Exact global alignment using A* with chaining 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

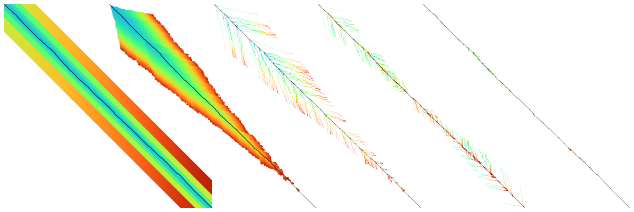
Motivation. Sequence alignment has been at the core of computational biology for half a century. Still, it is an open problem to design a practical algorithm for exact alignment of a pair of related sequences in linear-like time (Medvedev, 2022b).
Methods. We solve exact global pairwise alignment with respect to edit distance by using the A* shortest path algorithm. In order to efficiently align long sequences with high divergence, we extend the recently proposed *seed heuristic* (Ivanov *et al.*, 2022) with *match chaining*, *gap costs*, and *inexact matches*. We additionally integrate the novel *match pruning* technique and diagonal transition (Ukkonen, 1985) to improve the A* search. We prove the correctness of our algorithm, implement it in the A*PA aligner, and justify our extensions intuitively and empirically.
Results. On random sequences of divergence d=4% and length n, the empirical runtime of A*PA scales near-linearly (best fit $n^{1.05}$, $n \leq 10^7$ bp). A similar scaling remains up to d=12% (best fit $n^{1.24}$, $n \leq 10^7$ bp). For n=10^7 bp and d=4%, A*PA reaches $>300\times$ speedup compared to the leading exact aligners EDLIB and BIWFA. The performance of A*PA is highly influenced by long gaps. On long (n>500 kbp) ONT reads of a human sample it efficiently aligns sequences with d<10%, leading to $2\times$ median speedup compared to EDLIB and BIWFA. When the sequences come from different human samples, A*PA performs $1.4\times$ faster than EDLIB and BIWFA.
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 is known as *global pairwise alignment* (Navarro, 2001). Among others, it is applied to genome assembly, read mapping, variant detection, and multiple sequence alignment (Prjibelski *et al.*, 2019). Despite the centrality and age of pairwise alignment (Needleman and Wunsch, 1970), "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 subsequent analyses, so a common goal is to find a shortest sequence of edit operations (single letter insertions, deletions, and substitutions) that transforms one sequence into the other. The length of such a sequence is known as *Levenshtein distance* (Levenshtein, 1966) and *edit distance*. It has recently been proven that edit distance can not be computed in strongly subquadratic time, unless SETH is false (Backurs and Indyk, 2015). When the number of sequencing errors is proportional to the length, existing exact aligners scale quadratically both in the theoretical worst case and in practice. Given the increasing amounts of biological data and increasing read lengths, this is a computational bottleneck (Kucherov, 2019).

Our aim is to solve the global alignment problem provably correct and empirically fast. More specifically, we target near-linear scaling up to long sequences with high divergence. In contrast with approximate aligners that aim to find a good alignment, our algorithm proves that all other possible alignments are not better than the one we reconstruct.



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

Fig. 1. Computed states per algorithm. Various optimal alignment algorithms and their implementation are demonstrated on synthetic data (length n=500 bp, divergence d=16%). The colour indicates the order of computation from blue to red. (a) Band-doubling (EDLIB), (b) Dijkstra, (c) Diagonal transition/DT (WFA), (d) DT with divide-and-conquer/D&C (BIWFA), (e) A*PA with gap-chaining seed heuristic (GCSH), match pruning, and DT (seed length k=5 and exact matches).

1.1 Related work

Here we outline algorithms, tools, optimizations, and implementations for exact pairwise alignment of genetic sequences. Refer to Kucherov (2019) for approximate, probabilistic, and affine-cost algorithms and aligners.

Dynamic programming (DP). This classic approach to aligning two sequences computes a table where each cell contains the edit distance between a prefix of the first sequence and a prefix of the second by reusing the solutions for shorter prefixes. This quadratic DP was introduced for speech signals Vintsyuk (1968) and genetic sequences (Needleman and

Wunsch, 1970; Sankoff, 1972; Sellers, 1974; Wagner and Fischer, 1974). The quadratic $O(nm)$ runtime for sequences of lengths n and m allowed for aligning of long sequences for the time but speeding it up has been a central goal in later works. Implementations of this algorithm include SEQAN (Reinert *et al.*, 2017) and PARASAIL (Daily, 2016).

Band doubling and bit-parallelization. When the aligned sequences are similar, the whole DP table does not need to be computed. One such output-sensitive algorithm is the *band doubling* algorithm of Ukkonen (1985) (Fig. 1a) which considers only states around the main diagonal of the table, in a *band* with exponentially increasing width, leading to $O(ns)$ runtime, where s is the edit distance between the sequences. This algorithm, combined with the *bit-parallel optimization* by Myers (1999) is implemented in EDLIB (Šošić and Šikić, 2017) with $O(ns/w)$ runtime, where w is the machine word size (nowadays 64).

Diagonal transition (DT). The *diagonal transition* algorithm (Ukkonen, 1985; Myers, 1986) exploits the observation that the edit distance does not decrease along diagonals of the DP matrix. This allows for an equivalent representation of the DP table based on *farthest-reaching states* for a given edit distance along each diagonal. Diagonal transition has an $O(ns)$ worst-case runtime but only takes expected $O(n+s^2)$ time (Fig. 1c) for random input sequences (Myers, 1986) (which is still quadratic for a fixed divergence $d = s/n$). It has been extended to linear and affine costs in the *wavefront alignment* (WFA) (Marco-Sola *et al.*, 2021) in a way similar to Gotoh (1982). Its memory usage has been improved to linear in BIWFA (Marco-Sola *et al.*, 2023) by combining it with the divide-and-conquer approach of Hirschberg (1975), similar to Myers (1986) for unit edit costs.

Contours. The longest common subsequence (LCS) problem is a special case of edit distance, in which gaps are allowed but substitutions are forbidden. *Contours* partition the state-space into regions with the same remaining answer of the LCS subtask (Fig. 3). The contours can be computed in log-linear time in the number of matching elements between the two sequences which is practical for large alphabets (Hirschberg, 1977; Hunt and Szymanski, 1977).

