PESHO IVANOV

OPTIMAL SEQUENCE ALIGNMENT USING A*

DISS. ETH NO. ?

OPTIMAL SEQUENCE ALIGNMENT USING A*

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

PESHO IVANOV
Dipl., Eidgenössisches Polytechnikum

born on 29 May 1989
citizen of Bulgaria

accepted on the recommendation of

Prof. Dr. Martin Vechev, examiner
Prof. Dr. Gunnar Rätsch, co-examiner
Prof. Dr. Veli Mäkinen, co-examiner
Prof. Dr. Paul Medvedev, co-examiner

2022

DOI: 10.3929/ethz-a-

With gratitude to my high-school physics teacher Svilen Rusev.
Thank you for opening the world of science to me in an intuitive and visual way.

ABSTRACT

Sequence alignment is the process of detecting similarities between biological sequences, such as DNA, RNA and proteins. For the last half a century, sequence alignment has been of central importance for molecular biology. Applications include evolutionary biology, genome assembly, read mapping, variation detection and computational methods are crucial for the correctness of the analyses of the vast amounts of biological data. Can we use the the A* shortest path algorithm to find optimal alignments fast?

This thesis explores two variations of the alignment problem: *reads mapping* and *pairwise alignment*, also known as semi-global and global alignment. Biological sequences do not generally align perfectly due to biological differences and technical errors. Given two sequences, the desired alignment is a position-to-position correspondence between two sequences which minimizes the edit costs (substitutions, insertions or deletions). This task is closely related to calculating *edit distance*. Practical alignment algorithms are desired to ① find accurate alignments, ② apply to a wide range of data, and ③ use little time and memory.

Existing aligning algorithms are either optimal but quadratic or fast but appximate. Even though the resulting alignment is linear. This gap has motivated the development of various faster but approximate algorithms. Moreover, theoretical results suggest that it is that in general, alignment is not solvable in strongly subquadratic time.

We consider a principled alignment formulation based on shortest paths and demonstrate that the A* shortest path algorithm can be used to outperform current methods. Unlike existing methods, A* enables an *informed search* based on information from the unaligned sequence suffixes, thus radically improving the empyrical runtime scaling (up to linear) in the average case while providing optimality guarantees. On real data, this approach reaches orders of magnitude of speedup compared to existing approaches.

An optimal alignment can naturally be represented as a shortest path in an alignment graph (equivalent to the DP table). In order to find such a shortest path with minimal exploration, we instantiate the A* algorithm with a novel problem-specific heuristic function based on the unaligned parts of the sequences. This additional information is a problem-specific heuristic function and it heavily determines the efficiency of the search. For any explored state by A*, this heuristic function should compute a lower

bound on the remaining path length, or more specifically, the minimal cost of edit operations needed to align the remaining sequences.

Seed heuristic. In practice, while achieving polynomial speed ups on real data. It ① provides optimality guarantees according to edit distance, ② scales to long and noisy sequences, and ③ scales subquadratically with sequence length. To scale to large reference sequences, we extend the graph with a trie index. To scale to long queries, we introduce design an admissible *seed heuristic*, which is provably-optimal also efficient to compute. To scale to high error rates, we design

Many tools do not have a well-stated problem they optimize.

Probabilistic approach. Focus on the metric. Extend to MSA, local, affine. Prorotype implementations No asymptotics.

ZUSAMMENFASSUNG

Deutsche Zusammenfassung hier.

ACKNOWLEDGEMENTS

I would like to thank . . .

CONTENTS

NOTATION

FREQUENTLY USED SYMBOLS

E	energy
m	rest mass
p	impulse

PHYSICAL CONSTANTS

c	speed of light in vacuum, $c = 299\,792\,458\,\mathrm{m\,s^{-1}}$

(CODATA 2014 [**codata**])

1 INTRODUCTION

It is better to be wrong than to be vague.
— Freeman Dyson

The number of possible alignments grow exponentially with length. The usual underlying question to finding "correct" alignments. Regarding the precision of alignment, one is usually interested in base-to-base (aka letter-to-letter) correspondence between the sequences, even though for some applications a less detailed solution is sufficient: only the similarity between sequences or the location where a read maps to a reference. Exact alignment is only useful for very short sequences (often kmers), and for all other cases the optimized metric may be hamming distance, edit distance (unit costs), Levenshtein distance, affine costs, convex and concave costs, general costs and others.

Depending on the the number of aligned sequences, there is pairwise alignment and multiple sequence alignment (MSA). Depending on the parts of the sequences that are aligned to each other, we differentiate global, local and various semi-global alignemnts. There are generalizations to sequence-to-sequence alignment, including aligning to nonlinear structures, such as directed acyclic graphs, DAGs, general graphs and others. These structures are nowadays becoming more common as a compressed form of representing a set of references to which a sequence can be aligned. Often, one best alignment is sufficient but finding several best (top-K) alignments. In the context of read mapping, a set of reads is aligned to the same reference sequence so an indexing procedure is often useful for the performance.

We specifically consider the mapping of a set of reads to a general graph, and the global pairwise alignment.

Existing optimal algorithms are based on dynamic programming (DP) and run in quadratic time (assuming that the number of errors is proportional to the length)

we employ the A* algorithm which is an *informed search* algorithm. TODO: a case for the informed algorithms

2 SEED HEURISTIC FOR GLOBAL ALIGNMENT

MOTIVATION Sequence alignment has been a core problem in computational biology for the last half-century. It is an open problem whether exact pairwise alignment is possible in linear time for related sequences [1].

METHODS We solve exact global pairwise alignment with respect to edit distance by using the A* shortest path algorithm on the edit graph. In order to efficiently align long sequences with high error rate, we extend the *seed heuristic* for A* [2] with *match chaining*, *inexact matches*, and the novel *match pruning* optimization. We prove the correctness of our algorithm and provide an efficient implementation in A*PA.

RESULTS We evaluate A*PA on synthetic data (random sequences of length n with uniform mutations with error rate e) and on real long ONT reads of human data. On the synthetic data with e=5% and $n \leq 10^7$ *bp*, A*PA exhibits a near-linear empirical runtime scaling of $n^{1.08}$ and achieves $>250\times$ speedup compared to the leading exact aligners EDLIB and BIWFA. Even for a high error rate of e=15%, the empirical scaling is $n^{1.28}$ for $n \leq 10^7$ *bp*. On two real datasets, A*PA is the fastest aligner for 58% of the alignments when the reads contain only sequencing errors, and for 17% of the alignments when the reads also include biological variation.

AVAILABILITY github.com/RagnarGrootKoerkamp/astar-pairwise-aligner

CONTACT ragnar.grootkoerkamp@inf.ethz.ch, pesho@inf.ethz.ch

2.1 INTRODUCTION

The problem of aligning one biological sequence to another has been formulated over half a century ago [3] and is known as *global pairwise alignment* [4]. Pairwise alignment has numerous applications in computational biology, such as genome assembly, read mapping, variant detection, multiple sequence alignment, and differential expression [5]. Despite the centrality and age of pairwise alignment, "a major open problem is to implement an algorithm with linear-like empirical scaling on inputs where the edit distance is linear in n" [1].

Alignment accuracy affects the subsequent analyses, so a common goal is to find a shortest sequence of edit operations (insertions, deletions, and substitutions of single letters) that transforms one sequence into the other. Finding such a sequence of operations is at least as hard as computing the *edit distance*, which has recently been proven to not be computable in strongly subquadratic time, unless SETH is false [6]. Given that the number of sequencing errors is proportional to the length, existing exact aligners are limited by quadratic scaling not only in the worst case but also in practice. This is a computational bottleneck given the growing amounts of biological data and the increasing sequence lengths [7].

