative effects of the off-path matches, leading to only linearly many expanded states. Nevertheless, the runtime scaling of CSH is super-linear as a result of the increasing fraction of time (up to 93% for $n{=}10^7$) spent on reordering states because of outdated f values after pruning.

Performance. Table 2 compares the runtime and memory of A*PA to EDLIB and BIWFA for aligning a single pair of sequences of length $n{=}10^7$. A* with CSH is more than 250 times faster than EDLIB and BIWFA at $e{=}5\%$. For low error rate ($e{\leq}5\%$), there is no significant performance difference between SH and CSH. The memory usage of A*PA for $e{\leq}5\%$ is less than 600 MB for any n. At $e{=}15\%$, CSH uses at most 4.7 GB while SH goes out of memory ($\geq$ 30 GB) because there are too many expanded states to store.

5.3 Comparison on real data

In this section we compare the exact optimal aligners on long ONT reads from our human datasets (Fig. 5). With the presented minimalistic features, A*PA aligns some sequences faster than BIWFA and EDLIB, but the high runtime variance makes it slower overall (Appendix A.7).

On the dataset without biological variation CHM13, SH is faster than BIWFA and EDLIB on 58% of the alignments (29 of 50). On the dataset with biological variation NA12878, SH outperforms BIWFA and EDLIB on 17% of the alignments (8 of 48) and in other cases is over an order of magnitude slower. In both datasets, SH and CSH time out for the sequences with the highest edit distances, because they have an error rate larger than the heuristic can handle efficiently ($e \geq r/k = 2/15 = 13.3\%$).

CSH usually explores fewer states than SH since chaining seed heuristic dominates the seed heuristic. However, in certain cases CSH is slower than SH since needs more time to update the heuristic after pruning (Step 3′ in Section 4.3).

6 Discussion

Our graph-based approach to alignment differs considerably from dynamic programming approaches, mainly because of the ability to use information from the entire sequences. This additional information enables radically more focused path-finding at the cost of more complex algorithms.

Limitations. Our presented method has several limitations:

1. *Complex regions trigger quadratic search.* Since it is unlikely that edit distance in general can be solved in strongly subquadratic time, it is inevitable that there are inputs for which our algorithm requires quadratic time. In particular, regions with high error rate, long indels, and too many matches (Appendix A.8) are challenging and trigger quadratic exploration.
2. *High constant in runtime complexity.* Despite the near-linear scaling of the number of expanded states (Appendix A.6), A*PA only outperforms EDLIB and BIWFA for sufficiently long sequences (Fig. 4) due to the relatively high computational constant that the A* search induces.
3. *Complex parameter tuning.* The performance of our algorithm depends heavily on the sequences to be aligned and the corresponding choice of parameters (whether to use chaining, the seed length k, and whether to use inexact matches r). The parameter tuning (currently very simple (Section 5.1) may require a more comprehensive framework when introducing additional optimizations.
4. *Real data.* The efficiency of the presented algorithm has high variability on real data (Section 5.3) due to high error rates, long indels, and multiple repeats (demonstrated in Fig. 8). Further optimizations are needed to align complex data.

Future work. We foresee a multitude of extensions and optimizations that may lead to efficient global aligning for production usage.

1. *Performance.* The practical performance of our A* approach could be improved using multiple existing ideas from the alignment domain: diagonal transition method, variable seed lengths, overlapping seeds, combining heuristics with different seed lengths, gap costs between matches in a chain (Ukkonen, 1985; Wilbur and Lipman, 1984), more aggressive pruning, and better parameter tuning. More efficient implementations may be possible by using computational domains (Spouge, 1989), bit-parallelization (Myers, 1999), and SIMD (Marco-Sola *et al.*, 2021).
2. *Generalizations.* Our method can be generalized to more expressive cost models (non-unit costs, affine costs) and different alignment types (semi-global, ends-free, and possibly local alignment).
3. *Relaxations.* Abandoning the optimality guarantee enables various performance optimizations. Another relaxation of our algorithm would be to validate the optimality of a given alignment more efficiently than finding an optimal alignment from scratch.
4. *Analysis.* The near-linear scaling behaviour requires a thorough theoretical analysis (Medvedev, 2022a). The fundamental question that remains to be answered is: *What sequences and what errors can be tolerated while still scaling near-linearly with the sequence length?* We expect both theoretical and practical contributions to this question.