Shortest paths and heuristic search using A*. A pairwise alignment that minimizes edit distance corresponds to a shortest path in the *alignment 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* (Hart *et al.*, 1968; 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.

Gap cost. One common heuristic function to speed up alignments is the *gap cost* (Ukkonen, 1985; Myers and Miller, 1995; Spouge, 1989; Wu *et al.*, 1990; Papamichail and Papamichail, 2009) This is a lower bound on the remaining edit distance and counts the minimal number of indels needed to align the suffixes of two sequences.

Seed-and-extend. *Seed-and-extend* is a commonly used paradigm for approximately solving semi-global alignment by first matching similar regions between sequences (*seeding*) to find *matches* (also called *anchors*), followed by *extending* these matches (Kucherov, 2019). Aligning long reads requires the additional step of chaining the seed matches (*seed-chain-extend*). Seeds have also been used to solve the LCSk generalization of LCS (Benson *et al.*, 2014; Pavetić *et al.*, 2017). Except for the seed heuristic (Ivanov *et al.*, 2022), most seeding approaches seek for seeds with accurate long matches.

Seed heuristic. A* with *seed heuristic* is an exact algorithm that was recently introduced for exact semi-global sequence-to-graph alignment (Ivanov *et al.*, 2022). In a precomputation step, the query sequence is split into non-overlapping *seeds* each of which is matched exactly to the reference. When A* explores a new state, the admissible seed heuristic is computed as the number of remaining seeds that cannot be matched in the upcoming reference. A limitation of the existing seed heuristic is that it only closely approximates the edit distance for very similar sequences (divergence up to 0.3% to $< 4\%$).

1.2 Contributions

We solve global pairwise alignment exactly and efficiently using the A* shortest path algorithm. In order to handle long and divergent sequences, we generalize the existing seed heuristic to a novel chaining seed heuristic which includes chaining, inexact matches, and gap costs. Then we improve the heuristic using a novel *match pruning* technique and optimize the A* search using the existing diagonal transition technique. We prove that all our heuristics guarantee that A* finds a shortest path.

Chaining seed heuristic. The *seed heuristic* (Ivanov *et al.*, 2022) penalizes missing seed matches. Our chaining seed heuristic requires matches to be chained in order, which reduces the negative effect of spurious (off-track) matches (Wilbur and Lipman, 1984; Benson *et al.*, 2016), and improves performance for highly divergent sequences. To handle long indels, we introduce the *gap-chaining seed heuristic* which uses a *gap cost* for indels between consecutive matches.

Inexact matches. We match seeds *inexactly* by aligning each seed with up to 1 edit, enabling our heuristics to potentially double in value and cope with up to twice higher divergence.

Match pruning. In order to further improve our heuristics, 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 through this vertex can improve the global shortest path. When A* expands the start or end of a match, we remove (*prune*) this match from further consideration, thus improving (increasing) the heuristic for the states preceeding the match. The incremental nature of our heuristic search is remotely related to Real-Time Adaptive A* (Koenig and Likhachev, 2006). Match pruning enables near-linear runtime scaling with length (Fig. 1e).

Diagonal transition A*. We speed up our algorithm by combining it with the diagonal transition optimization that skips states where a farther state along the same diagonal has already been reached with the same distance (Fig. 1c).

Implementation. We implement our algorithm in A*PA (A* Pairwise Aligner). The efficiency of our implementation relies on contours, which enables near-constant time evaluation of the heuristic.

Scaling and performance. We compare the absolute runtime, memory, and runtime scaling of A*PA to other exact aligners on synthetic data, consisting of random genetic sequences (length $n \leq 10^7$ bp, uniform divergence $d \leq 12\%$). For $d=4\%$ and $n=10^7$ bp, A*PA is up to 300× faster than the fastest aligners EDLIB (Šošić and Šikić, 2017) and BIWFA (Marco-Sola *et al.*, 2023). Our real dataset experiments involve >500 kbp long Oxford Nanopore (ONT) reads with $d<19.8\%$. When only sequencing errors are present, A*PA is 2× faster (in median) than the second fastest, and when genetic variation is also present, it is 1.4× faster.

2 Preliminaries

This section provides background definitions which are used throughout the paper. A summary of notation is included in App. D.

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. The *divergence* is the observed number of errors per letter, $d := \mathrm{ed}(A, B)/n$, whereas the *error rate* e is the number of errors per letter *applied* to a sequence.

Alignment graph. Let *state* $\langle i,j\rangle$ denote the subtask of aligning the prefix $A_{<i}$ to the prefix $B_{<j}$. The *alignment graph* (also called *edit graph*) $G(V,E)$ is a weighted directed graph with vertices $V = \{\langle i,j\rangle \mid 0 \le i \le n, 0 \le j \le m\}$ corresponding to all states, and edges connecting 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 starting state $\langle 0,0\rangle$ by v_s, the target state $\langle n,m\rangle$ by v_t, and the distance between states u and v by $\mathrm{d}(u,v)$. For brevity we write $f\langle i,j\rangle$ instead of $f(\langle i,j\rangle)$.

Paths and alignments. A path π from $\langle i,j\rangle$ to $\langle i',j'\rangle$ in the alignment graph G corresponds to a *(pairwise) alignment* of the substrings $A_{i\dots i'}$ and $B_{j\dots j'}$ with cost $\mathrm{c}_{\mathrm{path}}(\pi)$. A shortest path π^* from v_s to v_t corresponds to an optimal alignment, thus $\mathrm{c}_{\mathrm{path}}(\pi^*) = \mathrm{d}(v_s,v_t) = \mathrm{ed}(A,B)$. We write $g^*(u) := \mathrm{d}(v_s,u)$ for the distance from the start to a state u and $h^*(u) := \mathrm{d}(u,v_t)$ for the distance from u to the target.