2.1.1 *Related work on alignment*

We outline algorithms for exact pairwise alignment and their fastest implementations for biological sequences. Refer to Kucherov [7] for approximate, probabilistic, and non-edit distance algorithms and aligners.

DYNAMIC PROGRAMMING The standard approach to sequence alignment is by successively aligning prefixes of the first sequence to prefixes of the second. Vintsyuk [8] was the first to introduce this $O(nm)$ dynamic programming (DP) approach for a comparing a pair speech signals with n and m elements. Independently, it was applied to compute edit distance for biological sequences [3, 9–11]. This well-known algorithm is implemented in modern aligners like SeqAn [12] and Parasail [13]. Improving the performance of these quadratic algorithms has been a central goal in later works.

BANDING AND BIT-PARALLELIZATION When similar sequences are being aligned, the whole DP table may not need to be computed. One

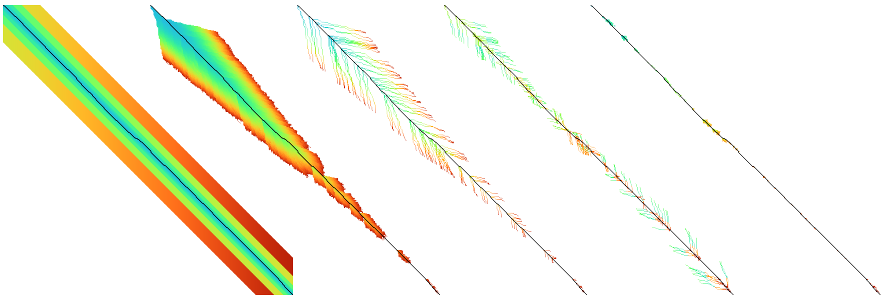


FIGURE 2.1: Demonstration of the computed states by various optimal alignment algorithms and corresponding aligners that implement them on synthetic data (length n=500 bp, error rate e=20%). Blue-to-red coloring indicates the order of computation. a Exponential banding algorithm (EDLIB), b Dijkstra, c Diagonal transition/DT (WFA), d Diagonal transition with divide-and-conquer/D&C (BIWFA), e A* with chaining seed heuristic and match pruning (seed length k=5 and exact matches).

such output-sensitive algorithm is the *banded* algorithm of Ukkonen [14] (Fig. 2.1a) which considers only states near the diagonal within an exponentially increasing *band*, and runs in $O(ns)$ time, where s is the edit distance between the sequences. This algorithm, combined with the *bit-parallel optimization* by Myers [15] is implemented by the EDLIB aligner [16] that runs in $O(ns/w)$ runtime, where w is the machine word size (nowadays 32 or 64).

DIAGONAL TRANSITION AND WFA The $O(ns)$ runtime complexity can be improved using the algorithm independently discovered by Ukkonen [14] and Myers [17] that is known as *diagonal transition* [4] (Fig. 2.1c). It has an $O(ns)$ runtime in the worst-case but only takes expected $O(n + s^2)$ time under the assumption that the input sequences are random [17]. This algorithm has been extended to linear and affine costs in the *wavefront alignment* (WFA) algorithm [18] in a way similar to Gotoh [19], and has been improved to only require linear memory in BIWFA [20] by combining it with the *divide and conquer* approach of Hirschberg [21], similar to Myers [17] algorithm for unit edit costs (Fig. 2.1d). Note that when each sequence letter has an error with a constant probability, the total number of errors s is proportional to n, so that even these algorithms have a quadratic runtime.

SHORTEST PATHS AND A* A pairwise alignment that minimizes edit distance corresponds to a shortest path in the *edit graph* [8, 14]. Assuming non-negative edit costs, a shortest path can be found using Dijkstra's algorithm [14] (Fig. 2.1b) or A* [22]. A* is an informed search algorithm which uses a task-specific heuristic function to direct its search. Depending on the heuristic function, a shortest path may be found significantly faster than by an uninformed search such as Dijkstra's algorithm. In the context of semi-global sequence-to-graph alignment, A* has been used to empirically scale sublinearly with the reference size for short reads [23].

SEEDS AND MATCHES Seed-and-extend is a commonly used paradigm for solving semi-global alignment approximately [7]. Seeds are also used to define and compute LCSk [24], a generalization of longest common subsequence (LCS). In contrast, the *seed heuristic* by Ivanov, Bichsel & Vechev [2] speeds up finding an optimal alignment by using seed matches to speed up the A* search. The seed heuristic enables empirical near-linear scaling to long HiFi reads (up to $30\,kbp$ with 0.3% errors) [2]. A limitation of the existing seed heuristic is the low tolerance to increasing error rates due to using only long exact matches without accounting for their order.

2.1.2 *Contributions*

Our algorithm exactly solves global pairwise alignment for edit distance costs, also known as *Levenshtein distance* [25]. It uses the A* algorithm to find a shortest path in the edit graph.

SEED HEURISTIC In order to handle higher error rates, we extend the *seed heuristic* [2] to *inexact matches*, allowing up to 1 error in each match. To handle cases with a large number of seed matches we introduce *match chaining*, constraining the order in which seed matches can be linked [26, 27]. We prove that our *chaining seed heuristic* with inexact matches is admissible, which guarantees that A* finds a shortest path.

MATCH PRUNING In order to reduce the number of states expanded by A* we apply the *multiple-path pruning* observation of Poole & Mackworth [28]: once a shortest path to a vertex has been found, no other paths to this vertex can improve the global shortest path. We prove that when a state at the start of a match is expanded, a shortest path to this state has been found. Since no other path to this state can be shorter, we show that we can *prune* (remove) the match, thus improving the seed heuristic. This incremental heuristic search has some similarities to Real-time Adaptive A* [29].

IMPLEMENTATION We efficiently implement our algorithm in the A*PA aligner. In particular, we use *contours* [30–32] to efficiently to compute the chaining seed heuristic, and update them when pruning matches.

SCALING AND PERFORMANCE We compare the scaling and performance of our algorithm to other exact aligners on synthetic data, consisting of random genetic sequences with up to 15% uniform errors and up to 10^7 bases. We demonstrate that inexact matches and match chaining enable scaling to higher error rates, while match pruning enables near-linear scaling with length by reducing the number of expanded states to not much more than the best path (Fig. 2.1e). Our empirical results show that for e=5% and n=10^7 *bp*, A*PA outperforms the leading aligners EDLIB [16] and BIWFA [20] by more than 250 times.

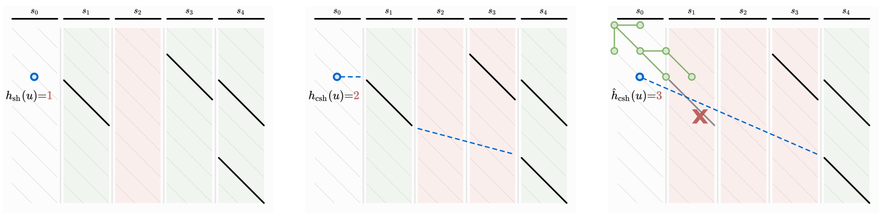
We demonstrate a limited applicability of our algorithm to long Oxford Nanopore (ONT) reads from human samples. A*PA is the fastest exact aligner on 58% of the alignments on a dataset with only sequencing errors, and on 17% of the alignments on a dataset with biological variation.

