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

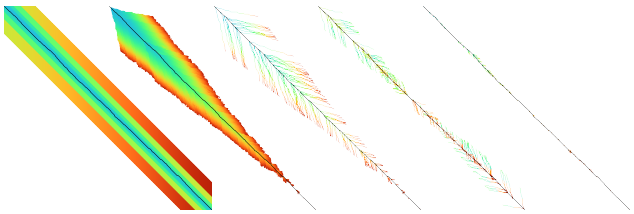
Motivation. Sequence alignment has been at the core of computational biology for half a century. Still, it is an open problem to design a practical algorithm for exact alignment of a pair of related sequences in linear-like time (Medvedev, 2022b).
Methods. We solve exact global pairwise alignment with 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 $n^{1.07}$, $n{\le}10^7\,\mathrm{bp}$). A similar scaling remains up to $d{=}12\%$ (best fit $n^{1.24}$, $n{\le}10^7\,\mathrm{bp}$). For $n{=}10^7\,\mathrm{bp}$ 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 ($n{>}500\,\mathrm{kbp}$) 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 and 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 $n{=}500\,\mathrm{bp}$, 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

© The Authors 2023.

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 the observation that the edit distance does not decrease along diagonals of the DP matrix. This allows for a compact equivalent representation based on the *farthest-reaching state* on each diagonal, for a fixed edit distance. 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 method. Instead of executing A* on the full $O(nm)$ state space of the alignment graph, this optimization only considers farthest-reaching states.

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 $\mathrm{c_{path}}(\pi)$. A shortest path π^* from v_s to v_t corresponds to an optimal alignment, thus $\mathrm{c_{path}}(\pi^*) = \mathrm{d}(v_s,v_t) = \mathrm{ed}(A,B)$. 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 start by intuitively motivating our general A* heuristics and its improvements in Sec. 3.1. Then we formally define our general heuristic (Sec. 3.2) with extensions *inexact matches*, *chaining*, and *gap costs* (Fig. 2). Next, we introduce the *match pruning* (Sec. 3.3) improvement of our heuristic and the *diagonal-transition* optimization of A* (Sections 3.3 and 3.4), and prove their correctness. We present a practical algorithm (Sec. 3.5) and implementation (App. A.3).

3.1 Intuition

In order to align sequences A and B with minimal cost, we use the A* algorithm to find a shortest path from the start to the end of the alignment graph (App. C.7). A* uses a problem-specific heuristic function that estimates the shortest distance from a current vertex to the target. To ensure A* finds a shortest path, a heuristic must be *admissible*, i.e. never overestimate the distance. A good heuristic estimates the distance accurately and is efficient to compute.

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).

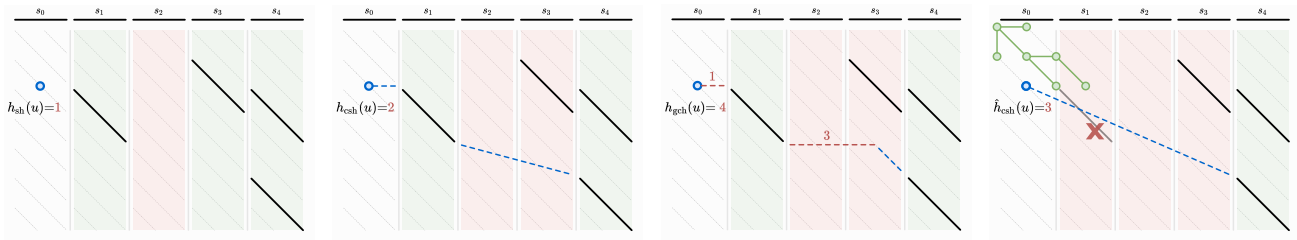
Gap costs (GCSH). In order to penalize long indels, the gap-chaining seed heuristic h_{gcs} (Fig. 2c) extends chaining seed heuristic by also incurring a *gap cost* if two consecutive matches in a chain do not lie on the same diagonal. In such cases, the transition to the next match requires at least 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). We improve the A* search using the *diagonal-transition* optimization. When the search becomes quadratic, this speeds it up several times by skipping intermediate states (Fig. 1c).

Finally, we prove correctness of all heuristics and optimizations, and in Sec. 3.5 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 $\mathrm{c_m}$ 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 ($0<r\leq k$) called the *seed potential* and define the set of *matches* $\mathcal{M}_s$ as all alignments m of seed s with *match cost* $\mathrm{c_m}(m) < r$. 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: $\mathrm{end}(m_i) \preceq_p \mathrm{start}(m_{i+1})$ 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 $\gamma(m, m') := \gamma(\mathrm{end}(m), \mathrm{start}(m'))$, from a state to a match $\gamma(u, m') := \gamma(u, \mathrm{start}(m'))$, and from a match to a state $\gamma(m, u) = \gamma(\mathrm{end}(m), u)$.

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 $m_{l+1} := m_\omega$):*

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

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 $\mathrm{c_m}$ 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 Diagonal transition for A*

For a given distance g, the diagonal transition method only considers the *farthest-reaching* (f.r.) state $u{=}\langle i,j\rangle$ on each diagonal $k{=}i{-}j$ at distance g. We use $F_{gk} := i{+}j$ to indicate the antidiagonal[1] of the farthest reaching state. Let X_{gk} be the farthest state on diagonal k adjacent to a state at distance $g{-}1$, which is then *extended* into F_{gk} by following as many matches as possible. The edit distance is the lowest g such that $F_{g,n-m} \geq n+m$, and we have the recursion

$$X_{gk} := \max(F_{g-1,k-1}{+}1, F_{g-1,k}{+}2, F_{g-1,k+1}{+}1), \tag{1}$$

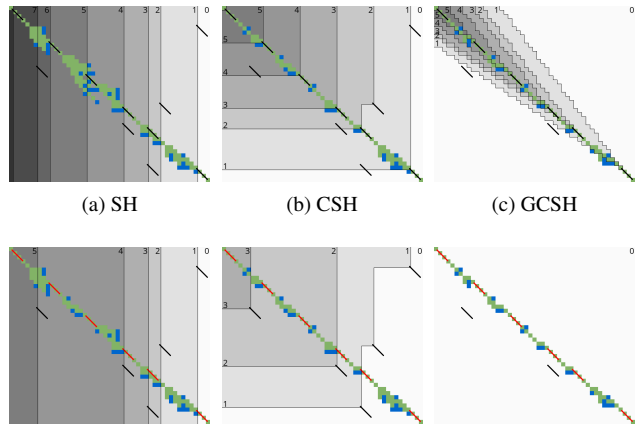
$$F_{gk} = X_{gk} + \mathrm{LCP}\left(A_{\geq(X_{gk}+k)/2}, B_{\geq(X_{gk}-k)/2}\right). \tag{2}$$

The base case is $X_{0,0}{=}0$ with default value $F_{gk}{=}{-}\infty$ for $k{>}|g|$, and LCP is the length of the longest common prefix of two strings. Each edge in a traceback path is either a match created by an extension (2), or a mismatch starting in a f.r. state (1). We call such a path an *f.r. path*.

We modify the A* algorithm with the *diagonal transition (DT) optimization* (pseudocode in App. A.2). In order to only consider f.r. paths, we replace g by F_{gk}. Each time a state u is opened or expanded, the check whether $g(u)$ decreases is replaced by a check whether F_{gk} increases. This causes the search to skip states that are known to not be farthest reaching. The proof of correctness (Thm. 2) still applies.

Alternatively, it is also possible to implement A* directly in the diagonal-transition state-space by pushing states F_{gk} to the priority queue, but for simplicity we keep the similarity with the original A*.

[1] Previous works indicate the column i of u, but using the antidiagonal $i{+}j$ keeps the symmetry between insertions and deletions.



(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 $m \not\preceq_T v_t$.

3.5 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

$$S_p(m) := \max_{m \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t} \left\{\mathrm{score}(m) + \cdots + \mathrm{score}(m_l)\right\}. \tag{3}$$

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

$$S_p(m) = \mathrm{score}(m) + \max_{m \preceq_p m' \preceq v_t} S_p(m'). \tag{4}$$

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. (4) 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 $u \preceq_i v$).*

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[2] $\mathcal{L}_\ell$, we store the set

[2] We refer to each of ℓ, $\mathcal{L}_\ell$, and $L[l]$ as layers.

$L[i]$ 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 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 $c_{\mathrm{seed}}(u, v) \geq c_{\mathrm{gap}}(u, v)$, as shown in App. B.4. Using an additional *consistency* constraint on the set of matches we can compute $h^{\mathcal{M}}_{\mathrm{gcs}}$ 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* $\mathrm{score}(m) > 1$*, for each* adjacent *pair of existing states* $(\langle i, j \pm 1\rangle, \langle i', j'\rangle)$ *and* $(\langle i, j\rangle, \langle i', j' \pm 1\rangle)$*, there is an* adjacent *match with corresponding start and end, and score at least* $\mathrm{score}(m) - 1$.

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 $\mathrm{c_m}(m) < r$, 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:*

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

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

		Length [kbp]			Divergence [%]			Max gap [kbp]		
Dataset	**Cnt**	min	mean	max	min	mean	max	min	mean	max
ONT	50	500	594	849	2.7	6.3	18.0	0.02	0.1	1
ONT+gen.var.	48	502	632	1053	4.4	7.4	19.8	0.05	**1.9**	**42**

Table 2. Human datasets statistics. ONT reads only include short gaps, while genetic variation also includes long gaps. **Cnt** is the number of sequence pairs. **Max gap** is the longest gap in the reconstructed alignment.

4 Results

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

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 datasets[4] of ultra-long Oxford Nanopore Technologies (ONT) reads of the human genome: one without and one with genetic variation (statistics in Table 2). The reads are filtered for length $n \geq 500$ kbp. 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.