Seeds and matches. 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, our heuristics will only depend on the *remaining* seeds $\mathcal{S}_{\ge i} := \{s_l \in \mathcal{S} \mid lk \ge i\}$ contained in the suffix $A_{\ge i}$. We denote the set of seeds between $u{=}\langle i,j\rangle$ and $v{=}\langle i',j'\rangle$ by $\mathcal{S}_{u\dots v} = \mathcal{S}_{i\dots i'} = \{s_l \in \mathcal{S} \mid i \le lk,\ lk+k \le i'\}$ and an *alignment* of s to a subsequence of B by π_s. The alignments of seed s with sufficiently low cost (Sec. 3.2) form the set $\mathcal{M}_s$ of *matches*.

Dijkstra and A*. Dijkstra's algorithm (Dijkstra, 1959) finds a shortest path from v_s to v_t by *expanding* (generating all successors) vertices in order of increasing distance $g^*(u)$ from the start. Each vertex to be expanded is chosen from a set of *open* vertices. The A* algorithm (Hart *et al.*, 1968; Pearl, 1984), 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, $h(u) \le h^*(u)$, which guarantees that A* has found a shortest path as soon as it expands v_t. Heuristic h_1 *dominates* (is *more accurate* than) another heuristic h_2 when $h_1(u) \ge h_2(u)$ 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 App. A.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 $i \le i'$ and $j \le j'$. 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 $m_1 \preceq \dots \preceq m_l$.

Gap cost. The number of indels to align substrings $A_{i\dots i'}$ and $B_{j\dots j'}$ is at least their difference in length: $c_{\mathrm{gap}}(\langle i,j\rangle,\langle i',j'\rangle) := |(i'{-}i){-}(j'{-}j)|$. For $u \preceq v \preceq w$, the gap cost satisfies the triangle inequalities $c_{\mathrm{gap}}(u,w) \le c_{\mathrm{gap}}(u,v) + c_{\mathrm{gap}}(v,w)$.

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 $u \preceq m_0 \preceq \dots$, 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 $S(u) < \ell$ (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

We intuitively motivate our A* heuristics and its improvements in Sec. 3.1. Then we formally define the general chaining seed heuristic (Sec. 3.2) that encompasses *inexact matches*, *chaining*, and *gap costs* (Fig. 2). Next, we introduce the *match pruning* (Sec. 3.3) improvement and integrate our A* algorithm with the *diagonal-transition* optimization (App. A.2). We present a practical algorithm (Sec. 3.4), implementation (App. A.3) and proofs of correctness (App. B).

3.1 Intuition

To align sequences A and B with minimal cost, we use the A* shortest path algorithm from the start to the end of the alignment graph (App. C.9). A* uses a heuristic function that estimates the shortest distance from the current vertex to the target. To ensure that A* finds a shortest path, the heuristic must be *admissible*, i.e. never overestimate the distance. A good heuristic efficiently computes an accurate estimate of the remaining distance.

Seed heuristic (SH). To define the *seed heuristic* h_{s}, we split A into short, non-overlapping substrings (*seeds*) of fixed length k. Since the whole sequence A has to be aligned, each of the seeds also has to be aligned somewhere in B. If a seed does not match anywhere in B without mistakes, then at least 1 edit has to be made to align it. To this end, we precompute for each seed from A, all positions in B where it matches exactly. Then, the SH is the number of remaining seeds (contained in $A_{\ge i}$) that do not match anywhere in B (Fig. 2a), and is a lower bound on the distance between the remaining suffixes $A_{\ge i}$ and $B_{\ge j}$. Next, we improve the seed heuristic with chaining, inexact matching, and gap costs.

Chaining (CSH). We improve on the SH by enforcing that the matches occur in the same order in B as their corresponding seeds occur in A, i.e., the matches form a *chain* going down and to the right. Now, the number of upcoming errors is at least the minimal number of remaining seeds that cannot be aligned on a single chain to the target. To compute this number efficiently, we instead count the maximal number of matches in a chain from the current state, and subtract that from the number of remaining seeds (Fig. 2b). This improves the seed heuristic when there are many spurious matches (i.e. outside the optimal alignment).

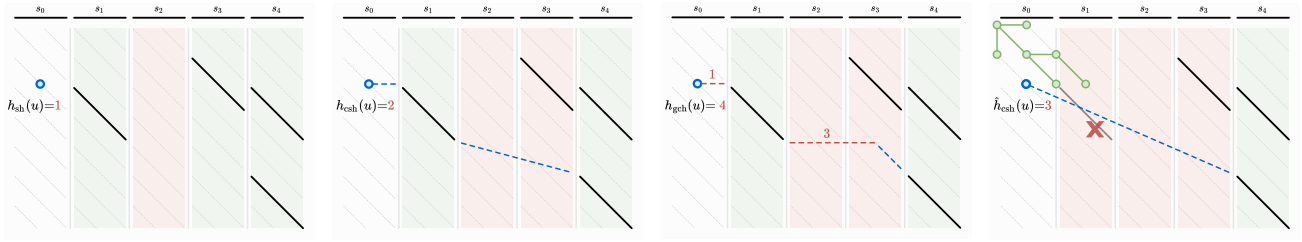
Gap costs (GCSH). In order to penalize long indels, the gap-chaining seed heuristic h_{gcs} (Fig. 2c) extends chaining seed heuristic by also incurring a *gap cost* if two consecutive matches in a chain do not lie on the same diagonal. In such cases, the transition to the next match requires at least as many gaps as the difference in length of the two substrings.

Inexact matches. To scale to higher divergence, we use *inexact matches*. For each seed in 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. Thus, the *potential* of our heuristics to penalize errors roughly doubles.

Pruning. We introduce the *match pruning* improvement (Fig. 2d) for all our heuristics: 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 can ignore all alternative paths to v. If v is the start or end of a match and v has been expanded, we are not interested in alternative paths using this match, and hence we simply ignore (*prune*) this match when evaluating a heuristic at states preceding v. Pruning a match increases the heuristic in states before the match, thus penalizing states preceding the "tip" of the A* search. Pruning significantly reduces the number of expanded states, and leads to near-linear scaling with the sequence length.

Diagonal transition (DT). The DT optimization (App. A.2) only computes a subset of *farthest reaching* states. This speeds up the A* in cases where the search becomes quadratic (Fig. 1c).

Finally, we prove correctness of all heuristics and optimizations, and in Sec. 3.4 we show how to compute the heuristics efficiently.