7 Conclusion

We presented an algorithm with an implementation in A*PA solving pairwise alignment between two sequences. The algorithm is based on A* with a seed heuristic, inexact matching, match chaining, and match pruning, which we proved to find an exact solution according to edit distance. For random sequences with up to 15% uniform errors, the runtime of A*PA scales near-linearly to very long sequences (10^7 bp) and outperforms other exact aligners. We demonstrate that on real ONT reads from a human genome, A*PA is faster than other aligners on only a limited portion of the reads.

8 Acknowledgements

The authors are grateful to Mykola Akulov for his help with producing Figs. 1 and 3, and to Benjamin Bichsel, Maximilian Mordig and André Kahles for valuable feedback on various draft versions. We thank Sergey Nurk for his help with the preparation of the human data. Ragnar Groot Koerkamp is financed by ETH Grant ETH-17 21-1 to Gunnar Rätsch.

References

Allison, L. (1992). Lazy dynamic-programming can be eager. *Information Processing Letters*.

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

Benson, G., Levy, A., and Shalom, R. (2014). Longest common subsequence in k-length substrings.

Benson, G., Levy, A., Maimoni, S., Noifeld, D., and Shalom, B. R. (2016). Lcsk: a refined similarity measure. *Theoretical Computer Science*, **638**, 11–26.

Bertsekas, D. P. (1991). *Linear network optimization: algorithms and codes*. MIT Press.

Bowden, R., Davies, R. W., Heger, A., Pagnamenta, A. T., de Cesare, M., Oikkonen, L. E., Parkes, D., Freeman, C., Dhalla, F., Patel, S. Y., *et al.* (2019). Sequencing of human genomes with nanopore technology. *Nature communications*, **10**(1), 1–9.

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

Deorowicz, S. and Grabowski, S. (2014). Efficient algorithms for the longest common subsequence in k-length substrings. *Information Processing Letters*, **114**(11), 634–638.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, **1**(1), 269–271.

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

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, **4**(2), 100–107.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, **18**(6), 341–343.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, **24**(4), 664–675.

Holte, R. C. (2010). Common misconceptions concerning heuristic search. In *Third Annual Symposium on Combinatorial Search*.

Hunt, J. W. and Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, **20**(5), 350–353.

Ivanov, P., Bichsel, B., Mustafa, H., Kahles, A., Rätsch, G., and Vechev, M. T. (2020). AStarix: Fast and Optimal Sequence-to-Graph Alignment. In *RECOMB 2020*.

Ivanov, P., Bichsel, B., and Vechev, M. (2022). Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds. In *RECOMB 2022*.

Koenig, S. and Likhachev, M. (2006). Real-time adaptive A*. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288.

Kucherov, G. (2019). Evolution of biosequence search algorithms: a brief survey. *Bioinformatics*, **35**(19), 3547–3552.

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

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

Marco-Sola, S., Eizenga, J. M., Guarracino, A., Paten, B., Garrison, E., and Moreto, M. (2022). Optimal gap-affine alignment in $O(s)$ space. *bioRxiv*.

Medvedev, P. (2022a). The limitations of the theoretical analysis of applied algorithms. *arXiv preprint:2205.01785*.

Medvedev, P. (2022b). Theoretical analysis of edit distance algorithms: an applied perspective. *arXiv preprint:2204.09535*.

Myers, E. W. (1986). An O(ND) difference algorithm and its variations. *Algorithmica*, **1**(1-4), 251–266.

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

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

Needleman, S. B. and Wunsch, C. D. (1970). 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.

Nurk, S., Koren, S., Rhie, A., Rautiainen, M., *et al.* (2022). The complete sequence of a human genome. *Science*, **376**(6588), 44–53.

Pavetić, F., Katanić, I., Matula, G., Žužić, G., and Šikić, M. (2017). Fast and simple algorithms for computing both LCSk and LCSk+. *arXiv preprint:1705.07279*.

Pearl, J. (1984). *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

Poole, D. L. and Mackworth, A. K. (2017). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, second edition.

Prjibelski, A. D., Korobeynikov, A. I., and Lapidus, A. L. (2018). Sequence analysis. In *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*, pages 292–322.

Reinert, K., Dadi, T. H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., *et al.* (2017). The SeqAn C++ template library for efficient sequence analysis: a resource for programmers. *Journal of biotechnology*, **261**, 157–168.