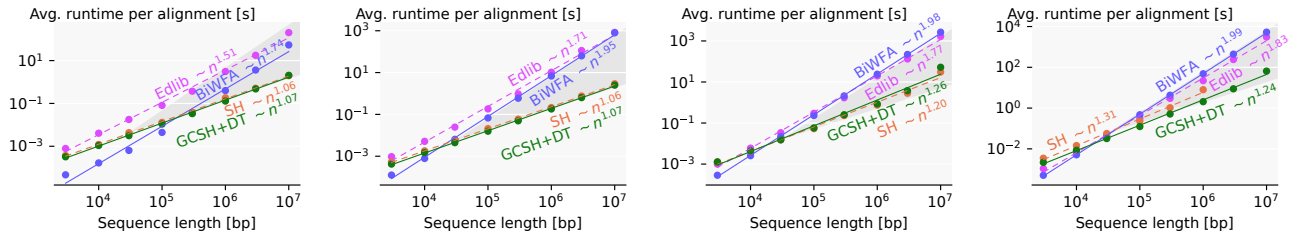
Execution. All evaluations are single threaded using PaBench[5]. on Arch Linux on an `Intel Core i7-10750H` processor with 64 GB of memory and 6 cores, without hyper-threading, frequency boost, and CPU power saving features. We fix the CPU frequency to `2.6GHz`, limit the memory usage per job to 32 GiB, and run only 1 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.

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

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

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



(a) d=1%, exact matches (b) d=4%, exact matches (c) d=8%, inexact matches (d) d=12%, inexact matches

Fig. 4. Runtime scaling with sequence length on synthetic data. Log-log plots comparing our simplest heuristic (SH) and our most accurate heuristic (GCSH, with DT) with EDLIB and BIWFA. The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. The seed length is k=15 and for d≥8% the heuristics use inexact matches (r=2). Averages are over total N=10^7 bp. The missing data points for SH at d=12% are due to exceeding the 32 GiB memory limit.

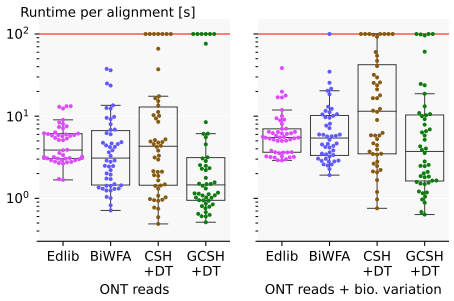
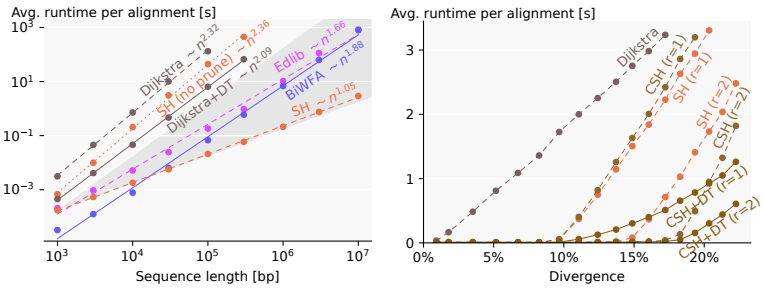


Fig. 5. Runtime on long human reads. Each dot is an alignment without (left) and with (right) genetic variation. Runtime is capped at 100 s. The median speedup of A*PA (GCSH with DT, k=15, r=2) is 2× (left) and 1.4× (right) over EDLIB and BIWFA.



(a) Runtime scaling with sequence length (b) Runtime scaling with divergence

Fig. 6. Runtime scaling comparison on synthetic data. (a) Scaling with length (log-log, synthetic, d=4%, 10^6 to 10^7 total bp, k=15, r=1). 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. (b) Runtime scaling with divergence (synthetic, n=10^4, 10^6 total bp, k=10). r=1 stands for exact matches, r=2 for inexact. GCSH (not shown) is as fast as CSH.

4.2 Comparison on synthetic data

Runtime scaling with length. In Fig. 4 we compare our A* heuristics with EDLIB and BIWFA in terms of runtime scaling with n and d. As expected from the analysis, EDLIB and BIWFA (Sec. 1.1) scale quadratically. An exception is EDLIB which is subquadratic for small edit distance due to the bit-parallel optimization. The empirical scaling of A*PA is subquadratic for d≤12 and n≤10^7, making it the fastest aligner for long sequences (n>30 kbp). For low divergence (d≤4%) even the simplest SH scales near-linearly with length (best fit $n^{1.07}$ for n≤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.1). Nevertheless, the scaling of GCSH with DT is not exactly linear due to state reordering and due to high memory usage Fig. 12a.

Performance. A*PA is >300× faster than EDLIB and BIWFA for d=4% and n=10^7 (Fig. 4b). For n=10^6 and d≤12%, memory usage is less than 500 MB for all heuristics (App. C.3).

4.3 Comparison on human data

We compare the different algorithms on human data for runtime (Fig. 5 and App. C.2), and memory usage (App. C.3). We configure A*PA to prune matches only when expanding their start (not their end). This leaves some matches on the optimal path unpruned which significantly speeds up the contour updates. When aligning ONT read errors, A*PA (GCSH with DT) is faster than EDLIB and BIWFA in all quartiles, with the median over 2× faster (1.45 s vs 3.08 s for BIWFA). However, the heuristic is not accurate enough to predict all errors for sequences with divergence d>10%, causing 6 alignments to time out.

When alignments include large gaps, A*PA has median runtime 1.4× lower than EDLIB and BIWFA. Compared to EDLIB, alignments with short indels are faster, while alignments with large indels are slower. This is caused by the need to expand quadratically many states around long gaps Fig. 7b. Three sequences with d≥10% time out. Since slow alignments dominate the overall runtime, EDLIB is faster on average.

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

We visualize the expanded regions for our heuristic variants for a single complex alignment in App. C.7.

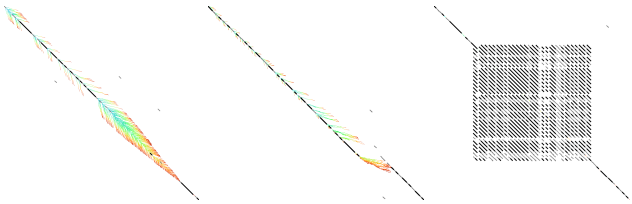
Pruning enables near-linear runtime. Figure 6a shows that match pruning changes the quadratic runtime of SH to near-linear, by penalizing the expansion of states before the search tip.

Inexact matches cope with higher divergence. Inexact matches increase the heuristic potential, allowing higher divergence (Fig. 6b). For low divergence, the runtime is nearly constant (App. C.4), 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. 6b), 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. 6a) and CSH 3× (Fig. 6b). 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.2).



(a) High divergence (b) Long indel (c) Short repeats

Fig. 7. Quadratic exploration behavior for complex alignments (GCSH with DT, r=2, k=10, synthetic sequences, n=1000). (a) A highly divergent region, (b) a deletion, and (c) a short repeated pattern inducing a quadratic number of matches. The colour corresponds to the order of expansion, from blue to red.

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.6).

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 (Fig. 7, App. C.7).
3. *Computational overhead of A*.* Despite the better scaling compared to EDLIB ($O(ns/w)$) and BIWFA (expected $O(n+s^2)$ for random sequences), A*PA is only faster for $n \geq 30\,000$ bp (Fig. 4). In the quadratic mode, A* is hundreds of times slower than others (compare Dijkstra to EDLIB and Dijkstra with DT to BIWFA in Fig. 6b).

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 family of chaining seed heuristics 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 behaviour of A* is not asymptotic and it requires a more thorough theoretical analysis (Medvedev, 2022a). Even though we presented an algorithm for near-linear time alignment of sequences with limited divergence and length, the fundamental question remains unanswered: *What class of sequences can be aligned in near-linear runtime?*

6 Conclusion

A*PA solves global pairwise alignment provably exact and empirically fast. To this end, we extended the existing seed heuristic for A* with inexact matches, match chaining, gap costs, match pruning, and diagonal-transition. We proved that our novel heuristics and optimizations are guaranteed to find an alignment with minimal edit distance.

On random sequences with $d \leq 8\%$ uniform divergence, the runtime of A*PA scales near-linearly to very long sequences ($n \leq 10^7$ bp) and outperforms other exact aligners. On long ONT reads of human data with $d \leq 10\%$, A*PA is over 2× faster than other aligners, and 1.4× faster when genetic variation is also present.

Acknowledgements

The authors are grateful to Mykola Akulov for his help with the visualizations (Figs. 1 and 3), to Daniel Liu for his involvement in developing PABENCH, and to Benjamin Bichsel, Maximilian Mordig and André Kahles for valuable comments on drafts. 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. (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.

A Pseudocode

A.1 A* algorithm for match pruning

We present our variant of A* (Hart *et al.*, 1968) that supports match pruning (Algorithm 1). All computed values of g are stored in a hash map, and all *open* states are stored in a bucket queue of tuples $(v, g(v), f(v))$ ordered by increasing f. Line 14 prunes (removes) a match and thereby increases some heuristic values before that match. As a result, some f-values in the priority queue may become outdated. Line 11 solves this problem by checking if the f-value of the state about to be expanded was changed, and if so, line 12 pushes the updated state to the queue, and proceeds by choosing a next best state. This way, we guarantee that the expanded state has minimal updated f. To reconstruct the best alignment we traceback from the target state using the hash map g (not shown).

A.2 Diagonal-transition optimization for A*

In Algorithm 2, we further modify A* to account for the DT optimization. We now skip opening and expanding states that are not farthest reaching (line 9). The map $g(u)$ that tracks the best distance to u in replaced by a map $F_{g,k}$ that tracks the farthest reaching state on diagonal k at distance g (lines 4, 18, and 19). Instead of $g(u)$ decreasing over time, we now ensure that $F_{g,k}$ increases over time.