(a) Seed heuristic (b) Chaining seed heuristic (c) Gap-chaining seed heuristic (d) CSH + match pruning

Fig. 2. Demonstration of seed heuristic, chaining seed heuristic, gap-chaining seed heuristic, and match pruning. Sequence A is split into 5 seeds (horizontal black segments —) on top. Each seed is exactly matched in B (diagonal black segments \). The heuristic is evaluated at state u (blue circles ○), based on the 4 remaining seeds. The heuristic value is based on a maximal chain of matches (green columns ▮ for seeds with matches; red columns ▮ otherwise). Dashed lines denote chaining of matches. (a) The seed heuristic $h_{\text{s}}(u)$=1 is the number of remaining seeds that do not have matches (only s_2). (b) The chaining seed heuristic $h_{\text{cs}}(u)$=2 is the number of remaining seeds without a match (s_2 and s_3) on a path going only down and to the right containing a maximal number of matches. (c) The gap-chaining seed heuristic $h_{\text{gcs}}(u)$=4 is minimal cost of a chain, where the cost of joining two matches is the maximum of the number of not matched seeds and the gap cost between them. Red dashed lines denote gap costs. (d) Once the start or end of a match is expanded (green circles ○), the match is *pruned* (red cross ✗), and future computations of the heuristic ignore it. s_1 is removed from the maximum chain of matches starting at u so $\hat{h}_{\text{cs}}(u)$ increases by 1.

3.2 General chaining seed heuristic

We introduce three heuristics for A* that estimate the edit distance between a pair of suffixes. Each heuristic is an instance of a *general chaining seed heuristic*. After splitting the first sequence into seeds $\mathcal{S}$, and finding all matches $\mathcal{M}$ in the second sequence, any shortest path to the target can be partitioned into a *chain* of matches and connections between the matches. Thus, the cost of a path is the sum of match costs c_m and *chaining costs* γ. Our simplest seed heuristic ignores the position in B where seeds match and counts the number of seeds that were not matched (γ=c_{seed}). To efficiently handle more errors, we allow seeds to be matched inexactly, require the matches in a path to be ordered (CSH), and include the gap-cost in the chaining cost γ= $\max(c_{\text{gap}}, c_{\text{seed}})$ to penalize indels between matches (GCSH).

Inexact matches. We generalize the notion of exact matches to *inexact matches*. We fix a threshold cost r (0<r≤k) called the *seed potential* and define the set of *matches* $\mathcal{M}_s$ as all alignments m of seed s with *match cost* $\text{c}_\text{m}(m) < r$. The inequality is strict so that $\mathcal{M}_s = \varnothing$ 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 with r=2 we allow both exact and *inexact* matches with one edit. We do not consider higher r in this paper. For notational convenience, we define $m_\omega \notin \mathcal{M}$ to be a match from v_t to v_t of cost 0.

Potential of a heuristic. We call the maximal value the heuristic can take in a state its *potential* P. The potential of our heuristics in state $\langle i, j\rangle$ is the sum of seed potentials r over all seeds after i: $P\langle i, j\rangle := r \cdot |\mathcal{S}_{\geq i}|$.

Chaining matches. Each heuristic depends on a *partial order* on states that limits how matches can be *chained*. We write $u \preceq_p v$ for the partial order implied by a function p: $p(u) \preceq p(v)$. A $\preceq_p$*-chain* is a sequence of matches $m_1 \preceq_p \cdots \preceq_p m_l$ that precede each other: $\text{end}(m_i) \preceq_p \text{start}(m_{i+1})$ for $1 \leq i < l$. To chain matches according only to their i-coordinate, SH is defined using $\preceq_i$-chains, while CSH and GCSH are defined using $\preceq$ that compares both i and j.

Chaining cost. The *chaining cost* γ is a lower bound on the path cost between two consecutive matches: from the end state u of a match, to the start v of the next match.

For SH and CSH, the *seed cost* is r for each seed that is not matched: $c_{\text{seed}}(u, v) := r \cdot |\mathcal{S}_{u\ldots v}|$. When $u \preceq_i v$ and v is not in the interior of a seed, then $c_{\text{seed}}(u, v) = P(u) - P(v)$.

For GCSH, we also include the gap cost $c_{\text{gap}}(\langle i, j\rangle, \langle i', j'\rangle) := |(i'-i)-(j'-j)|$ which is the minimal number of indels needed to correct for the difference in length between the substrings $A_{i\ldots i'}$ and $B_{j\ldots j'}$ between two consecutive matches (Sec. 2). Combining the seed cost and the chaining cost, we obtain the gap-seed cost $c_{\text{gs}} = \max(c_{\text{seed}}, c_{\text{gap}})$, which is capable of penalizing long indels and we use for GCSH. Note that γ=c_{seed}+c_{gap} would not give an admissible heuristic since indels could be counted twice, in both c_{seed} and c_{gap}.

For conciseness, we also define the γ, c_{seed}, c_{gap}, and c_{gs} between matches $\gamma(m, m') := \gamma(\text{end}(m), \text{start}(m'))$, from a state to a match $\gamma(u, m') := \gamma(u, \text{start}(m'))$, and from a match to a state $\gamma(m, u) = \gamma(\text{end}(m), u)$.

General chaining seed heuristic. We now define the general chaining seed heuristic that we use to instantiate SH, CSH and GCSH.

Definition 1 (General chaining seed heuristic)**.** *Given a set of matches $\mathcal{M}$, partial order $\preceq_p$, and chaining cost γ, the* general chaining seed heuristic $h^{\mathcal{M}}_{p,\gamma}(u)$ *is the minimal sum of match costs and chaining costs over all $\preceq_p$-chains (indexing extends to $m_0 := u$ and $m_{l+1} := m_\omega$):*

$$h^{\mathcal{M}}_{p,\gamma}(u) := \min_{\substack{u \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t \\ m_i \in \mathcal{M}}} \sum_{0 \leq i \leq l} \big[\gamma(m_i, m_{i+1}) + \text{c}_\text{m}(m_{i+1})\big].$$

Heuristic		Order	Chaining cost γ
$h_{\text{s}}(u)$	Seed heuristic (SH)	$\preceq_i$	c_{seed}
$h_{\text{cs}}(u)$	Chaining seed h. (CSH)	$\preceq$	c_{seed}
$h_{\text{gcs}}(u)$	Gap-chaining seed h. (GCSH)	$\preceq$	$\max(c_{\text{gap}}, c_{\text{seed}})$

Table 1. Definitions of our heuristic functions. SH orders the matches by i and uses only the seed cost. CSH orders the matches by both i and j. GCSH additionally exploits the gap cost.