Sankoff, D. (1972). Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, **69**(1), 4–6.

Sellers, P. H. (1974). On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, **26**(4), 787–793.

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

Spouge, J. L. (1989). Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal on Applied Mathematics*, **49**(5), 1552–1566.

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

Vintsyuk, T. K. (1968). Speech discrimination by dynamic programming. *Cybernetics*, **4**(1), 52–57.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM (JACM)*, **21**(1), 168–173.

Wilbur, W. J. and Lipman, D. J. (1984). The context dependent comparison of biological sequences. *SIAM Journal on Applied Mathematics*, **44**(3), 557–567.

A Appendix

A.1 The seed heuristic and chaining seed heuristic are admissible

THEOREM 3. *The seed heuristic h_{sh} and the chaining seed heuristic h_{csh} are* admissible.

PROOF. Let $u{=}\langle i,j\rangle$ be any state in the edit graph $G(V,E)$, and let π be any path from u to the target t. A heuristic h is admissible if $h(u) \leq \text{cost}(\pi)$ for all $u \in V$ and for all paths π.

We first show that h_{csh} is admissible. Each remaining seed in $\mathcal{S}_{\geq i}$ is aligned somewhere by the path π. Since the seeds do not overlap, their shortest alignments in π do not have overlapping edges. Each edge in π has a non-negative cost, and hence $\sum_{s\in\mathcal{S}_i} \text{cost}(\pi_s) \leq \text{cost}(\pi)$. The alignment π_s of each seed either has cost less than r and equals a match in $\mathcal{M}_s$, or it has cost at least r otherwise. Together this implies that $h^{\mathcal{M}}_{\text{csh}}$ is a lower bound on the remaining cost $h^*(u)$:

$$\begin{aligned} h^{\mathcal{M}}_{\text{csh}}(u) &= \min_{\pi\in G(u\rightsquigarrow t)} \sum_{s\in\mathcal{S}_{\geq i}} \begin{cases} \text{cost}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ r & \text{otherwise.} \end{cases} \\ &\leq \min_{\pi\in G(u\rightsquigarrow t)} \sum_{s\in\mathcal{S}_{\geq i}} \text{cost}(\pi_s) \leq \min_{\pi\in G(u\rightsquigarrow t)} \text{cost}(\pi) = h^*(u). \end{aligned} \tag{11}$$

We now show that the seed heuristic $h^{\mathcal{M}}_{\text{sh}}(u)$ is bounded above by $h^{\mathcal{M}}_{\text{csh}}(u)$, so that it is also admissible:

$$\begin{aligned} h^{\mathcal{M}}_{\text{sh}}(u) &= \sum_{s\in\mathcal{S}_{\geq i}} \begin{cases} \min_{m\in\mathcal{M}_s} \text{cost}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ r & \text{otherwise.} \end{cases} \\ &\leq \sum_{s\in\mathcal{S}_{\geq i}} \min_{\pi\in G(u\rightsquigarrow t)} \begin{cases} \text{cost}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ r & \text{otherwise.} \end{cases} \\ &\leq \min_{\pi\in G(u\rightsquigarrow t)} \sum_{s\in\mathcal{S}_{\geq i}} \begin{cases} \text{cost}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ r & \text{otherwise.} \end{cases} = h^{\mathcal{M}}_{\text{csh}}(u). \end{aligned}$$

A.2 Partially admissible heuristic

Here, we generalize the concept of an *admissible* heuristic to that of a *partially admissible heuristic* that may change value as the A⋆ progresses, and even become inadmissible in some vertices. In Theorem 4 we show that A⋆ still finds a shortest path when used with a partial admissible heuristic.

DEFINITION 4 (Fixed vertex). *A* fixed *vertex is a closed vertex u to which A⋆ has found a shortest path, that is, $g(u) = g^*(u)$.*

A fixed vertex is never opened, and hence remains fixed.

DEFINITION 5 (Partial admissible). *A heuristic $\hat{h}$ is* partially admissible *when there exists a shortest path π^* from v_s to v_t for which at any point during the A⋆ and for any non-fixed vertex $u \in \pi^*$, the heuristic in u is a lower bound on the remaining distance $h^*(u)$: $\hat{h}(u) \leq h^*(u)$.*

We follow the structure and notation of Hart *et al.* (1968) to prove that A⋆ finds a shortest path when used with a partially admissible heuristic.