A.3 Implementation notes

Here we list implementation details on performance and correctness.

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 value difference between 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 open state in the priority queue, but we do not cross seed boundaries in order to not interfere with match pruning. This optimization is superseded by the DT-optimization. We include greedily matched states in the reported number of expanded states.

Priority queue offset. Pruning the last remaining 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. We skip most of the update operations by storing a global offset to the f-values in the priority queue, which we update when all states in the priority queue precede u.

Split vector for layers. Pruning a match may trigger the removal of one or more layers of matches in L, and the shifting down of higher layers. To efficiently remove layers, we use a *split vector* data structure consisting of two stacks. In the first stack we store the layers before the last deleted layer, and in the second stack the remaining layers in reverse order. Before deleting 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. Removing layers in decreasing order of ℓ takes linear total time.

Binary search speedup. Instead of using binary search to determine the layer/score $S_p(u)$ (Algorithm 3), we first try a linear search around either the score of the parent of u or a known previous score at u. This linear search usually finds the correct layer in a few iterations, or otherwise we fall back to binary search.

In practice, most pruning happens near the tip of the search, and the number of layers between the start v_s and an open state u rarely changes. Thus, to account for changing scores, we store a *hint* of value $S_p(v_s) - S_p(u)$ in a hashmap and start the search at $S_p(v_s) - hint$.

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).

Algorithm 1 A* algorithm with match pruning.
Lines added for pruning (11, 12, and 14) are marked in **bold**.

```
1:  Input: Sequences A and B and pruning heuristic h
2:  Output: Edit distance between A and B
3:  function ASTAR(A, B, h)
4:      g(v_s) ← 0                          ▷ Hashmap of distances; default +∞
5:      q ← BucketQueue()                   ▷ Bucket queue of open states
6:      q.push((v_s, g=0, f=0))
7:      repeat
8:          (u, g_u, f_u) ← q.pop()         ▷ Pop u with minimal f
9:          if g_u > g(u) then
10:             continue                    ▷ u was already expanded
11:         else if f_u < g(u) + h(u) then  ▷ h(u) has increased
12:             q.push((u, g_u, g(u) + h(u)))   ▷ Reorder u
13:         else                            ▷ Expand u
14:             Prune(u)
15:             for successors v of u do
16:                 g_v ← g_u + d(u, v)
17:                 v ← Extend(v)           ▷ Greedy matching within seed
18:                 if g_v < g(v) then      ▷ Open v
19:                     g(v) ← g_v
20:                     q.push((v, g_v, g_v + h(v)))
21:     until v_t is expanded
22:     return g(v_t)
```

Algorithm 2 A*-DT algorithm with match pruning.
Lines changed for diagonal transition (4, 9, 18, and 19) are in **bold**.

```
1:  Input: Sequences A, B and pruning heuristic h
2:  Output: Edit distance between A and B
3:  function ASTAR-DT(A, B, h)
4:      F_{0,0} ← 0                         ▷ Hashmap of f.r. point per g and k; default −∞
5:      q ← BucketQueue()                   ▷ Bucket queue of open states
6:      q.push((v_s, g=0, f=0))
7:      repeat
8:          (u, g_u, f_u) ← q.pop()         ▷ Pop u with minimal f
9:          if i_u+j_u < F_{g_u, i_u−j_u} then
10:             continue                    ▷ u is not farthest reaching
11:         else if f_u < g(u) + h(u) then  ▷ h(u) has increased
12:             q.push((u, g_u, g(u) + h(u)))   ▷ Reorder u
13:         else                            ▷ Expand u
14:             Prune(u)
15:             for successors v of u do
16:                 g_v ← g_u + d(u, v)
17:                 v ← Extend(v)           ▷ Greedy matching in seed
18:                 if i_v+j_v > F_{g_v, i_v−j_v} then   ▷ Open v
19:                     F_{g_v, i_v−j_v} ← i_v+j_v
20:                     q.push((v, g_v, g_v + h(v)))
21:     until v_t is expanded
22:     return g(v_t)
```

A.4 Computation of the heuristic

Algorithm 3 shows how the heuristic is initialized, how $S_p(u)$ and $h(u)$ are computed, and how matches are pruned.

Algorithm 3 Computation of the heuristic.

Input: Sequences A and B
Output: Heuristic h
function INITIALIZEHEURISTIC(A, B, k)
  $S \leftarrow$ Non-overlapping k-mers of A ▷ Seeds
  $H \leftarrow$ Map of all k-mers (and $k{\pm}1$-mers for $r = 2$) of B
  $\mathcal{M}_s \leftarrow \bigcup_{s' \colon \mathrm{ed}(s,s') < r} H[s']$ ▷ Seed matches
  $\mathcal{M} \leftarrow \bigcup_{s \in \mathcal{S}} \mathcal{M}_s$ ▷ Matches
  $P(i) \leftarrow r \cdot |S_{\geq i}|$ for all $i \in [0, |A|]$ ▷ Potentials
  $L[0] = \{m_\omega\}$ ▷ Layers
  for $m \in \mathcal{M}$ in decreasing order of $\preceq_p$ **do**
    $\ell \leftarrow \mathrm{score}(m) + S_p(\mathrm{end}(m))$
    if $m \preceq_p m_\omega$ **then** ▷ If m precedes the target
      $L[\ell].push(m)$

function $S_p(u)$
  function TEST(ℓ)
    for $\ell' \in [\ell, \ell + r)$ **do**
      if $u \preceq_p m$ for some $m \in L[\ell']$ **then**
        return $True$
    return $False$
  return $\max\{\ell \mid \mathtt{TEST}(\ell)\}$ ▷ Binary search the highest ℓ

function $h_{\mathrm{s}}(u)$ ▷ SH
  return $P(u) - S_i(u)$

function $h_{\mathrm{cs}}(u)$ ▷ CSH
  return $P(u) - S_{\preceq}(u)$

function $h_{\mathrm{gcs}}(u)$ ▷ GCSH
  return $\max(P(u) - S_T(u), c_{\mathrm{gap}}(u, v_t))$

function PRUNE(u)
  $\ell_u \leftarrow S_p(u)$
  for all $m \in \mathcal{M}$: $start(m) = u$ **or** $end(m) = u$ **do**
    if $h \neq h_{\mathrm{gcs}}$ **or** $\mathcal{M} \backslash \{m\}$ is consistent **then**
      Remove m from $\mathcal{M}$
      for $\ell \in [S_p(start(m)) - r + 1, S_p(start(m))]$ **do**
        Remove m from $L[\ell]$ if present
  for $\ell \in [\ell_u + 1, S_p(0, 0)]$ **do**
    for all $m \in L[\ell]$ **do**
      $\ell' \leftarrow S_p(m)$
      if $\ell' \neq \ell$ **then**
        Move m from $L[\ell]$ to $L[\ell']$
    if r consecutive layers unchanged **then**
      return
    if $r{-}1{+}\ell{-}\ell'$ consecutive layers shifted exactly $\ell{-}\ell'$ **then**
      Remove empty layers $L[\ell'{+}1], \ldots, L[\ell]$
      ▷ Shift higher layers down
      return

B Proofs

B.1 Admissibility

Our heuristics are not *consistent*, but we show that a weaker variant holds for states *at the start of a seed*.

Definition 5 (Start of seed)**.** *A state $\langle i, j\rangle$ is* at the start of some seed *when i is a multiple of the seed length k, or when $i = n$.*

Lemma 1 (Weak triangle inequality)**.** *For states u, v, and w with v and w at the starts of some seeds, all $\gamma \in \{c_{\text{seed}}, c_{\text{gap}}, c_{\text{gs}}\}$ satisfy*

$$\gamma(u, v) + \gamma(v, w) \geq \gamma(u, v).$$

Proof. Both v and w are at the start of some seeds, so for $\gamma = c_{\text{seed}}$ we have the equality $c_{\text{seed}}(u, w) = c_{\text{seed}}(u, v) + c_{\text{seed}}(v, w)$.

For $\gamma = c_{\text{gap}}$,

$$\begin{aligned}
&c_{\text{gap}}(\langle i, j\rangle, \langle i', j'\rangle) + c_{\text{gap}}(\langle i', j'\rangle, \langle i'', j''\rangle) \\
&= |(i' - i) - (j' - j)| + |(i'' - i') - (j'' - j')| \\
&\geq |(i'' - i) - (j'' - j)| = c_{\text{gap}}(\langle i, j\rangle, \langle i'', j''\rangle).
\end{aligned}$$

And lastly, for $\gamma = c_{\text{gs}}$,

$$\begin{aligned}
&c_{\text{gs}}(u, v) + c_{\text{gs}}(v, w) \\
&= \max(c_{\text{gap}}(u, v), c_{\text{seed}}(u, v)) + \max(c_{\text{gap}}(v, w), c_{\text{seed}}(v, w)) \\
&\geq \max(c_{\text{gap}}(u, v) + c_{\text{gap}}(v, w), c_{\text{seed}}(u, v) + c_{\text{seed}}(v, w)) \\
&\geq \max(c_{\text{gap}}(u, w), c_{\text{seed}}(u, w)) = c_{\text{gs}}(u, w).
\end{aligned}$$

□

Lemma 2 (Weak consistency)**.** *Let $h \in \{h_{\text{s}}^{\mathcal{M}}, h_{\text{cs}}^{\mathcal{M}}, h_{\text{gcs}}^{\mathcal{M}}\}$ be a heuristic with partial order $\preceq_p$, and let $u \preceq_p v$ be states with v at the start of a seed. When there is a shortest path π^* from u to v such that $\mathcal{M}$ contains all matches of cost less than r on π^*, it holds that $h(u) \leq \mathrm{d}(u, v) + h(v)$.*