We instantiate our heuristics according to Table 1. Our admissibility proofs (App. B.1) are based on c_m and γ being lower bounds on disjoint parts of the remaining path. Since the more complex h_{gcs} dominates the other heuristics it usually expand fewer states.

Theorem 1. *The seed heuristic h_{s}, the chaining seed heuristic h_{cs}, and the gap-chaining seed heuristic h_{gcs} are* admissible*. Furthermore, $h^{\mathcal{M}}_{\text{s}}(u) \leq h^{\mathcal{M}}_{\text{cs}}(u) \leq h^{\mathcal{M}}_{\text{gcs}}(u)$ for all states u.*

We are now ready to instantiate A* with our admissible heuristics but we will first improve them and show how to compute them efficiently.

3.3 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 state has been found, no other path to this state could possibly improve the global shortest path (Poole and Mackworth, 2017). As soon as A* expands the start or end of a match, we *prune* it, so that the heuristic in preceding states no longer benefits from the match, and they get deprioritized by A*. We define *pruned* variants of all our heuristics that ignore pruned matches:

Definition 2 (Pruning heuristic)**.** *Let E be the set of expanded states during the A* search, and let $\mathcal{M}\backslash E$ be the set of matches that were not pruned, i.e. those matches not starting or ending in an expanded state. We say that $\hat{h} := h^{\mathcal{M}\backslash E}$ is a* pruning heuristic *version of h.*

The hat over the heuristic function ($\hat{h}$) denotes the implicit dependency on the progress of the A*, where at each step a different $h^{\mathcal{M}\backslash E}$ is used. Our modified A* algorithm (App. A.1) works for pruning heuristics by ensuring that the f-value of a state is up-to-date before expanding it, and otherwise *reorders* it in the priority queue. Even though match pruning violates the admissibility of our heuristics for some vertices, we prove that A* is sill guaranteed to find a shortest path (App. B.2). To this end, we show that our pruning heuristics are *weakly-admissible heuristics* (Def. 7) in the sense that they are admissible on at least one path from v_s to v_t.

Theorem 2. *A* with a weakly-admissible heuristic finds a shortest path.*

Theorem 3. *The pruning heuristics $\hat{h}_{\mathrm{s}}$, $\hat{h}_{\mathrm{cs}}$, $\hat{h}_{\mathrm{gcs}}$ are weakly admissible.*

Pruning will allow us to scale near-linearly with sequence length, without sacrificing optimality of the resulting alignment.

3.4 Computing the heuristic

We present an algorithm to efficiently compute our heuristics (pseudocode in App. A.4). At a high level, we rephrase the minimization of costs (over paths) to a maximization of *scores* (over chains of matches). We initialize the heuristic by precomputing all seeds, matches, potentials and a *contours* data structure used to compute the maximum number of matches on a chain. During the A* search, the heuristic is evaluated in all explored states, and the contours are updated whenever a match gets pruned.

Scores. The *score of a match* m is $\mathrm{score}(m) := r - \mathrm{c_m}(m)$ and is always positive. The *score of a $\preceq_p$-chain* $m_1 \preceq_p \cdots \preceq_p m_l$ is the sum of the scores of the matches in the chain. We define the chain score of a match m as

$$S_p(m) := \max_{m \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t} \big\{\mathrm{score}(m) + \cdots + \mathrm{score}(m_l)\big\}. \quad (1)$$

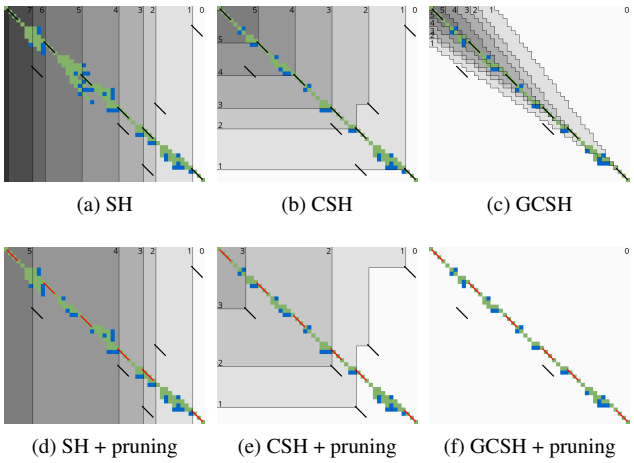
Since $\preceq_p$ is a partial order, S_p can be computed with base case $S_p(m_\omega) = 0$ and the recursion

$$S_p(m) = \mathrm{score}(m) + \max_{m \preceq_p m' \preceq_{v_t}} S_p(m'). \quad (2)$$

We also define the chain score of a state u as the maximum chain score over succeeding matches m: $S_p(u) = \max_{u \preceq_p m \preceq_p v_t} S_p(m)$, so that Eq. (2) can be rewritten as $S_p(m) = \mathrm{score}(m) + S_p(\mathrm{end}(m))$.

The following theorem allows us to rephrase the heuristic in terms of potentials and scores for heuristics that use $\gamma = c_{\mathrm{seed}}$ and respect the order of the seeds, which is the case for h_{s} and h_{cs} (proof in App. B.3):

Theorem 4. *$h^{\mathcal{M}}_{p,c_{\mathrm{seed}}}(u) = P(u) - S_p(u)$ for any partial order $\preceq_p$ that is a refinement of $\preceq_i$ (i.e. $u \preceq_p v$ must imply $u \preceq_i v$).*



(a) SH (b) CSH (c) GCSH

(d) SH + pruning (e) CSH + pruning (f) GCSH + pruning

Fig. 3. Contours and layers of different heuristics after aligning (n=48, m=42, r=1, k=3, edit distance 10). Exact matches are black diagonal segments (▬). The background colour indicates $S_p(u)$, the maximum number of matches on a $\preceq_p$-chain from u to the end starting, with $S_p(u) = 0$ in white. The thin black boundaries of these regions are *Contours*. The states of layer $\mathcal{L}_\ell$ *precede* contour ℓ. Expanded states are green (■), open states blue (■), and pruned matches red (▬). Pruning matches changes the contours and layers. GCSH ignores matches $m \not\preceq_T v_t$.