LEMMA 2 (Generalization of Hart *et al.* (1968), Lemma 1). *Assuming v_t is not fixed, there exists an open vertex n' on π^* with $g(n') = g^*(n')$ such that all vertices in π^* following n' are not fixed.*

PROOF. Let n^* be the last fixed vertex in π^*, and let n' be its successor on π^*. We have $g(n') = g^*(n')$ since π^* is a shortest path and n^* was expanded, and n' is open by our choice of n^*. □

COROLLARY 1 (Generalization of Hart *et al.* (1968), Corollary to Lemma 1). *Suppose that $\hat{h}$ is partially admissible, and suppose A⋆ has not terminated. Then n' as in Lemma 2 has $f(n') \leq g^*(v_t)$.*

PROOF. By definition of f we have $f(n') = g(n') + \hat{h}(n')$. Since $n' \in \pi^*$ and π^* does not contain any fixed vertices after n', the partial admissibility of $\hat{h}$ implies $\hat{h}(n') \leq h^*(n')$. Thus, $f(n') = g(n') + \hat{h}(n') = g^*(n') + \hat{h}(n') \leq g^*(n') + h^*(n') = g^*(v_t)$. □

THEOREM 4. *A⋆ with a partially admissible heuristic finds a shortest path.*

PROOF. The proof of Theorem 1 in Hart *et al.* (1968) applies, with the remark that we use the specific path π^* and the specific vertex n' from Corollary 1, instead of an arbitrary shortest path P. □

REMARK 1 (Monotone heuristic). *We call $\hat{h}(u)$* monotone *when it does not decrease as the A⋆ progresses. To find the vertex u with minimal $f(u)$, we add step A4 to the A⋆ algorithm in Section 4.1 to check whether the value of f in the priority queue is still up to date. If $f = f(u)$, we know that for each vertex (v, g_v, f_v) in the queue we have $f(u) \leq f_v \leq f(v)$ for all other open vertices v in the queue with stored value f_v, ensuring that u indeed has the minimal value of f value of all open vertices.*

A.3 Match pruning preserves finding a shortest path

LEMMA 3. *For both $\hat{h}_{\text{sh}}$ and $\hat{h}_{\text{csh}}$, any expanded state at the start of a seed is fixed.*

PROOF. Let $\hat{h}$ be either $\hat{h}_{\text{sh}}$ or $\hat{h}_{\text{csh}}$. We use a proof by contradiction. Thus, suppose that a state u at the start of a seed has minimal f among all open states, but the shortest path π^* from v_s to u has length $g^*(u) < g(u)$.

Let $n \neq u$ be the last expanded state on π^*, and let v be its successor. Let the unexpanded states of π^* following n that are at the start of a seed be $v \preceq w_0 \prec \cdots \prec w_l = u$, in this order. We will show that $f(v) < f(u)$, resulting in a contradiction with the fact that A⋆ always expands the open state with minimal f:

$$\begin{aligned} f(v) &= g(v) + \hat{h}(v) = g^*(v) + \hat{h}(v) && \pi^* \text{ is a shortest path to } v, \\ &\leq g^*(w_0) + \hat{h}(w_0) && \text{proven in part 1,} \\ &\leq \cdots \leq g^*(w_l) + \hat{h}(w_l) && \text{proven in part 2,} \\ &= g^*(u) + \hat{h}(u) < g(u) + \hat{h}(u) = f(u) && \text{by contradiction.} \end{aligned}$$

Part 1: Proof of $g^*(v) + h(v) \leq g^*(w_0) + h(w_0)$. Since w_0 is the first start of a seed at or after v, the set of seeds following v is the same as the set of seeds following u. Thus, the summation in $\hat{h}(v)$ and $\hat{h}(w_0)$ in Eq. (1) or Eq. (2) is over the same seeds. Since $v \preceq w_0$, this implies $\hat{h}(v) \leq \hat{h}(w_0)$ in both cases, and $g^*(v) \leq g^*(w_0)$ since $v \preceq w_0$.