2.2 PRELIMINARIES

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

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

EDIT GRAPH Let *state* $\langle i, j\rangle$ denote the subtask of aligning the prefix $A_{<i}$ to the prefix $B_{<j}$. The *edit graph* (also called *alignment graph*) $G(V, E)$ is a weighted directed graph with vertices $V = \{\langle i, j\rangle | 0 \leq i \leq n, 0 \leq j \leq m\}$ corresponding to all states, and edges connecting tasks to subtasks: edge



(a) Seed heuristic

(b) Chaining seed heuristic

(c) Chaining seed heuristic + match pruning

FIGURE 2.2: A demonstration of the seed heuristic, chaining seed heuristic, and match pruning. Sequence A is split into 5 seeds drawn as horizontal black segments (—) on top. Their exact matches in B are drawn as diagonal black segments (⟍). The heuristic is evaluated at the blue state (u, ○), based on the 4 remaining seeds. The dashed blue paths show maximal length chains of seed matches (green columns, 🟩). The seeds that are not matched (red columns, 🟥) count towards the heuristic. a The seed heuristic $h_{sh}(u) = 1$ is the number of remaining seeds that do not have matches (s_2). b The chaining seed heuristic $h_{csh}(u) = 2$ is the number of not matched remaining seeds (s_2 and s_3) for a path going only down and to the right containing a maximal number of matches. c Once the start of a match is expanded (shown as green circles, ○), the match is *pruned* (marked with red cross, ×), and future computations of the heuristic ignore it. This reduces the maximum chain of matches starting at u by 1 (s_1) so that $\hat{h}_{csh}(u)$ increases by 1.

$\langle i,j\rangle \to \langle i{+}1,j{+}1\rangle$ has cost 0 if $a_i = b_j$ (match) and 1 otherwise (substitution), and edges $\langle i,j\rangle \to \langle i{+}1,j\rangle$ (deletion) and $\langle i,j\rangle \to \langle i,j{+}1\rangle$ (insertion) have cost 1. We denote the root state $\langle 0,0\rangle$ by v_s and the target state $\langle n,m\rangle$ by v_t. For brevity we write $f(\langle i,j\rangle)$ as $f\langle i,j\rangle$. The edit graph is a natural representation of the alignment problem that provides a base for all alignment algorithms.

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

DIJKSTRA AND A* Dijkstra's algorithm [33] finds a shortest path from v_s to v_t by *expanding* vertices in order of increasing distance $g^*(u)$ from the start. The A* algorithm [34, 35], instead, directs the search towards a target by expanding vertices in order of increasing $f(u) := g(u) + h(u)$, where $h(u)$ is a heuristic function that estimates the distance $h^*(u)$ to the end and $g(u)$ is the shortest length of a path from v_s to u found so far. A heuristic is *admissible* if it is a lower bound on the remaining distance, $h(u) \leq h^*(u)$, which guarantees that A* has found a shortest path as soon as it expands v_t. Heuristic h_1 *dominates* another heuristic h_2 when $h_1(u) \geq h_2(u)$ for all vertices u. A dominant heuristic will usually, but not always [36], 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 §3.9.2.

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

CONTOURS To efficiently calculate maximal chains of matches, *contours* are used. Given a set of matches $\mathcal{M}$, $S(u)$ is the number of matches in the longest chain $u \preceq m_0 \preceq \dots$, 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 $S(u) < \ell$ (see Fig. 2.3). Note that contour ℓ is completely determined by the set of matches $m \in \mathcal{M}$ for which $S(\text{start}(m)) = \ell$ [30]. Hunt & Szymanski [31] give an algorithm to efficiently compute S when $\mathcal{M}$ is the set of single-letter matches between A and B, and Deorowicz & Grabowski [37] give an algorithm when $\mathcal{M}$ is the set of k-mer exact matches.

2.3 METHODS

In this section we define the *seed heuristic* with *inexact matches* and *chaining*, and *match pruning* (see Fig. 2.2). We motivate each of these extensions intuitively, define them formally, and prove that A* is guaranteed to find a shortest path. We end with a reformulation of our heuristic definitions that allows for more efficient computation. §3.14.4 contains a summary of the notation we use.

2.3.1 *Overview*

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

To define the seed heuristic h_{sh}, we split A into short, non-overlapping substrings (*seeds*) of fixed length k. Since the whole of sequence A has to be aligned, each of the seeds also has to be aligned somewhere. If a seed cannot be *exactly* matched in B without mistakes, then at least one edit has to be made to align it. We first compute all positions in B where each seed from A matches exactly. Then, a lower bound on the edit distance between the remaining suffixes $A_{\geq i}$ and $B_{\geq j}$ is given by the number of seeds entirely contained in $A_{\geq i}$ that do not match anywhere in B. An example is shown in Fig. 2.2a.

We improve the seed heuristic by enforcing that the seed matches occur in the same order as their seeds occur in A, i.e., they form a *chain*. Now, the number of upcoming errors is at least the minimal number of remaining seeds that cannot be aligned on a single path to the target. The chaining seed heuristic h_{csh} equals the number of remaining seeds minus the maximum length of a chain of matches, see Fig. 2.2b.

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

Finally, we use *match pruning*: once we find a shortest path to a state v, no shorter path to that state is possible. Since we are only looking for a single shortest path, we may ignore any other path going to v. In particular, when we are evaluating the heuristic h in some state u preceding v, and v is at the start of a match, all paths containing the match are excluded, and thus we may simply ignore (*prune*) this match for future computations of the heuristic. This increases the value of the heuristic, as shown in Fig. 2.2c, and leads to a significant reduction of the number of states being expanded by the A*.

2.3.2 *Seed heuristic and chaining seed heuristic*

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

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

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

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

Next, we define two heuristic functions:

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

$$h_{\mathrm{sh}}^{\mathcal{M}}(u) := \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \min_{m \in \mathcal{M}_s} \mathrm{cost}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ r & \text{otherwise.} \end{cases} \tag{2.1}$$

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

$$h^{\mathcal{M}}_{\text{csh}}(u) := \min_{\pi\in G(u\rightsquigarrow v_t)} \sum_{s\in\mathcal{S}_{\geq i}} \begin{cases} \text{cost}(\pi_s) & \textit{if } \pi_s \in \mathcal{M}_s, \\ r & \textit{otherwise.} \end{cases} \tag{2.2}$$

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

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

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

The proof follows directly from the definitions and can be found in §2.8.1. As a consequence of the proof, $h_{\text{csh}}(u)$ dominates $h_{\text{sh}}(u)$, that is, $h_{\text{csh}}(u) \geq h_{\text{sh}}(u)$ for all u. Hence $h_{\text{csh}}(u)$ usually expands fewer states, at the cost of being more complex to compute.

2.3.3 *Seed heuristic as a maximization problem*

In order to efficiently evaluate the seed heuristic, Eq. (2.1), we rewrite it from a minimization of match costs to a maximization of match scores. We first define a few new concepts, and then rewrite the equation into form that will be simpler to compute.