Layers and contours. We compute h_{s} and h_{cs} efficiently using *contours*. Let *layer* $\mathcal{L}_\ell$ be the set of states u with score $S_p(u) \geq \ell$, so that $\mathcal{L}_\ell \subseteq \mathcal{L}_{\ell-1}$. The ℓth *contour* is the boundary of $\mathcal{L}_\ell$ (Fig. 3). Layer $\mathcal{L}_\ell$ ($\ell > 0$) contains exactly those states that precede a match m with score $\ell \leq S_p(m) < \ell + r$ (Lemma 5 in App. B.3).

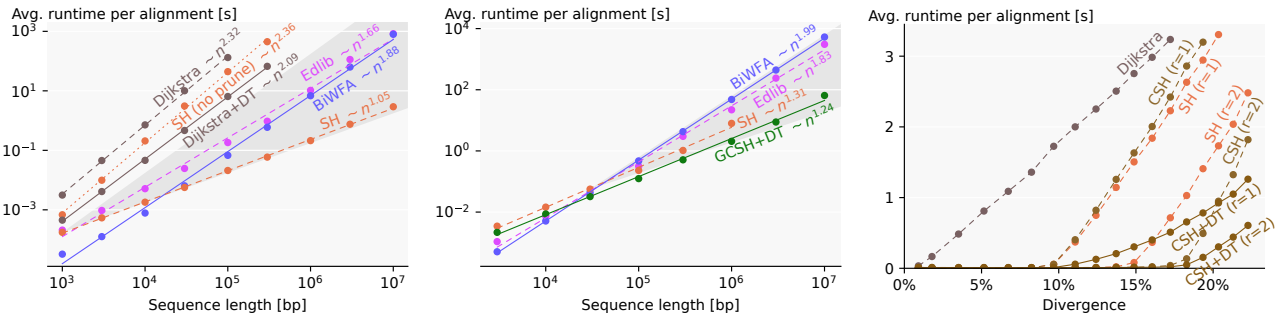
Computing $S_p(u)$. This last observation inspires our algorithm for computing chain scores. For each layer $\mathcal{L}_\ell$, we store the set $L[i]$ of matches having score ℓ: $L[\ell] = \{m \in \mathcal{M} \mid S_p(m) = \ell\}$. The score $S_p(u)$ is then the highest ℓ such that layer $L[\ell]$ contains a match m reachable from u ($u \preceq_p m$). From Lemma 5 we know that $S_p(u) \geq \ell$ if and only if one of the layers $L[\ell']$ for $\ell' \in [\ell, \ell + r)$ contains a match preceded by u. We use this to compute $S_p(u)$ using a binary search over the layers ℓ. We initialize $L[0] = \{m_\omega\}$ (m_ω is a fictive match at the target v_t), sort all matches in $\mathcal{M}$ by $\preceq_p$, and process them in decreasing order (from the target to the start). After computing $S_p(\mathrm{end}(m))$, we add m to layer $S_p(m) = \mathrm{score}(m) + S_p(\mathrm{end}(m))$. Matches that do not precede the target ($\mathrm{start}(m) \not\preceq_p m_\omega$) are ignored.

Pruning matches from L. When pruning matches starting or ending in state u in layer $\ell_u = S_p(u)$, we remove all matches that start at u from layers $L[\ell_u - r + 1]$ to $L[\ell_u]$, and all matches starting in some v and ending in u from layers $L[\ell_v - r + 1]$ to $L[\ell_v]$.

Pruning a match may change S_p in layers above ℓ_u, so we update them after each prune. We iterate over increasing ℓ starting at $\ell_u + 1$ and recompute $\ell' := S_p(m) \leq \ell$ for all matches m in $L[\ell]$. If $\ell' \neq \ell$, we move m from $L[\ell]$ to $L[\ell']$. We stop iterating when either r consecutive layers were left unchanged, or when all matches in $r - 1 + \ell - \ell'$ consecutive layers have shifted down by the same amount $\ell - \ell'$. In the former case, no further scores can change, and in the latter case, S_p decreases by $\ell - \ell'$ for all matches with score $\geq \ell$. We remove the emptied layers $L[\ell' + 1]$ to $L[\ell]$ so that all higher layers shift down by $\ell - \ell'$.

SH. Due to the simple structure of the seed heuristic, we also simplify its computation by only storing the start of each layer and the number of matches in each layer, as opposed to the full set of matches.

GCSH. Thm. 4 does not apply to gap-chaining seed heuristic since it uses chaining cost $\gamma = \max(c_{\mathrm{gap}}(u, v), c_{\mathrm{seed}}(u, v))$ which is different from $c_{\mathrm{seed}}(u, v)$. It turns out that in this new setting it is never optimal to chain two matches if the gap cost between them is higher than the seed



(a) Runtime scaling with length (d=4%, r=1) (b) Runtime scaling with length (d=12%, r=2) (c) Runtime scaling with divergence

Fig. 4. Runtime comparison on synthetic data (a)(b) Log-log plots comparing our simplest (SH) and most accurate heuristic (GCSH with DT) to EDLIB, BIWFA, and other algorithms (averaged over 10^6 to 10^7 total bp, seed length k=15). The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. SH without pruning is dotted, and variants with DT are solid. At 4% divergence, the complex techniques CSH, GCSH, and DT (not shown) are as fast as SH. Missing data points are due to exceeding the 32 GiB memory limit. (c) Runtime scaling with divergence (n=10^4, 10^6 total bp, k=10). GCSH (not shown) is as fast as CSH.

cost. Intuitively, it is better to miss a match than to incur additional gapcost to include it. We capture this constraint by introducing a transformation T such that $u \preceq_T v$ holds if and only if $c_{\text{seed}}(u,v) \geq c_{\text{gap}}(u,v)$, as shown in App. B.4. Using an additional *consistency* constraint on the set of matches we can compute $h^{\mathcal{M}}_{\text{gcs}}$ via S_T as before.