Part 2: Proof of $g^*(w_i)+\hat{h}(w_i) \leq g^*(w_{i+1})+\hat{h}(w_{i+1})$. Note that w_i and w_{i+1} are at the start of the same or consecutive seeds. In the first case, $\hat{h}(w_i) \leq \hat{h}(w_{i+1})$ follows by definition since $w_i \preceq w_{i+1}$. Else, we have $\hat{h}(w_i) \leq \hat{h}(w_{i+1}) + r$. When $d := d(w_i, w_{i+1}) < r$, there is an unpruned match m from w_i to w_{i+1} in $M - E$ of cost d, since w_i is not expanded. This implies $\hat{h}(w_i) \leq \hat{h}(w_{i+1}) + d$, and we obtain

$$\begin{aligned} g^*(w_i) + \hat{h}(w_i) &= (g^*(w_{i+1}) - d) + \hat{h}(w_i) \\ &\leq (g^*(w_{i+1}) - \min(d,r)) + (\hat{h}(w_{i+1}) + \min(d,r)) \\ &= g^*(w_{i+1}) + \hat{h}(w_{i+1}). \end{aligned}$$ □

LEMMA 4. *The heuristics $\hat{h}_{\text{csh}}$ and $\hat{h}_{\text{sh}}$ are partially-admissible.*

PROOF. Let π^* be a shortest path from s to v_t. Let n be the last expanded state on π^* at the start of a seed. By Lemma 3 n is fixed. By

choice of n, no states on π^* following n are expanded, so no matches on π^* following n are pruned. The proof of Theorem 3 applies to the path π^* without changes, implying that $h(u) \leq h^*(u)$ for all u on π^* following n, for both $h = \hat{h}_{\text{sh}}$ and $h = \hat{h}_{\text{csh}}$. □

THEOREM 5. *A⋆ with match pruning heuristic $\hat{h}_{\text{sh}}$ or $\hat{h}_{\text{csh}}$ finds a shortest path.*

PROOF. Follows from Theorem 3 in Appendix A.2 and Lemma 4.

A.4 Aligner versions and parameters

A*PA. github.com/RagnarGrootKoerkamp/astar-pairwise-aligner (commit `550bbe5`, and for human data commit `394b629`), run as

```
astar-pairwise-aligner -i {input} --silent2
  -k {k} -r {r} --algorithm {Dijkstra,SH,CSH}
  [--no-prune]
```

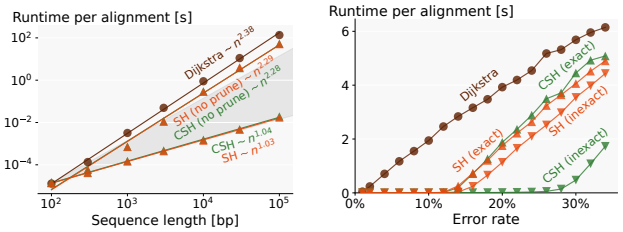
BIWFA. github.com/smarco/WFA2-lib (commit `ffa2439`), run as

```
align_benchmark -i {input} -a edit-wfa
  --wfa-memory-mode ultralow
```

EDLIB. We forked `v1.2.7` at github.com/RagnarGrootKoerkamp/edlib (commit `b34805c`) to support WFA's input format. Run as

```
edlib-aligner -p -s {input}
```

A.5 Effect of pruning, chaining, and inexact matches



(a) The effect of pruning on the runtime scaling with n (e=5%, k=15, exact matches). Note that SH and CSH coincide almost exactly.

(b) The effect of chaining and inexact matching on the runtime scaling with e (n=10^4, k=9, averaged over 100 alignments).

Fig. 6. The effect of optimizations on runtime scaling on **synthetic data**.

The optimizations and generalizations of the seed heuristic (Section 2) impact the performance in a complex way. Here we aim to provide intuitive explanations (Fig. 6).

Pruning enables near-linear scaling with length. Fig. 6a shows that match pruning has a crucial effect on the runtime scaling with length for both SH and CSH. Essentially, this optimization changes the quadratic runtime to near-linear runtime. The pruned variants of SH and CSH are averaged over $\lfloor 10^7/n \rfloor$ sequence pairs, while the no-prune variants and Dijkstra are averaged over $\lfloor 10^5/n \rfloor$ pairs.

Inexact matching and match chaining enable scaling to high error rates. Fig. 6b shows that inexact matches can tolerate higher error rates. Because of the larger number of matches, chaining is needed to preserve the near-linear runtime. There are two distinctive modes of operation: the runtime is close to constant up to a certain error rate, after which the runtime grows linearly in e. Thus, our heuristics can direct the search up to a certain fraction of errors, after which does a Dijkstra-like exploration step for each additional error. A reasonable quantification of the effect of different optimizations is to mark the error rate at which the heuristic transfers to the second (slow) mode of operation. For n=10^4 and k=9, Dijkstra starts a linear exploration at e=0%, SH and CSH with exact matches start at around 12%, SH with inexact matches start at around 14%, and CSH with inexact matches start at around 27%.