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

$$\text{score}(s) := \begin{cases} \max_{m\in\mathcal{M}_s} \text{score}(m) & \text{if } \mathcal{M}_s \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

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

$$S_{\text{sh}}\langle i,j\rangle := \sum_{s\in\mathcal{S}_{\geq i}} \text{score}(s). \tag{2.4}$$

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

$$P\langle i,j\rangle := r\cdot|\mathcal{S}_{\geq i}|. \tag{2.5}$$

OBJECTIVE Eq. (2.1) can now be rewritten in terms of potential and score,

$$\begin{aligned} h^{\mathcal{M}}_{\text{sh}}(u) &\overset{(2.1)}{=} \sum_{s\in\mathcal{S}_{\geq i}} \left(r - \begin{cases} \max_{m\in\mathcal{M}_s} r-\text{cost}(m) & \text{if } \mathcal{M}_s\neq\emptyset,\\ 0 & \text{otherwise}\end{cases}\right)\\ &= r\cdot|\mathcal{S}_{\geq i}| - \sum_{s\in\mathcal{S}_{\geq i}} \begin{cases} \max_{m\in\mathcal{M}_s} \text{score}(m) & \text{if } \mathcal{M}_s\neq\emptyset,\\ 0 & \text{otherwise}\end{cases}\\ &\overset{(2.3)}{=} r\cdot|\mathcal{S}_{\geq i}| - \sum_{s\in\mathcal{S}_{\geq i}} \text{score}(s) \overset{(2.4),(2.5)}{=} P(u) - S_{\text{sh}}(u). \end{aligned} \tag{2.6}$$

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

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

2.3.4 *Chaining seed heuristic as a maximization problem*

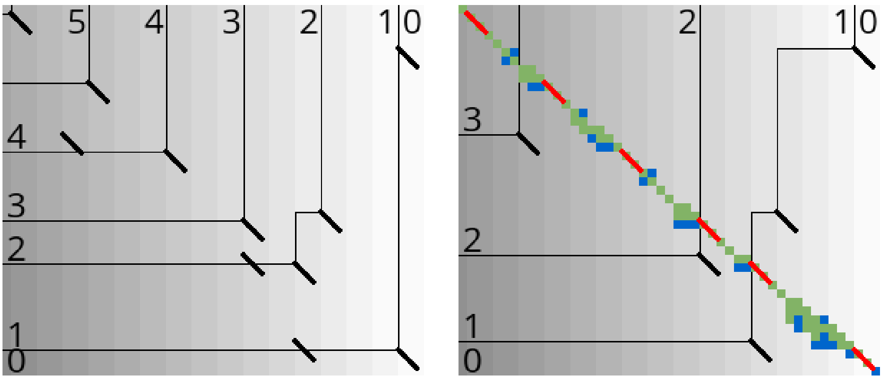
Similar to the seed heuristic, we rewrite Eq. (2.2) into a maximization form that will be simpler to compute.

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

$$S_{\text{chain}}(m) := \text{score}(m) + \max_{m\preceq m'} S_{\text{chain}}(m'). \tag{2.7}$$

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

$$S_{\text{csh}}(u) := \max_{u\preceq m} S_{\text{chain}}(m). \tag{2.8}$$



(a) Before (b) After

FIGURE 2.3: An example of the layers and contours used by the chaining seed heuristic before and after the A⋆ execution with match pruning for r=1 and seed length k=3. Exact matches with score(m)=1 are shown as black diagonal segments (╲). *Contours* are shown as horizontal and vertical black lines and indicate the bottom-right boundaries of *layers* $\mathcal{L}_\ell$ consisting of the states above/left of the contour marked with ℓ. States u between two contours have the same maximal number of matches $S_{\text{csh}}(u)$ on a chain to the end. The value of the heuristic is shown in the background, from white (h=0) to darker grey. Expanded states are shown in green (🟩), open states in blue (🟦), and pruned matches in red (╲). Note that pruning matches during the A⋆ execution shifts the contours and changes the layers.

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

$$\begin{aligned}
h^{\mathcal{M}}_{\text{csh}}(u) &\overset{(2.2)}{=} \min_{\pi \in G(u \rightsquigarrow v_t)} \sum_{s \in \mathcal{S}_{\geq i}} \left(r - \begin{cases} \text{score}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ 0 & \text{otherwise} \end{cases} \right) \\
&= r \cdot |\mathcal{S}_{\geq i}| - \max_{\pi \in G(u \rightsquigarrow v_t)} \sum_{s \in \mathcal{S}_{\geq i}} \begin{cases} \text{score}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ 0 & \text{otherwise} \end{cases} \\
&= r \cdot |\mathcal{S}_{\geq i}| - \max_{u \preceq m_1 \preceq \cdots \preceq m_l} \left\{ \text{score}(m_1) + \cdots + \text{score}(m_l) \right\} \\
&\overset{(2.5),(2.8)}{=} P(u) - S_{\text{csh}}(u).
\end{aligned} \tag{2.9}$$

CONTOURS Like fore the seed heuristic, we define layer $\mathcal{L}_\ell$ as those states u with score $S_{\text{csh}}(u)$ *at least* ℓ. The ℓth *contour* is the bottom-right boundary of $\mathcal{L}_\ell$ (see Fig. 2.3). A state $u \in \mathcal{L}_\ell$ is *dominant* when there is no state $v \in \mathcal{L}_\ell$ with $v \neq u$ and $u \preceq v$. Both $\mathcal{L}_\ell$ and the corresponding contour are completely determined by the matches starting in the dominant states. The following lemma makes this precise:

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

$$\mathcal{L}_\ell = \{u \mid \exists m \in \mathcal{M} : u \preceq m \text{ and } \ell \leq S_{\text{chain}}(m) < \ell + r\}. \tag{2.10}$$

Proof. It follows directly from Eq. (2.8) that $\mathcal{L}_\ell = \{u \mid \exists m \in \mathcal{M} : u \preceq m \text{ and } S_{\text{chain}}(m) \geq \ell\}$. Suppose that m has score $S_{\text{chain}}(m) \geq \ell + r$. By definition, $\text{score}(m) \leq r$, so by Eq. (2.7) there must be a match $m \preceq m'$ with $S_{\text{chain}}(m') = S_{\text{chain}}(m) - \text{score}(m) \geq (\ell + r) - r = \ell$. This implies that $u \preceq m \preceq m'$, which allows m, and hence any match with $S_{\text{chain}}(m) \geq \ell + r$, to be omitted from consideration. □ □ □

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

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

2.3.5 *Match pruning*

In order to reduce the number of states expanded by the A* algorithm, we apply the *multiple-path pruning* observation: once a shortest path to a vertex has been found, no other path to this vertex could possibly improve the global shortest path [28]. When A* expands the start of a match we *prune* this match, so that the heuristic values of preceding states do no longer benefit from it, thus getting deprioritized by the A*. We define *pruned* variants of the seed heuristic and the chaining seed heuristic that ignore pruned matches:

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

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

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

The proof can be found in §2.8.2. Since the heuristic may increase over time, our algorithm ensures that f is up-to-date when expanding a state by *reordering* nodes with outdated f values (see Remark 1 in §2.8.2).

2.4 ALGORITHM

Our algorithm computes the shortest path in the edit graph using a variant of A* that handles pruning (§3.9.2). This depends on the efficient computation of the heuristics (§2.4.2 and §2.4.3). §2.4.4 contains further implementation notes.

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

2.4.1 *A* algorithm*

We give our variant of the A* algorithm [34] that adds steps A4 and A6a (marked as **bold**) to handle the pruning of matches. All computed values of g are stored in a map, and all states in the front are stored in a priority queue of tuples $(v, g(v), f(v))$ ordered by increasing f.

A1. Set $g(v_s) = 0$ in the map and initialize the priority queue with $(v_s, g(v_s){=}0, f(v_s))$.

A2. Pop the tuple (u, g, f) with minimal f.

A3. If $g > g(u)$, i.e. the value of $g(u)$ in the map has decreased since pushing the tuple, go to step 2. This is impossible for a consistent heuristic.

A4. If $f \neq f(u)$, *reorder u*: push $(u, g(u), f(u))$ and go to step 2. This can only happen when f changes value, i.e. after pruning.