Definition 3 (Consistent matches)**.** *A set of matches $\mathcal{M}$ is* consistent *when for each $m \in \mathcal{M}$ (from $\langle i,j\rangle$ to $\langle i',j'\rangle$) with* $\text{score}(m)$>1*, for each* adjacent *pair of existing states $(\langle i,j\pm1\rangle, \langle i',j'\rangle)$ and $(\langle i,j\rangle, \langle i',j'\pm1\rangle)$, there is an* adjacent *match with corresponding start and end, and score at least* $\text{score}(m)-1$.

This condition means that for r=2, each exact match must be adjacent to four (or less around the edges of the graph) inexact matches starting or ending in the same state. Since we find all matches m with $c_m(m)<r$, our initial set of matches is consistent. To preserve consistency, we do not prune matches if that would break the consistency of $\mathcal{M}$.

Definition 4 (Gap transformation)**.** *The partial order $\preceq_T$ on states is induced by comparing both coordinates after the* gap transformation

$$T: \quad \langle i,j\rangle \mapsto (i-j-P\langle i,j\rangle, j-i-P\langle i,j\rangle)$$

Theorem 5. *Given a* consistent *set of matches $\mathcal{M}$, the gap-chaining seed heuristic can be computed using scores in the transformed domain:*

$$h^{\mathcal{M}}_{\text{gcs}}(u) = \begin{cases} P(u) - S_T(u) & \text{if } u \preceq_T v_t, \\ c_{\text{gap}}(u, v_t) & \text{if } u \not\preceq_T v_t. \end{cases}$$

Using the transformation of the match coordinates, we can now reduce c_{gs} to c_{seed} and efficiently compute GCSH in any explored state.

4 Results

Our algorithm is implemented in the aligner A*PA[1] in Rust. We compare it with state of the art exact aligners on synthetic (Sec. 4.2) and human (Sec. 4.3) data[2] using PABENCH[3]. We justify our heuristics and optimizations by comparing their scaling and performance (Sec. 4.4).

[1] github.com/RagnarGrootKoerkamp/astar-pairwise-aligner (1e3841e)

[2] github.com/pairwise-alignment/pa-bench/releases/tag/datasets

[3] github.com/pairwise-alignment/pa-bench (commit 55db710)

4.1 Setup

Synthetic data. Our synthetic datasets are parameterized by sequence length n, induced error rate e, and total number of basepairs N, resulting in N/n sequence pairs. The first sequence in each pair is uniform-random from Σ^n. The second is generated by sequentially applying $\lfloor e{\cdot}n \rfloor$ edit operations (insertions, deletions, and substitutions with equal 1/3 probability) to the first sequence. Introduced errors can cancel each other, making the *divergence* d between the sequences less than e. Induced error rates of 1%, 5%, 10%, and 15% correspond to divergences of 0.9%, 4.3%, 8.2%, and 11.7%, which we refer to as 1%, 4%, 8%, and 12%.

Human data. We use two datasetsof ultra-long Oxford Nanopore Technologies (ONT) reads of the human genome: one without and one with genetic variation. All reads are 500–1100 kbp long, with mean divergence around 7%. The average length of the longest gap in the alignment is 0.1 kbp for ONT reads, and 2 kbp for ONT reads with genetic variation (detailed statistics in Table 2). The reference genome is CHM13 (v1.1) (Nurk *et al.*, 2022). The reads used for each dataset are:

- *ONT*: 50 reads sampled from those used to assemble CHM13.
- *ONT with genetic variation*: 48 reads from another human (Bowden *et al.*, 2019), as used in the BIWFA paper (Marco-Sola *et al.*, 2023).

Algorithms and aligners. We compare SH, CSH, and GCSH as implemented in A*PA to the state-of-the-art exact aligners BIWFA and EDLIB. We also compare to Dijkstra and to variants without pruning, and without diagonal transition. We exclude SEQAN and PARASAIL since they are outperformed by BIWFA and EDLIB (Marco-Sola *et al.*, 2021).

A*PA parameters. We run all aligners with unit edit costs with traceback enabled, returning an alignment. In A*PA we fix r=2 (inexact matches) and seed length k=15. Both are a trade-off: inexact matches and lower k increase the heuristic potential to efficiently handle more divergent sequences, while too low k gives too many matches.

Execution. We use PABENCH on Arch Linux on an `Intel Core i7-10750H` processor with 64 GB of memory and 6 cores, without hyper-threading, frequency boost, and CPU power saving features. We fix the CPU frequency to `2.6GHz`, limit the memory usage to 32 GiB, and run 1 single-threaded job at a time with niceness −20.

Measurements. PABENCH first reads the dataset from disk and then measures the wall-clock time and increase in memory usage of each aligner. Plots and tables refer to the average alignment time per aligned pair. Best-fit polynomials are calculated via a linear fit in the log-log domain using the least squares method.

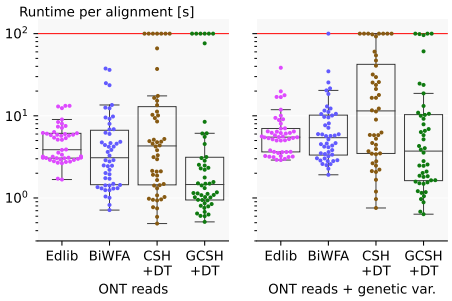


Fig. 5. Runtime on long human reads. Each dot is an alignment without (left) and with (right) genetic variation. Runtime is capped at 100 s. The median speedup of A*PA (GCSH + DT, k=15, r=2) is 2× (left) and 1.4× (right) over EDLIB and BIWFA.

4.2 Comparison on synthetic data

Runtime scaling with length. We compare our A* heuristics with EDLIB and BIWFA in terms of runtime scaling with n and d (Fig. 4, extended comparison in App. C.1). As theoretically predicted, EDLIB and BIWFA scale quadratically. For small edit distance, EDLIB is subquadratic due to the bit-parallel optimization. The empirical scaling of A*PA is subquadratic for $d \leq 12$ and $n \leq 10^7$, making it the fastest aligner for long sequences ($n > 30$ kbp). For low divergence ($d \leq 4\%$) even the simplest SH scales near-linearly with length (best fit $n^{1.05}$ for $n \leq 10^7$). For high divergence (d=12%) we need inexact matches, and the runtime of SH sharply degrades for long sequences ($n > 10^6$ bp) due to spurious matches. This is countered by chaining the matches in CSH and GCSH, which expand linearly many states (App. C.2). GCSH with DT is not exactly linear due to state reordering and high memory usage (App. C.6).