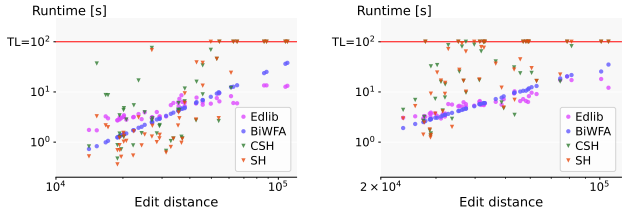
A.6 Expanded states and equivalent band

		SH				CSH			
e	n	10^4	10^5	10^6	10^7	10^4	10^5	10^6	10^7
1%		1.08	1.08	1.08	1.08	1.07	1.07	1.07	1.07
5%		1.90	1.92	1.92	1.95	1.89	1.91	1.90	1.91
10%		2.85	2.88	3.16	16.6	2.79	2.80	2.81	2.82
15%		18.9	21.7	43.4	ML	18.5	20.1	20.4	20.3

Table 3. Equivalent band for aligning **synthetic sequences** of a given length and error rate. Each cell averages over $\lfloor 10^7/n \rfloor$ alignments. ML stands for exceeding the memory limit of 30 GB.

Table 3 shows the *equivalent band* of each alignment: the number of expanded states divided by n. An equivalent band of 1 is the theoretical optimum for equal sequences, resulting from expanding only the states on the main diagonal. The equivalent band for CSH is always lower than the equivalent band for SH because the chaining seed heuristic dominates over seed heuristic. In practice, this difference becomes significant for large enough e and n (e>5% and n>10^5). The equivalent band of SH and CSH is constant for e=1%, indicating a linear scaling with n. In other cases (e≤5% for SH and e≤15% for CSH), the band increases slowly with n, indicating a near-linear scaling.

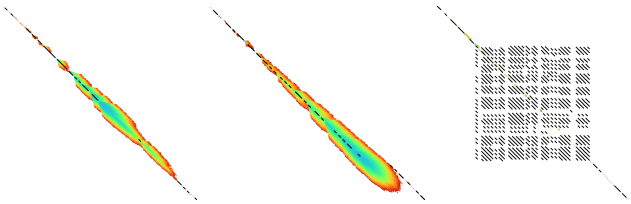
A.7 Performance variation on human genome



(a) CHM13 (b) NA12878

Fig. 7. Log-log plots of the runtime for aligning **real sequences** from two datasets of varying error distance. The sequence length varies between 500 kbp and 840 kbp for the CHM13 reads, and between 502 kbp and 1 053 kbp for the NA12878 reads. Alignments that timed out after 100 seconds are shown at 100 s. Parameters used for SH and CSH are k=15 and r=2.

Note that the BIWFA data points lie on a line, corresponding to the expected runtime of BIWFA of $O(s^2)$. Furthermore, the runtime of EDLIB can be seen to jump up around powers of two (2^{14}=16 384, 2^{15}=32 768, 2^{16}=65 536), corresponding to the exponential search of the edit distance. EDLIB has a runtime complexity of $O(ns)$, and hence is slower than BIWFA for small edit distances, but faster than BIWFA for large edit distances.



(a) High error rate (b) Long indel (c) Short repeats

Fig. 8. Expanded states by CSH (r=2, k=10) when aligning pairs of **synthetic sequences** (n=1000) with 8% random mutations containing (a) a region of length 200 with larger error rate than the heuristic can efficiently account for (50%), (b) a deletion of length 100, and (c) 60 mutated copies of a pattern of length 10. Seed matches are shown in black and occur on the best paths and in the repeated region. The order of expansion is shown by the color gradient from blue to red.

A.8 Complex alignments

Our algorithm finds optimal alignments very efficiently when both the sequence and the errors are uniformly random and the error rate is limited (Section 5). Nevertheless, since the alignment problem is fundamentally unsolvable in strictly subquadratic time (Backurs and Indyk, 2015), there are sequences which cannot be aligned fast. We give three cases (Fig. 8) where the complexity of our algorithm degrades.