A5. If $u = v_t$, terminate and report a best path.

A6. Otherwise, *expand* u:
   a. Prune matches starting at u.

   b) For each successor v of u, *open* v when either $g(v)$ has not yet been computed or when $g(u) + d(u, v) < g(v)$. In this case, update the value of $g(v)$ in the map and insert $(v, g(v), f(v))$ in the priority queue. Go to step 2. This makes u the *parent* of v.

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

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

2.4.2 *Computing the seed heuristic*

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

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

PRECOMPUTATION

R1. Compute the set of seeds $\mathcal{S}$ by splitting A into consecutive kmers.

R2. Build a hashmap containing all kmers of B.

R3. For each seed, find all matches via a lookup in the hashmap. In case of inexact matches ($r = 2$), all sequences at distance 1 from each seed are looked up independently. The matches $\mathcal{M}$ are stored in a hashmap keyed by the start of each match.

R4. Initialize the array of potentials P by iterating over the seeds backwards.

R5. Initialize the array of layer starts LS by setting $LS[0] = n$ and iterating over the seeds backwards. For each seed $s = A_{i\ldots i'}$ that has matches, push score(s) copies of i to the end of LS.

EVALUATING THE HEURISTIC IN $u = \langle i, j\rangle$

E1. Look up the potential $P[i]$.

E2. $S_{\text{sh}}(u) = \max\{\ell : LS[\ell] \geq i\}$ is found by a binary search over ℓ.

E3. Return $h(u) = P[i] - S_{\text{sh}}(u)$.

PRUNING WHEN EXPANDING MATCH START $u = \langle i, j \rangle$

P1. Compute $seedscore_{old} := \text{score}(s)$ and $\ell_{old} := S_{\text{sh}}(u)$.

P2. Remove all matches from $\mathcal{M}$ that start at u.

P3. Compute $seedscore_{new} := \text{score}(s)$ and set $\ell_{new} := \ell_{old} - seedscore_{old} + seedscore_{new}$. If $\ell_{new} < \ell_{old}$, remove layers $LS[\ell_{new} + 1]$ to $LS[\ell_{old}]$ from LS and shift down larger elements $LS[\ell]$ with $\ell > \ell_{old}$ correspondingly.

2.4.3 *Computing the chaining seed heuristic*

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

Our algorithm for computing the chaining seed heuristic is similar to the one for the seed heuristic (§2.4.2). Hence we highlight only the steps which differ (e.g. step 2′ is used instead of 2 for the seed heuristic).

PRECOMPUTATION

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

EVALUATING THE HEURISTIC IN $u = \langle i, j \rangle$

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

PRUNING WHEN EXPANDING MATCH START $u = \langle i, j \rangle$

P2′. Compute $\ell_u = S_{\text{csh}}(u)$, and remove all matches that start at u from layers $LM[\ell_u - r + 1]$ to $LM[\ell_u]$.

P3′. Iterate over increasing ℓ starting at $\ell = \ell_u + 1$ and recompute $\ell' := S_{\text{chain}}(m) \leq \ell$ for all matches m in $LM[\ell]$. Move m from $LM[\ell]$ to layer $LM[\ell']$ whenever $\ell' \neq \ell$. Stop when either r consecutive layers are unchanged, in which case no further changes of $S_{\text{chain}}(m)$ can happen because of Eq. (2.7) and $\text{score}(m) \leq r$, or when all matches in r consecutive layers have shifted down by the same amount, say $\Delta := \ell - \ell'$. In the latter case, $S_{\text{chain}}(m)$ decreases by Δ for all matches with score at least ℓ. We remove the emptied layers $LM[\ell - \Delta + 1]$ to $LM[\ell]$ so that all higher layers shift down by Δ.

2.4.4 *Implementation*

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

BUCKET QUEUE We use a hashmap to store all computed values of g in the A* algorithm. Since the edit costs are bounded integers, we implement the priority queue using a *bucket queue* [38]. Unlike heaps, this data structure has amortized constant time push and pop operations since the difference between the value of consecutive pop operations is bounded.

GREEDY MATCHING OF LETTERS From a state $\langle i, j \rangle$ where $a_i = b_j$, it is sufficient to only consider the matching edge to $\langle i+1, j+1 \rangle$ [23, 39], and ignore the insertion and deletion edges to $\langle i, j+1 \rangle$ and $\langle i+1, j \rangle$. During alignment, we greedily match as many letters as possible within the current seed before inserting only the last opened state in the priority queue. We do not cross seed boundaries in order to not interfere with match pruning. We include greedily matched states in the reported number of expanded states.

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

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

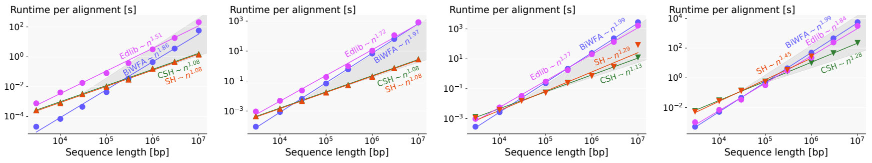
BINARY SEARCH HINTS When we open v from its parent u in step A6b of the A* algorithm, the value of $h(v)$ is close to the value of $h(u)$. In particular, $S_{\text{sh}}(v)$ (resp. $S_{\text{csh}}(v)$) is close to and at most the score in u. Instead of a binary search (step E2), we do a linear search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

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

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

2.5 RESULTS

We implemented the algorithms from §2.2 in the aligner A*PA and refer to the two of our algorithms as SH for seed heuristic, and CSH for chaining seed heuristic. Here we empirically compare the runtime and memory usage to the exact optimal aligners EDLIB and BIWFA on both synthetic and real data. We demonstrate the benefit of match pruning on the runtime scaling with sequence length, and the benefits from match chaining and inexact matches on scaling to high error rates. All code, evaluation scripts and data are available in the A*PA repository.



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

FIGURE 2.4: Log-log plots of the runtime for aligning **synthetic sequences** of increasing length for A*PA seed heuristic (SH), A*PA chaining seed heuristic (CSH), EDLIB (●) and BIWFA (●). The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. For $e \leq 5\%$, SH (▼) and CSH (▼) use k=15, r=1. For $e \geq 10\%$, SH (▲) and CSH (▲) use k=15, r=2. The missing data points for SH at e=15% are due to exceeding the memory limit (30 GB). Each runtime is the average over $\lfloor 10^7/n \rfloor$ alignments.

2.5.1 *Setup*

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

HUMAN DATA We consider two datasets[1] of Oxford Nanopore Technologies (ONT) reads that are aligned to regions of the telomere-to-telomere assembly of a human genome CHM13 (v1.1) [40]. Statistics are shown in Table 2.1.

- *CHM13: Human reads without biological variation.* We randomly sampled 50 reads of length at least 500 *kbp* from the first 12 *GB* of the ultra-Long ONT reads that were used to assemble CHM13. We removed soft clipped

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

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

Table 2.1: Statistics on the **real data**: ONT reads from human samples.

regions and paired each read to its corresponding reference region in CHM13[2].

- *NA12878: Human reads with biological variation.* We consider the long ONT MinION reads from a different reference sample NA12878 [41] which was used to evaluate BiWFA [20]. This dataset contains 48 reads longer than 500 *kbp* that were mapped to CHM13.