Proof. The path π^* covers each seed in $\mathcal{S}_{u\ldots v}$ that must to be fully aligned between u and v. Since the seeds do not overlap, their shortest alignments π_s^* in π^* do not have overlapping edges. Let $u \preceq m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v$ be the chain of matches $m_i \in \mathcal{M}$ corresponding to those π_s^* of cost less than r (Fig. 8). Since the matches and the paths between them are disjoint, $\mathrm{c}_{\text{path}}(\pi^*)$ is at least the cost of the matches $\mathrm{c}_{\text{m}}(m_{i+1}) = \mathrm{d}(\text{start}(m_{i+1}), \text{end}(m_{i+1}))$ plus the cost to chain these matches $\gamma(\text{end}(m_i), \text{start}(m_{i+1})) \leq \mathrm{d}(\text{end}(m_i), \text{start}(m_{i+1}))$. Putting this together:

$$\begin{aligned}
&\gamma(u, m_1) + \mathrm{c}_{\text{m}}(m_1) + \cdots + \mathrm{c}_{\text{m}}(m_l) + \gamma(m_l, v) \\
&\leq \mathrm{d}(u, \text{start}(m_1)) + \mathrm{d}(\text{start}(m_1), \text{end}(m_1)) + \cdots + \mathrm{d}(\text{end}(m_l), v) \\
&\leq \mathrm{d}(u, v).
\end{aligned}$$

Now let $v \preceq_p m_{l+1} \preceq_p \cdots \preceq_p m_{l'} \preceq_p v_t$ be a chain of matches minimizing $h(v)$ (Def. 1) with $w := \text{start}(m_{l+1})$. This chain also minimizes $h(w)$ and thus $h(v) = \gamma(v, w) + h(w)$. We can now bound the cost of the joined chain from u to v and from w to the end and get our result via $\gamma(m_l, w) \leq \gamma(m_l, v) + \gamma(v, w)$ (Lemma 1)

$$\begin{aligned}
h(u) &\leq \gamma(u, m_1) + \cdots + \gamma(m_l, m_{l+1}) + \mathrm{c}_{\text{m}}(m_{l+1}) + \cdots + \gamma(m_{l'}, v_t) \\
&= \gamma(u, m_1) + \cdots + \gamma(m_l, w) \qquad + h(w) \\
&\leq \gamma(u, m_1) + \cdots + \gamma(m_l, v) + \gamma(v, w) + (h(v) - \gamma(v, w)) \\
&\leq \mathrm{d}(u, v) + h(v).
\end{aligned}$$

□

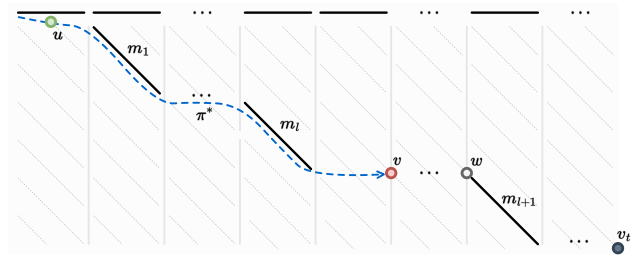


Fig. 8. Variables of the proof of Lemma 2.

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

Proof. We will prove $h_{\text{s}}^{\mathcal{M}}(u) \overset{(1)}{\leq} h_{\text{cs}}^{\mathcal{M}}(u) \overset{(2)}{\leq} h_{\text{gcs}}^{\mathcal{M}}(u) \overset{(3)}{\leq} h^*(u)$, which implies the admissibility of all three heuristics.

(1) Note that $u \preceq v$ implies $u \preceq_i v$ and hence any $\preceq$-chain is also a $\preceq_i$-chain. A minimum over the superset of $\preceq_i$-chains is at most the minimum of the subset of $\preceq$-chains, and hence $h_{\text{s}}^{\mathcal{M}} = h_{\preceq_i, c_{\text{seed}}}^{\mathcal{M}} \leq h_{\preceq, c_{\text{seed}}}^{\mathcal{M}} = h_{\text{cs}}^{\mathcal{M}}$.

(2) The only difference between $h_{\text{cs}}^{\mathcal{M}}$ and $h_{\text{gcs}}^{\mathcal{M}}$ is that the former uses c_{seed} and the latter uses the gap-seed cost $c_{\text{gs}} := \max(c_{\text{gap}}, c_{\text{seed}})$. Since $c_{\text{seed}} \leq c_{\text{gs}}$ we have $h_{\text{cs}}^{\mathcal{M}} = h_{\preceq, c_{\text{seed}}}^{\mathcal{M}} \leq h_{\preceq, c_{\text{gs}}}^{\mathcal{M}} = h_{\text{gcs}}^{\mathcal{M}}$.

(3) When $\mathcal{M}$ is the set of all matches with costs strictly less than r, admissibility follows directly from Lemma 2 with $v = v_t$ via

$$h_{\text{gcs}}^{\mathcal{M}}(u) \leq \mathrm{d}(u, v_t) + h_{\text{gcs}}^{\mathcal{M}}(v_t) = \mathrm{d}(u, v_t) = h^*(u).$$

□

B.2 Match pruning

During the A* search, we continuously improve our heuristic using match pruning. The pruning increases the value of our heuristics and breaks their admissibility. Nevertheless, we prove in two steps that A* with match pruning still finds a shortest path. First, we introduce the concept of a *weakly-admissible heuristic* and show that A* using a weakly-admissible heuristic finds a shortest path (Thm. 2). Then, we show that our pruning heuristics are indeed weakly admissible (Thm. 3).

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

Definition 6 (Fixed vertex)**.** *A vertex u is* fixed *if it is expanded and A* has found a shortest path to it, that is, $g(u) = g^*(u)$.*

A fixed vertex cannot be opened again (Algorithm 1, line 18), and hence remains fixed.

Definition 7 (Weakly admissible)**.** *A heuristic $\hat{h}$ is* weakly admissible *if at any moment during the A* search there exists a shortest path π^* from v_s to v_t in which all vertices $u \in \pi^*$ after its last fixed vertex n^* satisfy $\hat{h}(u) \leq h^*(u)$.*

To prove that A* finds a shortest path when used with a weakly-admissible heuristic, we follow the structure of Hart *et al.* (1968). First we restate their Lemma 1 in our notation with a slightly stronger conclusion that follows directly from their proof.

Lemma 3 (Lemma 1 of Hart *et al.* (1968))**.** *For any unfixed vertex n and for any shortest path π^* from v_s to n, there exists an open vertex n' on π^* with $g(n') = g^*(n')$ such that π^* does not contain fixed vertices after n'.*

Next, we prove that in each step the A* algorithm can proceed along a shortest path to the target:

Corollary 1 (Generalization of Corollary to Lemma 1 of Hart *et al.* (1968))**.** *Suppose that $\hat{h}$ is weakly admissible, and that A* has not*

terminated. Then, there exists an open vertex n' on a shortest path from v_s to v_t with $f(n') \le g^*(v_t)$.

Proof. Let π^* be the shortest path from v_s to v_t given by the weak admissibility of $\hat{h}$ (Def. 7). Since A* has not terminated, v_t is not fixed. Substitute $n = v_t$ in Lemma 3 to derive that there exists an open vertex n' on π^* with $g(n') = g^*(n')$. By definition of f we have $f(n') = g(n') + \hat{h}(n')$. Since π^* does not contain any fixed vertices after n', the weak admissibility of $\hat{h}$ implies $\hat{h}(n') \le h^*(n')$. Thus, $f(n') = g(n') + \hat{h}(n') \le g^*(n') + h^*(n') = g^*(v_t)$. □

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

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

Our heuristics are weakly admissible. A *consistent* heuristic finds the correct distance to each vertex as soon as it is expanded. While our heuristics are not consistent, this property is true for states *at the starts of seeds* (when i is a multiple of the seed length k, or when $i = n$).

Lemma 4. *For $\hat{h} \in \{\hat{h}_{\mathrm{s}}, \hat{h}_{\mathrm{cs}}, \hat{h}_{\mathrm{gcs}}\}$, every state at the start of a seed becomes fixed immediately when A* expands it.*

Proof. We use a proof by contradiction: suppose that v is a state at the start of some seed that is expanded but not fixed. In other words, $f(v)$ is minimal among all open states, but the shortest path π^* from v_s to v has strictly smaller length $g^*(v) < g(v)$.