1. *High error rate.* When the error rate becomes too high (larger than r/k), seeds can not increase the heuristic enough penalize all errors. Each unpenalized error increases f and needs more states to be searched, similar to Dijkstra. This can be mitigated by using shorter seeds and/or inexact matches, at the cost of introducing more matches.
2. *Long indel.* If there is a long insertion or deletion, the search has to accumulate the high cost for the long indel. Our heuristics do not account for long indels, and hence the search needs to expand states until the end of the gap is reached. Again, this causes a big increase of f and a search similar to Dijkstra. This may be improved in future work by introducing a *gap-cost* term to the chaining seed heuristic that penalizes gaps between consecutive matches in a chain.
3. *Short repeats.* When a short pattern repeats many times, this can result in a quadratic number of matches. This makes the corresponding seeds ineffective for the seed heuristic, and slows down the computation and pruning of the chaining seed heuristic. This can be partially mitigated by increasing the seed length and/or exact matches, at the cost of reducing the potential of the heuristic. Alternatively, seeds with too many matches could be completely ignored.

A.9 Notation

Table 4 summarizes the notation used in this work.

Table 4. Notation used in this paper.

Object	Notation
Sequences	
Alphabet	$\Sigma = \{A, C, G, T\}$
Sequences	$A = \overline{a_0 a_1 \dots a_i \dots a_{n-1}} \in \Sigma^*$
	$B = \overline{b_0 b_1 \dots b_j \dots b_{m-1}} \in \Sigma^*$
Subsequence	$a_{i..i'} := a_i a_{i+1} \dots a_{i'-1}$
Edit distance	$ed(A, B)$
Edit graph	
Graph	$G = (V, E)$
Vertices (states)	$u, v \in V = \{\langle i, j\rangle \mid 0 \le i \le n, 0 \le j \le m\}$
Edges	match/substitution $\langle i, j\rangle \to \langle i+1, j+1\rangle$
	deletion $\langle i, j\rangle \to \langle i+1, j\rangle$
	insertion $\langle i, j\rangle \to \langle i, j+1\rangle$
Distance	$d(u, v)$
Path, alignment	$\pi : (u \rightsquigarrow v)$
Shortest path	π^*
Cost	$\mathrm{cost}(\pi)$
Preceding	$u \preceq v, m_1 \preceq m_2$
A*	
Start and target state	$v_s, v_t \in V$
Distance from v_s	$g^* = d(v_s, \cdot)$
Distance to v_t	$h^* = d(\cdot, v_t)$
Heuristic	h
Best distance from start	g
Estimated distance	$f = g + h$
Admissible heuristic	$h \le h^*$
Consistent heuristic	$h(u) \le d(u, v) + h(v)$
Seeds and matches	
Seed length and potential	k, r
Seeds	$s \in \mathcal{S}, s_l = A_{l \cdot k..(l+1) \cdot k}$
Seeds in suffix	$\mathcal{S}_{\ge i}$
Matches (per seed)	$m \in \mathcal{M}, \mathcal{M}_s$
Cost of match	$0 \le \mathrm{cost}(m) < r$
Score of match	$0 < \mathrm{score}(m) = r - \mathrm{cost}(m) \le r$
Score of seed	$\mathrm{score}(s) = \max_{m \in \mathcal{M}_s} \mathrm{score}(m)$
Seed heuristic and chaining seed heuristic	
Potential	$P\langle i, j\rangle = r \cdot \lvert\mathcal{S}_{\ge i}\rvert$
State score (sh)	$S_{\mathrm{sh}}\langle i, j\rangle = \sum_{s \in \mathcal{S}_{\ge i}} \mathrm{score}(s)$
Chain score	$S_{\mathrm{chain}}(m) = \max_{m \preceq m_1 \preceq \dots} \sum_i \mathrm{score}(m_i)$
State score (csh)	$S_{\mathrm{csh}}(u) = \max_{u \preceq m} S_{\mathrm{chain}}(m)$
Seed heuristic	$h^{\mathcal{M}}_{\mathrm{sh}}(u) = P(u) - S_{\mathrm{sh}}(u)$
Chaining seed heuristic	$h^{\mathcal{M}}_{\mathrm{csh}}(u) = P(u) - S_{\mathrm{csh}}(u)$
Layer	$\mathcal{L}_\ell = \{u \mid S_{\mathrm{csh}}(u) \ge \ell\}$