Performance. A*PA is >300× faster than EDLIB and BIWFA for d=4% and n=10^7 (Fig. 4a). For n=10^6 and $d \leq 12\%$, memory usage is less than 500 MB for all heuristics (App. C.4).

4.3 Comparison on human data

We compare runtime (Fig. 5 and App. C.3), and memory usage (App. C.4) on human data. We configure A*PA to prune matches only when expanding their start (not their end), leaving some matches on the optimal path unpruned and speeding up the contour updates. A*PA (GCSH with DT) aligns ONT reads faster than EDLIB and BIWFA in all quartiles, being >2× faster in median. However, the heuristic does not predict all errors when $d \geq 10\%$, causing 6 alignments to time out. With genetic variation, A*PA is 1.4× faster than EDLIB and BIWFA in median. Low-divergence alignments are faster than EDLIB, while high-divergence alignments are slower (3 sequences with $d \geq 10\%$ time out) because of expanding quadratically many states in complex regions (App. C.8). Since slow alignments dominate the total runtime, EDLIB is faster on average.

4.4 Effect of pruning, inexact matches, chaining, and DT

We visualize the effects of complex sequences on the A* search (App. C.9).

Pruning enables near-linear runtime. Figure 4a shows that match pruning changes the quadratic runtime of SH to near-linear, by penalizing the expansion of states before the search tip.

Inexact matches cope with higher divergence. Inexact matches increase the heuristic potential, allowing higher divergence (Fig. 4c). For low divergence, the runtime is nearly constant (App. C.5), while for higher divergence it switches to linear. Inexact matches nearly double the threshold for constant runtime from $d \leq 10\% = 1/k$ to $d \leq 18\% \approx 2/k$.

Chaining copes with spurious matches. When seeds have many spurious matches (typical for inexact matches, r=2 in Fig. 4c), SH loses its strength. CSH reduces the degradation of SH by chaining matches.

Gap-chaining copes with indels. Gap costs penalize both short and long indels which are common in genetic variation. As a result, GCSH is significantly faster than CSH with genetic variation (Fig. 5).

Diagonal transition speeds up quadratic search. When A* explores quadratically many states, it behaves similar to Dijkstra. DT speeds up Dijkstra 20× (Fig. 4a) and CSH 3× (Fig. 4c). For Dijkstra this is close to $1/d$, the expected reduction in number of expanded states. DT also speeds up the alignment of human data (App. C.3).

5 Discussion

Seeds are necessary, matches are optional. Unlike the seed-and-extend paradigm which reconstructs good alignments by connecting seed matches, we use the lack of matches to penalize alignments. Given the admissibility of our heuristics, the more seeds without matches, the higher the penalty for alignments and the easier it is to dismiss suboptimal ones. In the extreme, not having any matches can be sufficient for finding an optimal alignment in linear time (App. C.7).

Modes: Near-linear and quadratic. The A* algorithm with a seed heuristic has two modes of operation that we call *near-linear* and *quadratic*. In the near-linear mode A*PA expands few vertices because the heuristic successfully penalizes all edits between the sequences. Every edit that is not penalized by the heuristic widens the explored band, leading to the quadratic mode similar to Dijkstra.

Limitations.

1. *Limited divergence.* Our heuristics can estimate the remaining edit distance up to about $d \leq 10\%$ (on human data). Higher divergence triggers a slow quadratic search.
2. *Complex regions.* Regions with high divergence, long indels, or many matches trigger quadratic runtime (Appendices C.8 and C.9).
3. *Computational overhead of A*.* Graph algorithms are ≫100× less efficient than DP (compare Dijkstra and EDLIB in Fig. 4a). Despite the near-linear scaling of A*PA, it is only faster than EDLIB and BIWFA for $n \geq 30$ kbp.

Future work.

1. *Performance.* The runtime could be improved by variable seed lengths, overlapping seeds, more aggressive pruning, and better parameter tuning. Implementation optimizations may include computational domains (Spouge, 1989), block computations (Liu and Steinegger, 2021), SIMD (similar to BIWFA), or bit-parallelization (Myers, 1999) (similar to EDLIB).
2. *Generalizations.* Our chaining seed heuristic could be generalized to non-unit and affine costs, and to semi-global alignment. Cost models that better correspond to the data can speed up the alignment.
3. *Relaxations.* At the expense of optimality guarantees, inadmissible heuristics could speed up A* considerably. Another possible relaxation would be to validate the optimality of a given alignment instead of computing it from scratch.
4. *Analysis.* The near-linear scaling of A* is not asymptotic and requires a more thorough theoretical analysis (Medvedev, 2022a).

Acknowledgements

We are grateful to Mykola Akulov for his help with Figs. 1 and 3, to Daniel Liu for his involvement in developing PABENCH, to Benjamin Bichsel, Maximilian Mordig and André Kahles for valuable comments on drafts, and to Sergey Nurk for his help with the human data. R. Groot Koerkamp is financed by ETH Research 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. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, pages 707–710.

Liu, D. and Steinegger, M. (2021). Block aligner: fast and flexible pairwise sequence alignment with simd-accelerated adaptive blocks. *bioRxiv*.

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. (2023). Optimal gap-affine alignment in $o(s)$ space. *Bioinformatics*, **39**(2).

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.

Myers, G. and Miller, W. (1995). Chaining multiple-alignment fragments in sub-quadratic time. In *SODA*, volume 95, pages 38–47.

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.

Papamichail, D. and Papamichail, G. (2009). Improved algorithms for approximate string matching (extended abstract). *BMC Bioinformatics*, **10**(S1).

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. (2019). *Sequence Analysis*, pages 292–322. Academic Press, Oxford.

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.

Wu, S., Manber, U., Myers, G., and Miller, W. (1990). An o(np) sequence comparison algorithm. *Information Processing Letters*, **35**(6), 317–323.