Let n^* be the last fixed state on π^* before v, and let $u \in \pi^*$ be the successor of n^*. State u is open because its predecessor n^* is fixed and on a shortest path to u. Let the chain of all matches of cost less than r on π^* between u and v be $u \le m_1 \le \cdots \le m_l \le v$. Since n^* is the last fixed state on π^*, none of these matches has been pruned, and they are all in $\mathcal{M} \backslash E$ as well. This means we can apply Lemma 2 to get $h(u) \le \mathrm{d}(u, v) + h(v)$, so

$$\begin{aligned} f(u) = g(u) + h(u) &= g^*(u) + h(u) && \pi^* \text{ is a shortest path} \\ &\le g^*(u) + \mathrm{d}(u, v) + h(v) && \text{shown above} \\ &= g^*(v) + h(v) && \pi^* \text{ is a shortest path} \\ &< g(v) + h(v) = f(v) && \text{by assumption.} \end{aligned}$$

This proves that $f(u) < f(v)$, resulting in a contradiction with the assumption that u is an open state with minimal f. □

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

Proof. Let π^* be a shortest path from v_s to v_t. At any point during the A* search, let n^* be the farthest expanded state on π^* that is at the start of a seed. By Lemma 4, n^* is fixed. By the choice of n^*, no states on π^* after n^* that are at the start of a seed are expanded, so no matches on π^* following n^* are pruned. Now the proof of Thm. 1 applies to the part of π^* after n^* without changes, implying that $\hat{h}(u) \le h^*(u)$ for all u on π^* following n^*, for any $\hat{h} \in \{\hat{h}_{\mathrm{s}}, \hat{h}_{\mathrm{cs}}, \hat{h}_{\mathrm{gcs}}\}$. □

B.3 Computation of the (chaining) seed heuristic

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 $u \preceq_i v$).*

Proof. For a chain of matches $\{m_i\} \subseteq \mathcal{M}$, let s_i and t_i be the start and end states of m_i. We translate the terms of our heuristic from costs to potentials and match scores (Sec. 3.5):

$$\begin{aligned} & c_{\mathrm{seed}}(m_i, m_{i+1}) + \mathrm{c_m}(m_{i+1}) \\ = \; & \big(P(t_i) - P(s_{i+1})\big) + \big(P(s_{i+1}) - P(t_{i+1}) - \mathrm{score}(m_{i+1})\big) \\ = \; & P(t_i) - P(t_{i+1}) - \mathrm{score}(m_{i+1}). \end{aligned}$$

The heuristic (Def. 1) can then be rewritten as follows:

$$\begin{aligned} & h^{\mathcal{M}}_{p,c_{\mathrm{seed}}}(u) \\ &= \min_{\substack{u \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t \\ m_i \in \mathcal{M}}} \sum_{0 \le i \le l} \big[P(t_i) - P(t_{i+1}) - \mathrm{score}(m_{i+1})\big] \\ &= P(u) - \max_{\substack{u \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t \\ m_i \in \mathcal{M}}} \sum_{0 \le i \le l} \mathrm{score}(m_{i+1}) \\ &= P(u) - S_p(u). \end{aligned}$$

□

Lemma 5. *Layer $\mathcal{L}_\ell$ ($\ell > 0$) is fully determined by the set of those matches m for which $\ell \le S_p(m) < \ell + r$:*

$$\mathcal{L}_\ell = \{u \mid \exists m \in \mathcal{M} : u \preceq_p m \text{ and } S_p(m) \in [\ell, \ell + r)\}.$$

Proof. Take any state $u \in \mathcal{L}_\ell$. Its score $S_p(u) \ge \ell > 0$ implies that there is a non-empty $\preceq_p$-chain $u \preceq_p m_1 \preceq_p m_2 \preceq_p \ldots$ with $\mathrm{score}(m_1) + \mathrm{score}(m_2) + \cdots \ge \ell$. The score of each match is less than r and thus there must be a match m_i so that the subset of the chain starting at m_i has score $S_p(m_i) = \mathrm{score}(m_i) + \mathrm{score}(m_{i+1}) + \cdots +$ in the interval $[\ell, \ell + r)$. This implies that for any u with score $S_p(u) \ge \ell > 0$ there is a match with score in $[\ell, \ell + r)$ succeeding u, as required. □

B.4 Computation of the gap-chaining seed heuristic

In this section we prove (Thm. 5) that we can change the dependency of GCSH on c_{gs} to c_{seed} by introducing a new partial order $\preceq_T$ on the matches. This way, Thm. 4 applies and we can efficiently compute GCSH. Recall that the gap-seed cost is $c_{\mathrm{gs}} = \max(c_{\mathrm{gap}}, c_{\mathrm{seed}})$, and that the gap transformation is:

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

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

The following lemma allows us to determine whether c_{gap} or c_{seed} dominates the cost c_{gs} between two matches, based on the relation $\preceq_T$.

Lemma 6. *Let u and v be two states with v at the start of some seed. Then $u \preceq_T v$ if and only if $c_{\mathrm{gap}}(u, v) \le c_{\mathrm{seed}}(u, v)$. Furthermore, $u \preceq_T v$ implies $u \preceq v$.*

Proof. Let $u = \langle i, j\rangle$ and $v = \langle i', j'\rangle$. By definition, $u \preceq_T v$ is equivalent to

$$\begin{cases} i - j - P(u) \le i' - j' - P(v) \\ j - i - P(u) \le j' - i' - P(v) \end{cases} \Leftrightarrow \begin{cases} -\big((i' - i) - (j' - j)\big) \le P(u) - P(v) \\ (i' - i) - (j' - j) \le P(u) - P(v). \end{cases}$$

This simplifies to

$$c_{\mathrm{gap}}(u, v) = |(i' - i) - (j' - j)| \le P(u) - P(v) = c_{\mathrm{seed}}(u, v),$$

where the last equality holds because v is at the start of a seed.

For the second part, $u \preceq_T v$ implies $0 \le c_{\mathrm{gap}}(u, v) \le c_{\mathrm{seed}}(u, v) = P(u) - P(v)$ and hence $P(u) \ge P(v)$. Since v is at the start of a seed, this directly implies $i \le i'$. Since seeds have length $k \ge r$ we have

$$|(i' - i) - (j' - j)| \le P(u) - P(v) = r \cdot |\mathcal{S}_{i \ldots i'}| \le r \cdot (i' - i)/k \le i' - i.$$

This implies $j' - j \ge 0$ and hence $j \le j'$, as required. □

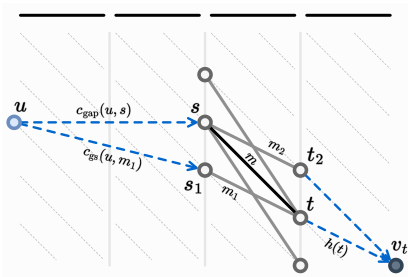


Fig. 9. Variables in Case 2 of the proof of Lemma 7. Match m has $\text{score}(m) = 2$, so it has adjacent matches m_1 and m_2 (gray) with $\text{score}(m_i) \geq 1$.

A direct corollary is that for $u \preceq v$ with v at the start of some seed, we have

$$c_{\text{gs}}(u,v) = \begin{cases} c_{\text{seed}}(u,v) & \text{if } u \preceq_T v, \\ c_{\text{gap}}(u,v) & \text{if } u \not\preceq_T v. \end{cases} \tag{5}$$

A second corollary is that $\text{start}(m) \preceq_T \text{end}(m)$ for all matches $m \in \mathcal{M}$, since a match from u to v satisfies $c_{\text{gap}}(u,v) < r = c_{\text{seed}}(u,v)$ by definition.

Lemma 7. *When the set of matches $\mathcal{M}$ is consistent, $h^{\mathcal{M}}_{\text{gcs}}(u)$ can be computed using $\preceq_T$-chains only:*

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

Proof. We write $h := h^{\mathcal{M}}_{\text{gcs}} = h^{\mathcal{M}}_{\preceq,c_{\text{gs}}}$ (Def. 1) and $h' := h^{\mathcal{M}}_{\preceq_T,c_{\text{gs}}}$.

Case 1: $u \not\preceq_T v_t$. Let $u \preceq m_1 \preceq \cdots \preceq m_l \preceq v_t$ be a chain minimizing $h(u)$ in Def. 1, so

$$h(u) = c_{\text{gs}}(u,m_1) + \text{c}_{\text{m}}(m_1) + c_{\text{gs}}(m_1,m_2) + \cdots + c_{\text{gs}}(m_l,v_t).$$

By definition, $c_{\text{gs}} \geq c_{\text{gap}}$, and $\text{c}_{\text{m}}(m_i) \geq c_{\text{gap}}(\text{start}(m_i),\text{end}(m_i))$, so the weak triangle inequality (Lemma 1) for c_{gap} gives

$$h(u) \geq c_{\text{gap}}(u,m_1) + c_{\text{gap}}(m_1) + \cdots + c_{\text{gap}}(m_l,v_t) \geq c_{\text{gap}}(u,v_t).$$

Since $u \not\preceq_T v_t$, the empty chain $u \preceq v_t$ has cost $h(u) \leq c_{\text{gs}}(u,v_t) = c_{\text{gap}}(u,v_t)$. Combining the two inequalities, $h(u) = c_{\text{gap}}(u,v_t)$.

Case 2: $u \preceq_T v_t$. First rewrite h and h' recursively as

$$h(u) = \min_{\substack{m \in \mathcal{M} \\ u \preceq m \preceq v_t}} \big(c_{\text{gs}}(u,m) + \text{c}_{\text{m}}(m) + h(\text{end}(m))\big) \tag{6}$$

$$h'(u) = \min_{\substack{m \in \mathcal{M} \\ u \preceq_T m \preceq_T v_t}} \big(c_{\text{gs}}(u,m) + \text{c}_{\text{m}}(m) + h'(\text{end}(m))\big), \tag{7}$$

both with base case $h(v_t) = h'(v_t) = 0$ after eventually taking m_ω. We will show that

$$h(u) = \min_{\substack{m \in \mathcal{M} \\ u \preceq_T m \preceq_T v_t}} \big(c_{\text{gs}}(u,m) + \text{c}_{\text{m}}(m) + h(\text{end}(m))\big), \tag{8}$$

which is exactly the recursion for h', so that by induction $h(u) = h'(u)$.

By Lemma 6, every $\preceq_T$-chain is a $\preceq$-chain, so $h(u) \leq h'(u)$. To prove $h(u) = h'(u)$, it remains to show the reverse inequality, $h'(u) \leq h(u)$. To this end, choose a match m that

(priority 0) minimizes $h(u)$ in Eq. (6), and among those, has
(priority 1) maximal $c_{\text{seed}}(u,m)$, and among those, has
(priority 2) minimal $c_{\text{gap}}(u,m)$, and among those, has
(priority 3) minimal $c_{\text{gap}}(m,v_t)$.

We show that $u \preceq_T m$ (in 2.A) and $m \preceq_T v_t$ (in 2.B), which proves Eq. (8).