COMPARED ALGORITHMS AND ALIGNERS We compare the seed heuristic and chaining seed heuristic, implemented in A*PA, to the state-of-the-art exact pairwise aligners BiWFA and Edlib. In order to study the performance of the A* heuristic functions and the pruning optimization, we also compare to Dijkstra's algorithm (which is equivalent to A* with a zero heuristic) and to a no-pruning variant of A*, all implemented in A*PA. The exact aligners SeqAn and Parasail are not included in this evaluation since they have been outperformed by BiWFA and Edlib [18]. We execute all aligners with unit edit costs and compute not only the edit distance but also an optimal alignment. See §2.8.4 for aligner versions and parameters.

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

MEASUREMENTS The runtime (wall-clock time) and memory usage (resident set size) of each run are measured using `time`. The runtime in all plots and tables refers to the average alignment time per pair of sequences. To

2 https://github.com/RagnarGrootKoerkamp/bam2seq

				Runtime [s]			
	Aligner	**Algorithm**	$e =$	1%	5%	10%	15%
●	EDLIB	Banding, exponential search, bit-parallel		206	839	1653	3073
●	BIWFA	Diagonal transition, divide & conquer		56.9	806	2799	5492
▲▼	A*PA	A*, match-pruning, seed heuristic		**1.41**	**2.79**	82.5	ML
▲▼	A*PA	A*, match-pruning, chaining seed heuristic		1.60	2.98	**12.3**	**220**

TABLE 2.2: Runtime and memory comparison on **synthetic sequences** of length n=10^7 bp for various error rates. ML stands for exceeding the memory limit of 30 GB. Note that we run our A* heuristics with exact matches (▲▲) when e≤5%, and with inexact matches (▼▼) when e≥10%.

reduce startup overhead, we average faster alignments over more pairs of sequences, as specified in the figure captions. Memory usage is measured as the maximum used memory during the processing of the whole input file. To estimate how algorithms scale with sequence length, we calculate best fit polynomials via a linear fit in the log-log domain using the least squares method.

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

2.5.2 *Comparison on synthetic data*

Here we compare the runtime scaling and performance of the exact optimal aligners on synthetic data. §2.8.5 compares the effects of pruning and inexact matches, whereas §2.8.6 compares SH and CSH in terms of the number of expanded states.

SCALING WITH SEQUENCE LENGTH First, we compare the aligners by runtime scaling with sequence length n for various error rates e (Fig. 2.4). As expected from the theoretical analysis, EDLIB and BIWFA scale approximately quadratically regardless of the error rate. Unlike them, SH and CSH scale subquadratically which explains why they are faster for long enough sequences.

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

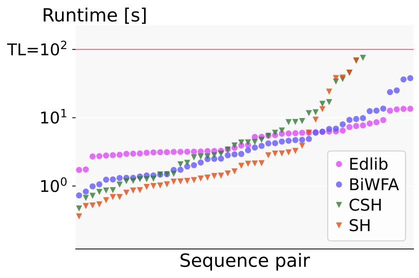
For high error rates ($e \geq 10\%$) and large n, the need to match the seeds inexactly causes a significant number of off-path matches. These off-path matches lower the value of the heuristics and prevent the punishment of suboptimal paths. This causes SH to expand a super-linear number of states (§2.8.6). Chaining the matches enforces them to follow the order of the seeds, which greatly reduces the negative effects of the off-path matches, leading to only linearly many expanded states. Nevertheless, the runtime scaling of CSH is super-linear as a result of the increasing fraction of time (up to 93% for $n=10^7$) spent on reordering states because of outdated f values after pruning.

PERFORMANCE Table 2.2 compares the runtime and memory of A*PA to EDLIB and BIWFA for aligning a single pair of sequences of length $n=10^7$. $A^\star$ with CSH is more than 250 times faster than EDLIB and BIWFA at e=5%. For low error rate ($e \leq 5\%$), there is no significant performance difference between SH and CSH. The memory usage of A*PA for $e \leq 5\%$ is less than 600 *MB* for any n. At e=15%, CSH uses at most 4.7 *GB* while SH goes out of memory ($\geq$ 30 *GB*) because there are too many expanded states to store.

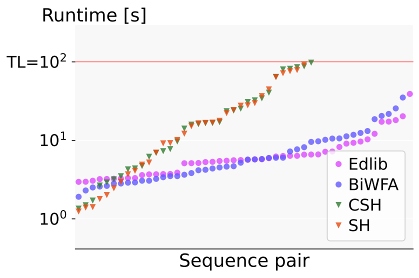
2.5.3 *Comparison on real data*

In this section we compare the exact optimal aligners on long ONT reads from our human datasets (Fig. 2.5). With the presented minimalistic features, A*PA aligns some sequences faster than BIWFA and EDLIB, but the high runtime variance makes it slower overall (§2.8.7).

On the dataset without biological variation CHM13, SH is faster than BIWFA and EDLIB on 58% of the alignments (29 of 50). On the dataset with biological variation NA12878, SH outperforms BIWFA and EDLIB on 17% of the alignments (8 of 48) and in other cases is over an order of magnitude slower. In both datasets, SH and CSH time out for the sequences with



(a) CHM13: ONT read errors



(b) NA12878: ONT read errors + biological mutations

FIGURE 2.5: Log plot comparison of the aligners' runtime on **real data**. The data points for each individual aligner are sorted by alignment time. Alignments that timed out after 100 seconds are not shown.

the highest edit distances, because they have an error rate larger than the heuristic can handle efficiently ($e \geq r/k = 2/15 = 13.3\%$).

CSH usually explores fewer states than SH since chaining seed heuristic dominates the seed heuristic. However, in certain cases CSH is slower than SH since needs more time to update the heuristic after pruning (Step 3′ in §2.4.3).

2.6 DISCUSSION

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

LIMITATIONS Our presented method has several limitations:

1. *Complex regions trigger quadratic search.* Since it is unlikely that edit distance in general can be solved in strongly subquadratic time, it is inevitable that there are inputs for which our algorithm requires quadratic time. In particular, regions with high error rate, long indels, and too many matches (§2.8.8) are challenging and trigger quadratic exploration.

2. *High constant in runtime complexity.* Despite the near-linear scaling of the number of expanded states (§2.8.6), A*PA only outperforms EDLIB and

BiWFA for sufficiently long sequences (Fig. 2.4) due to the relatively high computational constant that the A* search induces.

3. *Complex parameter tuning.* The performance of our algorithm depends heavily on the sequences to be aligned and the corresponding choice of parameters (whether to use chaining, the seed length k, and whether to use inexact matches r). The parameter tuning (currently very simple (§2.5.1) may require a more comprehensive framework when introducing additional optimizations.

4. *Real data.* The efficiency of the presented algorithm has high variability on real data (§2.5.3) due to high error rates, long indels, and multiple repeats (demonstrated in Fig. 2.8). Further optimizations are needed to align complex data.

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

1. *Performance.* The practical performance of our A* approach could be improved using multiple existing ideas from the alignment domain: diagonal transition method, variable seed lengths, overlapping seeds, combining heuristics with different seed lengths, gap costs between matches in a chain [14, 26], more aggressive pruning, and better parameter tuning. More efficient implementations may be possible by using computational domains [22], bit-parallelization [15], and SIMD [18].

2. *Generalizations.* Our method can be generalized to more expressive cost models (non-unit costs, affine costs) and different alignment types (semi-global, ends-free, and possibly local alignment).

3. *Relaxations.* Abandoning the optimality guarantee enables various performance optimizations. Another relaxation of our algorithm would be to validate the optimality of a given alignment more efficiently than finding an optimal alignment from scratch.

4. *Analysis.* The near-linear scaling behaviour requires a thorough theoretical analysis [42]. The fundamental question that remains to be answered is: *What sequences and what errors can be tolerated while still scaling near-linearly with the sequence length?* We expect both theoretical and practical contributions to this question.