Part 2.A: $u \preceq_T m$. Let s and t be the begin and end of m (Fig. 9), and let m' be a match minimizing $h(t)$ in Eq. (6) so

$$h(t) = c_{\text{gs}}(t,m') + \text{c}_{\text{m}}(m') + h(\text{end}(m')).$$

Since m' comes after t we have $c_{\text{seed}}(u,m') > c_{\text{seed}}(u,m)$ (p.1) and hence m' does not minimize $h(u)$ (p.0):

$$h(u) < c_{\text{gs}}(u,m') + \text{c}_{\text{m}}(m') + h(\text{end}(m')).$$

Using the minimality of m, the non-minimality of m', and the triangle inequality we get

$$\begin{aligned} c_{\text{gs}}(u,s) + \text{c}_{\text{m}}(m) + h(t) &= h(u) \\ &< c_{\text{gs}}(u,m') \qquad\qquad + \text{c}_{\text{m}}(m') + h(\text{end}(m')) \\ &\leq c_{\text{gs}}(u,t) + c_{\text{gs}}(t,m') + \text{c}_{\text{m}}(m') + h(\text{end}(m')) \\ &= c_{\text{gs}}(u,t) + h(t) \end{aligned}$$

so we have

$$c_{\text{gs}}(u,m) + \text{c}_{\text{m}}(m) < c_{\text{gs}}(u,t). \tag{9}$$

From the triangle inequality for c_{gap}, from $c_{\text{gap}}(u,s) \leq c_{\text{gs}}(u,s)$, and from $c_{\text{gap}}(s,t) \leq \text{c}_{\text{m}}(m)$, and from Eq. (9) we obtain

$$c_{\text{gap}}(u,t) \leq c_{\text{gap}}(u,s) + c_{\text{gap}}(s,t) \leq c_{\text{gs}}(u,s) + \text{c}_{\text{m}}(m) < c_{\text{gs}}(u,t).$$

This implies $c_{\text{gs}}(u,t) = c_{\text{seed}}(u,t)$ and hence reusing Eq. (9)

$$\begin{aligned} c_{\text{gap}}(u,s) + \text{c}_{\text{m}}(m) &\leq c_{\text{gs}}(u,s) + \text{c}_{\text{m}}(m) \\ &< c_{\text{seed}}(u,t) = c_{\text{seed}}(u,s) + c_{\text{seed}}(s,t). \end{aligned}$$

We have $\text{c}_{\text{m}}(m) = c_{\text{seed}}(s,t) - \text{score}(m)$, so the above simplifies to $c_{\text{gap}}(u,s) < c_{\text{seed}}(u,s) + \text{score}(m)$ and since these are integers $c_{\text{gap}}(u,s) \leq c_{\text{seed}}(u,s) + \text{score}(m) - 1$.

When $\text{score}(m) = 1$, this implies $c_{\text{gap}}(u,s) \leq c_{\text{seed}}(u,s)$ and $u \preceq_T s = \text{start}(m)$ by Lemma 6.

When $\text{score}(m) > 1$, suppose that $c_{\text{gap}}(u,s) > c_{\text{seed}}(u,s) \geq 0$. That means that u is either above or below the diagonal of s. Let $s_1 = \langle s_i, s_j \pm 1 \rangle$ be the state adjacent to s on the same side of this diagonal as u. This state exists since $u \preceq s_1 \preceq t$. Then $c_{\text{gap}}(u,s_1) = c_{\text{gap}}(u,s) - 1$, and by consistency of $\mathcal{M}$ there is a match m_1 from s_1 to t with $\text{c}_{\text{m}}(m_1) \leq \text{c}_{\text{m}}(m) + 1$. Then

$$\begin{aligned} &c_{\text{gs}}(u,s_1) + \text{c}_{\text{m}}(m_1) + h(t) \\ &\leq c_{\text{gs}}(u,s) - 1 + \text{c}_{\text{m}}(m) + 1 + h(t) = h(u), \end{aligned}$$

showing that m_1 minimizes $h(u)$ (p.0). Also $c_{\text{seed}}(u,m_1) = c_{\text{seed}}(u,m_1)$ (p.1) and $c_{\text{gap}}(u,m_1) < c_{\text{gap}}(u,m)$ (p.2), so that m_1 contradicts the minimality of m. Thus, $c_{\text{gap}}(u,s) > c_{\text{seed}}(u,s)$ is impossible and $u \preceq_T s$.

Part 2.B: $m \preceq_T v_t$. When there is some match m' succeeding m in the chain, we have $m \preceq m' \preceq v_t$ and hence $m \preceq v_t$. Thus, suppose that m is the only match in the chain $u \preceq m \preceq v_t$ minimizing $h(u)$. We repeat the proofs of Part 2.A in the reverse direction to show that $m \preceq_T v_t$.

Since $c_{\text{seed}}(u, m)$ is maximal, $h(u) < c_{\text{gs}}(u, m_\omega)$ and thus

$$h(u) = c_{\text{gs}}(u, s) + \text{c}_{\text{m}}(m) + c_{\text{gs}}(m, v_t) < c_{\text{gs}}(u, v_t).$$

By assumption we have $u \preceq_T v_t$ and thus $c_{\text{gs}}(u, v_t) = c_{\text{seed}}(u, v_t)$. This gives

$$\begin{aligned} c_{\text{seed}}(u, s) + c_{\text{seed}}(s, t) - \text{score}(m) + c_{\text{gap}}(t, v_t) \\ < c_{\text{seed}}(u, v_t) = c_{\text{seed}}(u, s) + c_{\text{seed}}(s, t) + c_{\text{seed}}(t, v_t). \end{aligned}$$

Cancelling terms we obtain $c_{\text{gap}}(t, v_t) < c_{\text{seed}}(t, v_t) + 1$ and since they are integers $c_{\text{gap}}(t, v_t) \leq c_{\text{seed}}(t, v_t) + \text{score}(m) - 1$. When $\text{score}(m) = 1$, this implies $t \preceq_T v_t$, as required.

When $\text{score}(m) > 1$, suppose that $c_{\text{gap}}(t, v_t) > c_{\text{seed}}(t, v_t)$. Let $t_2 = \langle t_i, t_j \pm 1 \rangle$ be the state adjacent to t on the same side of the diagonal as v_t. By consistency of $\mathcal{M}$ there is a match m_2 from s to t_2 with $\text{score}(m_2) \geq \text{score}(m) - 1$ and $c_{\text{gap}}(t_2, v_t) = c_{\text{gap}}(t, v_t) - 1$. Then m_2 minimizes $h(u)$ (p.0)

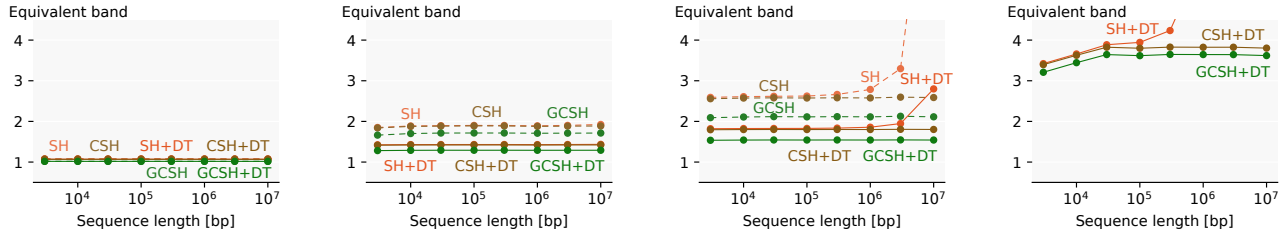
$$\begin{aligned} c_{\text{gs}}(u, s) + \text{c}_{\text{m}}(m_2) + c_{\text{gs}}(t_2, v_t) \\ \leq c_{\text{gs}}(u, s) + \text{c}_{\text{m}}(m) + 1 + c_{\text{gs}}(t, v_t) - 1 = h(u), \end{aligned}$$

and furthermore $c_{\text{seed}}(u, m_2) = c_{\text{seed}}(u, m)$ (p.1), $c_{\text{gap}}(u, m_2) = c_{\text{gap}}(u, m)$ (p.2), and $c_{\text{gap}}(m_2, v_t) < c_{\text{gap}}(m, v_t)$ (p.3), contradicting the choice of m, so $c_{\text{gap}}(t, v_t) > c_{\text{seed}}(t, v_t)$ is impossible and $t \preceq_T v_t$. □

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

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

Proof. Write $h := h^{\mathcal{M}}_{\text{gcs}}$ and $h' := h^{\mathcal{M}}_{\preceq_T, c_{\text{gs}}}$. When $u \not\preceq_T v_t$, $h(u) = c_{\text{gap}}(u, v_t)$ by Lemma 7. Otherwise, when $u \preceq_T v_t$, we have $h(u) = h'(u)$. Let $u \preceq_T m_1 \preceq_T \ldots \preceq_T v_t$ be a $\preceq_T$-chain for h' as in Def. 1. All terms in h' satisfy $\text{end}(m_i) \preceq_T \text{start}(m_{i+1})$, so $c_{\text{gap}} \leq c_{\text{seed}}$ and by Lemma 6 $c_{\text{gs}}(m_i, m_{i+1}) = c_{\text{seed}}(m_i, m_{i+1})$. Thus, $h' = h_{\preceq_T, c_{\text{seed}}}$, and $h^{\mathcal{M}}_{\text{gcs}}(u) = P(u) - S_T(u)$ by Thm. 4. □



(a) d=1%, exact matches (b) d=4%, exact matches (c) d=8%, inexact matches (d) d=12%, inexact matches

Fig. 10. Equivalent band scaling with sequence length on synthetic data. (k=15, r=2 for d≥8%). The equivalent band is the number of expanded states per bp for aligning synthetic sequences. Averages are over total $N=10^7$ bp. At d=12%, the bands of non-DT methods are ≥ 3× wider.