2.7 CONCLUSION

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

ACKNOWLEDGEMENTS

The authors are grateful to Mykola Akulov for his help with producing Figs. 2.1 and 2.3, and to Benjamin Bichsel, Maximilian Mordig and André Kahles for valuable feedback on various draft versions. We thank Sergey Nurk for his help with the preparation of the human data. Ragnar Groot Koerkamp is financed by ETH Grant ETH-17 21-1 to Gunnar Rätsch.

2.8 APPENDIX

2.8.1 *The seed heuristic and chaining seed heuristic are admissible*

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

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

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

$$\begin{aligned} h^{\mathcal{M}}_{\text{csh}}(u) = \min_{\pi\in G(u\rightsquigarrow t)} \sum_{s\in\mathcal{S}_{\ge i}} &\begin{cases} \text{cost}(\pi_s) & \text{if } \pi_s \in \mathcal{M}_s, \\ r & \text{otherwise.} \end{cases} \\ \le \min_{\pi\in G(u\rightsquigarrow t)} \sum_{s\in\mathcal{S}_{\ge i}} \text{cost}(\pi_s) \le \min_{\pi\in G(u\rightsquigarrow t)} &\text{cost}(\pi) = h^*(u). \end{aligned} \tag{2.11}$$

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

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

□ □

2.8.2 *Partially admissible heuristic*

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

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

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

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

We follow the structure and notation of Hart, Nilsson & Raphael [34] to prove that A* finds a shortest path when used with a partially admissible heuristic.

Lemma 2 (Generalization of Hart, Nilsson & Raphael [34], Lemma 1)**.** *Assuming v_t is not fixed, there exists an open vertex n' on π^* with $g(n') = g^*(n')$ such that all vertices in π^* following n' are not fixed.*

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

Corollary 1 (Generalization of Hart, Nilsson & Raphael [34], Corollary to Lemma 1)**.** *Suppose that $\hat{h}$ is partially admissible, and suppose A* has not terminated. Then n' as in* **??** *2 has $f(n') \leq g^*(v_t)$.*

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

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

Proof. The proof of Theorem 1 in Hart, Nilsson & Raphael [34] applies, with the remark that we use the specific path π^* and the specific vertex n' from **??** 1, instead of an arbitrary shortest path P. □ □ □

Remark 1 (Monotone heuristic). *We call $\hat{h}(u)$* monotone *when it does not decrease as the A* progresses. To find the vertex u with minimal $f(u)$, we add step A4 to the A* algorithm in §3.9.2 to check whether the value of f in the priority queue is still up to date. If $f = f(u)$, we know that for each vertex (v, g_v, f_v) in the queue we have $f(u) \leq f_v \leq f(v)$ for all other open vertices v in the queue with stored value f_v, ensuring that u indeed has the minimal value of f value of all open vertices.*

2.8.3 *Match pruning preserves finding a shortest path*

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

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

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

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

Part 1: Proof of $g^*(v) + h(v) \leq g^*(w_0) + h(w_0)$. Since w_0 is the first start of a seed at or after v, the set of seeds following v is the same as the set of seeds following u. Thus, the summation in $\hat{h}(v)$ and $\hat{h}(w_0)$ in Eq. (2.1) or Eq. (2.2) is over the same seeds. Since $v \preceq w_0$, this implies $\hat{h}(v) \leq \hat{h}(w_0)$ in both cases, and $g^*(v) \leq g^*(w_0)$ since $v \preceq w_0$.

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

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

□ □

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

Proof. Let π^* be a shortest path from s to v_t. Let n be the last expanded state on π^* at the start of a seed. By **??** 3 n is fixed. By choice of n, no states on π^* following n are expanded, so no matches on π^* following n are pruned. The proof of **??** 3 applies to the path π^* without changes, implying that $h(u) \leq h^*(u)$ for all u on π^* following n, for both $h = \hat{h}_{\text{sh}}$ and $h = \hat{h}_{\text{csh}}$. □ □ □

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

Proof. Follows from **??** 3 in §2.8.2 and **??** 4. □ □

2.8.4 *Aligner versions and parameters*

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

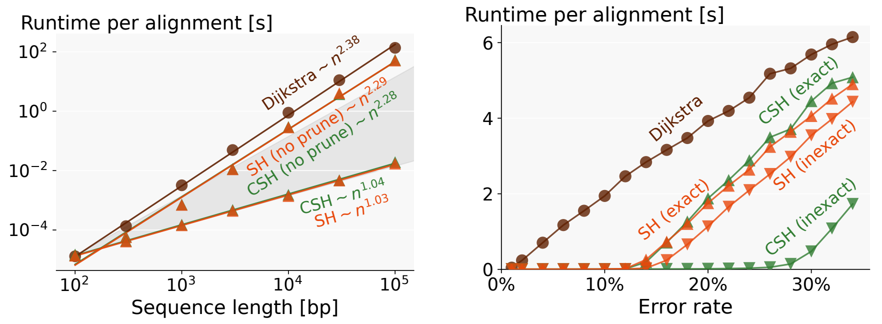
```
astar-pairwise-aligner -i {input} --silent2
  -k {k} -r {r} --algorithm {Dijkstra,SH,CSH}
  [--no-prune]
```

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

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

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

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



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

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

FIGURE 2.6: The effect of optimizations on runtime scaling on **synthetic data**.

2.8.5 *Effect of pruning, chaining, and inexact matches*

The optimizations and generalizations of the seed heuristic (§2.2) impact the performance in a complex way. Here we aim to provide intuitive explanations (Fig. 2.6).

PRUNING ENABLES NEAR-LINEAR SCALING WITH LENGTH Fig. 2.6a shows that match pruning has a crucial effect on the runtime scaling with length for both SH and CSH. Essentially, this optimization changes the quadratic runtime to near-linear runtime. The pruned variants of SH and CSH are averaged over $\lfloor 10^7/n \rfloor$ sequence pairs, while the no-prune variants and Dijkstra are averaged over $\lfloor 10^5/n \rfloor$ pairs.

INEXACT MATCHING AND MATCH CHAINING ENABLE SCALING TO HIGH ERROR RATES Fig. 2.6b shows that inexact matches can tolerate higher error rates. Because of the larger number of matches, chaining is needed to preserve the near-linear runtime. There are two distinctive modes of operation: the runtime is close to constant up to a certain error rate, after which the runtime grows linearly in e. Thus, our heuristics can direct the search up to a certain fraction of errors, after which does a Dijkstra-like exploration step for each additional error. A reasonable quantification of the effect of different optimizations is to mark the error rate at which the heuristic transfers to the second (slow) mode of operation. For n=10^4 and

k=9, Dijkstra starts a linear exploration at e=0%, SH and CSH with exact matches start at around 12%, SH with inexact matches start at around 14%, and CSH with inexact matches start at around 27%.

2.8.6 *Expanded states and equivalent band*

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

TABLE 2.3: Equivalent band for aligning **synthetic sequences** of a given length and error rate. Each cell averages over $\lfloor 10^7/n \rfloor$ alignments. ML stands for exceeding the memory limit of 30 GB.

Table 2.3 shows the *equivalent band* of each alignment: the number of expanded states divided by n. An equivalent band of 1 is the theoretical optimum for equal sequences, resulting from expanding only the states on the main diagonal. The equivalent band for CSH is always lower than the equivalent band for SH because the chaining seed heuristic dominates over seed heuristic. In practice, this difference becomes significant for large enough e and n (e>5% and $n>10^5$). The equivalent band of SH and CSH is constant for e=1%, indicating a linear scaling with n. In other cases ($e\leq$5% for SH and $e\leq$15% for CSH), the band increases slowly with n, indicating a near-linear scaling.