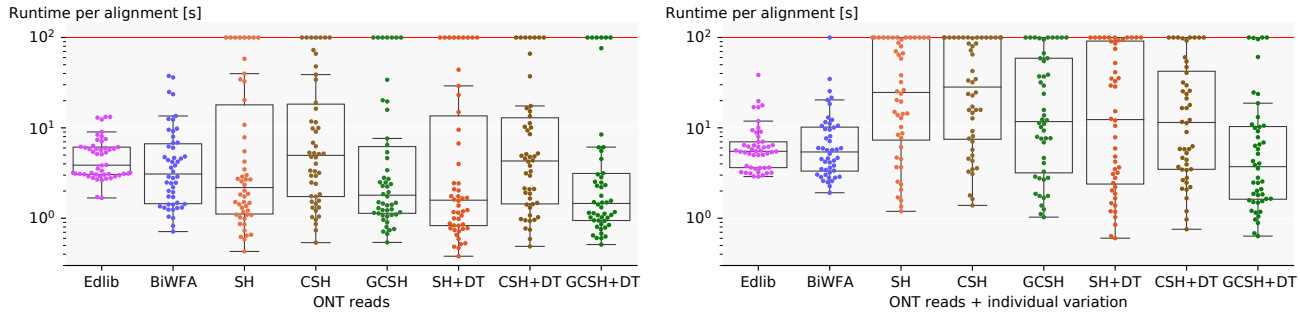


Fig. 11. Runtime on long reads of human data (logarithmic, ≥ 500 kbp). Each dot corresponds to an alignment of a sequence pair. The ONT reads are without (left) and with (right) genetic variation. Runtime is capped at 100 s.

C Further results

C.1 Expanded states and equivalent band

The main benefit of an A* heuristic is a lower number of expanded states, which translates to faster runtime. Instead of evaluating the runtime scaling with length (Fig. 4), we can judge how well a heuristic approximates the edit distance by directly measuring the *equivalent band* (Fig. 10) of each alignment: the number of expanded states divided by sequence length n, or equivalently, the number of expanded states per base pair. The theoretical lower bound is an equivalent band of 1, resulting from expanding only the states on the main diagonal.

The equivalent band tends to be constant in n, indicating that the number of expanded states is linear on the given domain. The equivalent band of SH starts to grow around $n \geq 10^6$ at divergence d=8%, and around $n \geq 10^5$ at d=12%. Because of the chaining, CSH and GCSH cope with spurious matches and remain constant in equivalent band (i.e. linear expanded states with n). The equivalent band for GCSH is slightly lower than CSH due to better accounting for indels. The DT variants expand fewer states by skipping non-farthest reaching states.

C.2 Human data results

We compare all our heuristics on human data (Fig. 11).

For ONT reads without genetic variation, SH is faster than other methods in median, but slower for sequences with high divergence and larger gaps. CSH is slightly slower than SH due to additional bookkeeping without significant benefit for the heuristic. Penalizing indels with GCSH improves performance several times.

When genetic variation is included, SH and CSH do not sufficiently penalize long gaps, and A* with GCSH estimates the remaining edit distance significantly better. The diagonal transition optimization considerably speeds up the alignment of long indels where the search regresses to quadratic.

C.3 Memory usage

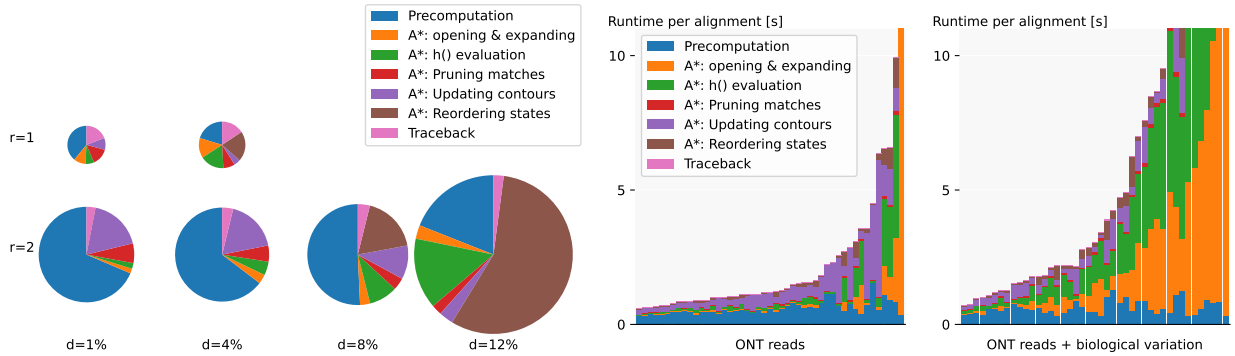
Table 3 compares memory usage of aligners on synthetic sequences of length $n=10^6$. Alignments with inexact matches (d≥8%) use significantly more memory than those with exact matches because more k-mer hashes need to be stored to find inexact matches. Table 4 compares memory usage on the human data sets.

	Memory usage [MB]			
Aligner	d=1%	d=4%	d=8%	d=12%
EDLIB	2	1	2	2
BIWFA	15	13	14	18
SH	50	59	151	480
CSH	52	59	151	261
GCSH	49	50	151	255
SH + DT	49	49	151	150
CSH + DT	49	49	151	150
GCSH + DT	48	46	152	150

Table 3. Memory usage per algorithm (synthetic data, $n=10^6$). Exact matches are used when $d \leq 4\%$, and inexact matches when $d \geq 8\%$.

	ONT reads		+ ind. variation	
Aligner	Median	Max	Median	Max
EDLIB	2	5	2	6
BIWFA	11	19	15	24
A*PA (GCSH + DT)	160	3478	270	6926

Table 4. Memory usage [MB] of aligners on human data. Medians are over all alignments; maximums are over alignments not timing out.



(a) Synthetic data

(b) Human data

Fig. 12. Runtime distributions per stage of A*PA (GCSH with DT) (stages do not overlap). Stage *A** includes expanding and opening states. *Pruning matches* includes consistency checks. *Updating contours* includes updating of contours after pruning. (a) On synthetic data ($n{=}10^6$ bp, $N{=}10^7$ bp total). The circle area is proportional to the total runtime. Figures for $r{=}1$ and $d{\geq}8\%$ are skipped due to timeouts (100 s). (b) On human data ($r{=}2$). Alignments are sorted by total runtime (timeouts not shown).

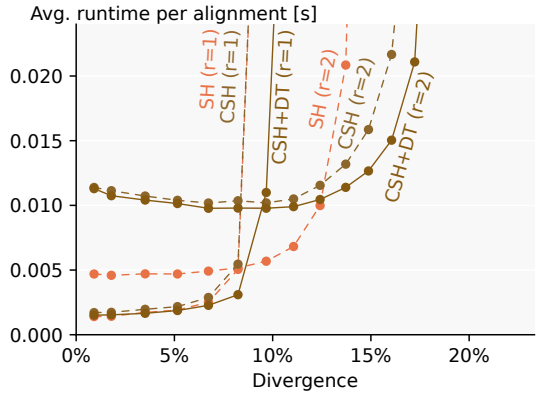


Fig. 13. Runtime scaling with divergence (zoomed) (linear, synthetic, $n{=}10^4$, 10^6 bp total, $k{=}10$). The figure shows the same results as Fig. 6b, but zoomed in to small runtimes where the scaling with n degrades from linear to quadratic. $r{=}1$ and $r{=}2$ indicate exact and inexact matches, respectively.

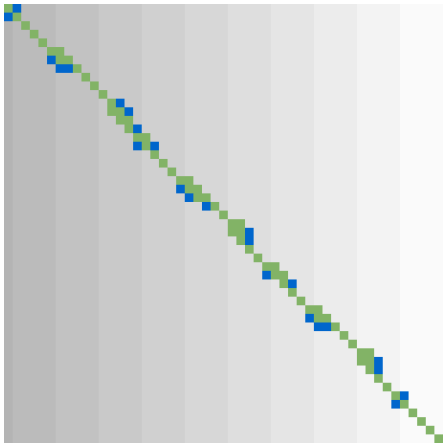


Fig. 14. Artificial example of A* with seed heuristic with no matches ($n{=}m{=}50$, $r{=}1$, $k{=}5$, 80% similarity, 1 mutation per seed alternating substitution, insertion, and deletion). The background colour indicates $h_s(u)$ with higher values darker. Expanded states are green (■), open states blue (■).

C.4 Scaling with divergence

Figure 13 shows the runtime scaling with divergence for various heuristics. The heuristics using inexact matches ($r{=}2$) are slower for low-divergence sequences due to spending relatively a lot of time on the initialization of the heuristic, which requires finding all inexact matches.

The threshold for near-constant runtime is near $10\% = 1/k$ for exact matches and near $20\% = 2/k$ for inexact matches. As the divergence approaches this threshold, the runtime (and number of expanded states) starts to grow. This can be explained as follows.

For each error that is not accounted for by a heuristic, A* needs to increase its best estimate f of the total cost by 1. This means it needs to *backtrack* and expand more states with this higher f value, including states before the search tip. Pruning compensates for this by increasing h before the tip. However, the more unaccounted errors occur, the more pruned matches are needed. Once the total cost is equal to the potential, *all* pruned matches are needed to compensate. At that point, each edit that is not accounted for effectively increases the band width by 2.

C.5 Runtime of A*PA internals

Figures 12a and 12b compare the time used by stages of A*PA.

C.6 Linear mode without matches

States are penalized by the number of remaining seeds that cannot be matched. So, curiously, matches are not always needed to direct the A* search to an optimal path. In fact, when each seed contains exactly one mutation seed heuristic scales linearly even though there are no matches, as shown by the artificial example in Fig. 14.

C.7 Comparison of heuristics and techniques

Figure 15 shows the effect of our heuristics and optimizations for aligning complex short sequences. The effect of pruning is most noticeable for CSH and GCSH without DT. GCSH is our most accurate heuristic, so, as expected, it leads to the lowest number of expanded states.

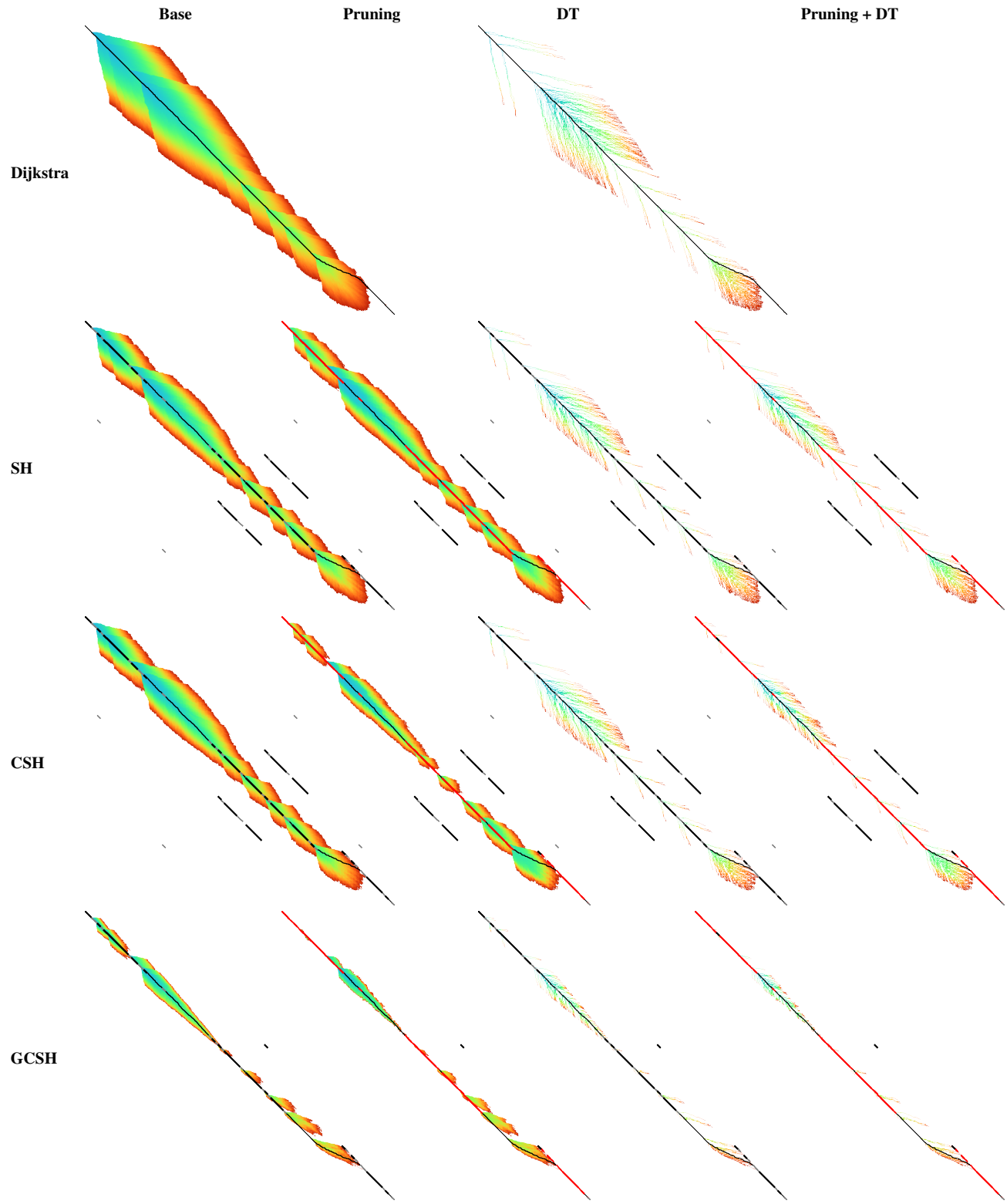


Fig. 15. Expanded states for various heuristics and techniques, on a sequence containing a noisy region, a repeat, and an indel (n=1000, d=17.5%). The colour shows the order of expanding, from blue to red. The sequences include a highly divergent region, a repeat, and a gap. Matches are shown as **black diagonals**, with inexact matches in grey and pruned matches in red. The final path is **black**. Dijkstra does not have pruning variants, and Dijkstra with DT is equivalent to WFA. More accurate heuristics reduce the number of expanded states by more effectively punishing repeats (CSH) and gaps (GCSH). Pruning reduces the number of expanded states before the pruned matches, and diagonal transition reduces the density of expanded states in quadratic regions.

D Notation

Object	Notation
Sequences	
Alphabet	$\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$
Sequences	$A = \overline{a_0 a_1 \dots a_i \dots a_{n-1}} \in \Sigma^*$
	$B = \overline{b_0 b_1 \dots b_j \dots b_{m-1}} \in \Sigma^*$
Substring	$A_{i..i'} = \overline{a_i \dots a_{i'-1}}$
Prefix	$A_{<i} = \overline{a_0 \dots a_{i-1}}$
Suffix	$A_{\geq i} = \overline{a_i \dots a_{n-1}}$
Edit distance	$\mathrm{ed}(A, B)$
Divergence	$d = \mathrm{ed}(A, B)/n$
Error rate	e
Alignment graph	
Graph	$G = (V, E)$
Vertices (states)	$u, v \in V = \{\langle i, j\rangle \mid 0 \leq i \leq n, 0 \leq j \leq m\}$
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	$\mathrm{d}(u, v)$
Path, shortest path	π, π^*
Cost	$\mathrm{c}_{\mathrm{path}}(\pi)$
Diagonal transition	
Farthest-reaching state	$F_{gk} = i{+}j$ on diagonal $k{=}i{-}j$
A*	
Start and target state	$v_s = \langle 0, 0\rangle$, $v_t = \langle n, m\rangle$
Distance from v_s	$g^* = \mathrm{d}(v_s, \cdot)$
Distance to v_t	$h^* = \mathrm{d}(\cdot, v_t)$
Heuristic	h
Best distance from start	g
Estimated distance	$f = g + h$
Admissible heuristic	$h \leq h^*$
Consistent heuristic	$h(u) \leq \mathrm{d}(u, v) + h(v)$
Expanded states	E

Object	Notation
Seeds and matches	
Seed length	k
Seed potential	r
Seeds	$s \in \mathcal{S}$, $s_l = A_{lk..lk+k}$
Seeds in suffix	$\mathcal{S}_{\geq i} = \{s_l \in \mathcal{S} \mid lk \geq i\}$
Alignment of seed	π_s
Matches (per seed)	$m \in \mathcal{M}$, $\mathcal{M}_s$
Terminal match	m_ω from v_t to v_t
Cost of match	$0 \leq \mathrm{c_m}(m) < r$
Score of match	$0 < \mathrm{score}(m) = r - \mathrm{c_m}(m) \leq r$
Score of seed	$\mathrm{score}(s) = \max_{m \in \mathcal{M}_s} \mathrm{score}(m)$
Chains	
Preceding states	$\langle i, j\rangle \preceq \langle i', j'\rangle$ when $i \leq i'$ and $j \leq j'$
Preceding matches	$m \preceq m'$ when $\mathrm{end}(m) \preceq \mathrm{start}(m')$
	$u \preceq m$ when $u \preceq \mathrm{start}(m)$
Partial order	$u \preceq_p v$ when $p(u) \leq p(v)$
i-order	$\langle i, j\rangle \preceq_i \langle i', j'\rangle$ when $i \leq i'$
$\preceq_p$-chain	$m_1 \preceq_p \dots \preceq_p m_l \preceq_p v_t$
Chaining costs	
Chaining cost	$\gamma(m, m')$
Gap cost	$c_{\mathrm{gap}}(\langle i, j\rangle, \langle i', j'\rangle) := \lvert(i'{-}i){-}(j'{-}j)\rvert$
Seed cost	$c_{\mathrm{seed}}(u, v) = r \cdot \mathcal{S}_{u \dots v}$
Gap-seed cost	$c_{\mathrm{gs}} = \max(c_{\mathrm{gap}}, c_{\mathrm{seed}})$
Scores	
Potential	$P\langle i, j\rangle = r \cdot \lvert\mathcal{S}_{\geq i}\rvert$
Chain score	$S_p(m) = \max_{m \preceq_p m_1 \preceq_p \dots \preceq_p v_t} \mathrm{score}(m) + \dots$
	$S_p(u) = \max_{u \preceq_p m \preceq_p v_t} S_p(m)$
Computation	$h_{p, c_{\mathrm{seed}}}(u) = P(u) - S_p(u)$
Heuristics	
SH	$h_{\mathrm{s}}(u) = P(u) - S_i(u)$
CSH	$h_{\mathrm{cs}}(u) = P(u) - S_{\preceq}(u)$
GCSH	$h_{\mathrm{gcs}}(u) = \max(c_{\mathrm{gap}}(u, v_t), P(u) - S_T(u))$
	$T : \langle i, j\rangle \mapsto (i{-}j{-}P\langle i, j\rangle, j{-}i{-}P\langle i, j\rangle)$
Pruning heuristic	$\hat{h}^{\mathcal{M}}$
Layers	
Layer	$\mathcal{L}_\ell = \{u \mid S_p(u) \geq \ell\}$
Dominant state	$u \in \mathcal{L}_\ell$ s.t. $\{v \in \mathcal{L}_\ell \mid u \preceq v\} = \{u\}$