2.8.7 *Performance variation on human genome*

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

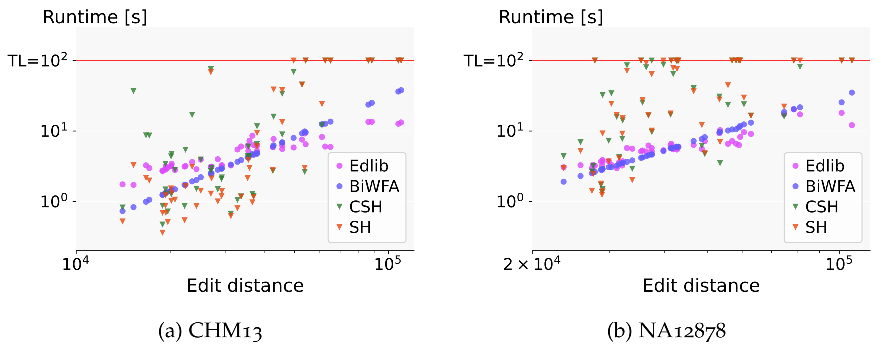


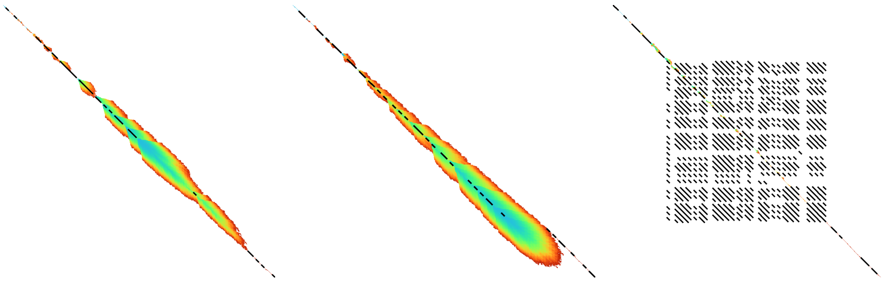
FIGURE 2.7: Log-log plots of the runtime for aligning **real sequences** from two datasets of varying error distance. The sequence length varies between 500 kbp and 840 kbp for the CHM13 reads, and between 502 kbp and 1 053 kbp for the NA12878 reads. Alignments that timed out after 100 seconds are shown at 100 s. Parameters used for SH and CSH are k=15 and r=2.

2.8.8 *Complex alignments*

Our algorithm finds optimal alignments very efficiently when both the sequence and the errors are uniformly random and the error rate is limited (§2.5). Nevertheless, since the alignment problem is fundamentally unsolvable in strictly subquadratic time [6], there are sequences which cannot be aligned fast. We give three cases (Fig. 2.8) where the complexity of our algorithm degrades.

1. *High error rate.* When the error rate becomes too high (larger than r/k), seeds can not increase the heuristic enough penalize all errors. Each unpenalized error increases f and needs more states to be searched, similar to Dijkstra. This can be mitigated by using shorter seeds and/or inexact matches, at the cost of introducing more matches.

2. *Long indel.* If there is a long insertion or deletion, the search has to accumulate the high cost for the long indel. Our heuristics do not account for long indels, and hence the search needs to expand states until the end of the gap is reached. Again, this causes a big increase of f and a search similar to Dijkstra. This may be improved in future work by introducing a *gap-cost* term to the chaining seed heuristic that penalizes gaps between consecutive matches in a chain.



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

FIGURE 2.8: Expanded states by CSH (r=2, k=10) when aligning pairs of **synthetic sequences** (n=1000) with 8% random mutations containing (a) a region of length 200 with larger error rate than the heuristic can efficiently account for (50%), (b) a deletion of length 100, and (c) 60 mutated copies of a pattern of length 10. Seed matches are shown in black and occur on the best paths and in the repeated region. The order of expansion is shown by the color gradient from blue to red.

3. *Short repeats.* When a short pattern repeats many times, this can result in a quadratic number of matches. This makes the corresponding seeds ineffective for the seed heuristic, and slows down the computation and pruning of the chaining seed heuristic. This can be partially mitigated by increasing the seed length and/or exact matches, at the cost of reducing the potential of the heuristic. Alternatively, seeds with too many matches could be completely ignored.

2.8.9 *Notation*

Table 3.4 summarizes the notation used in this work.

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

3 TRIE

We present an algorithm for the *optimal alignment* of sequences to *genome graphs*. It works by phrasing the edit distance minimization task as finding a shortest path on an implicit alignment graph. To find a shortest path, we instantiate the A* paradigm with a novel domain-specific heuristic function that accounts for the upcoming subsequence in the query to be aligned, resulting in a provably optimal alignment algorithm called AStarix.

Experimental evaluation of AStarix shows that it is 1–2 orders of magnitude faster than state-of-the-art optimal algorithms on the task of aligning Illumina reads to reference genome graphs. Implementations and evaluations are available at https://github.com/eth-sri/astarix.

keywords: Next-generation sequencing, Optimal alignment, Genome graph, Shortest path, A* algorithm

3.1 INTRODUCTION

The analysis and understanding of genetic variation encoded in the genome of an organism lies at the center of computational biology and medicine. Variation is usually identified through matching sequences obtained from DNA/RNA-sequencing back to a reference (genome) sequence in the process of *variant calling*, making the alignment task a core problem in sequence bioinformatics.

Historically, a single linear reference sequence has been used to represent the most common variants in a population. While providing a working abstraction for most cases, rare or sub-population specific variation is especially hard to model in this setting, creating a reference allele bias [43, 44]. Consequently, in the last few years, the field has shifted first towards using sets of reference sequences, and more recently to graph data structures (so-called *genome graphs*), to represent many genomes or haplotypes simultaneously [45–47].

Both for sequence-to-sequence alignment and sequence-to-graph alignment, heuristics are employed to keep alignment tractable [47–49], especially for large populations of human-sized genomes. While such heuristics find the correct alignment for simple references, they often perform poorly in regions of very high complexity, such as in the human major histocompatibility complex (MHC) [45], in complex but rare genotypes arising from somatic-subclones in tumor sequencing data [50], or in the presence of frequent sequencing errors [51]. Importantly, these cases can be of specific clinical or biological interest, and incorrect alignment can cause severe biases for downstream analyses. For instance, the combination of high variability of MHC sequences in humans and small differences between alleles [52] leads to a risk of misclassifications due to suboptimal alignment. Guaranteeing optimal alignment against all variations represented in a graph is a major step towards alleviating those biases.

Formally, we consider the optimal *sequence-to-graph alignment* problem, the task of finding an optimal base-to-base correspondence between a query sequence and a (possibly cyclic) walk in the graph. Related alignment problems have already been formulated as graph shortest path problems [53, 54].

3.1.1 *Related Work*

Seed-and-Extend. Since optimal alignment is often intractable, many aligners use heuristics, most commonly the *seed-and-extend* paradigm [48, 49, 55]. In this approach, alignment initiation sites (*seeds*) are determined, which are then *extended* to form the *alignments* of the query sequence. The fundamental issue with this approach, however, is that the seeding and extension phases are mostly decoupled during alignment. Thus, an algorithm with a provably optimal extension phase may not result in optimal alignments due to the selection of a suboptimal seed in the first phase. In cases of high sequence variability, the seeding phase may even fail to find an appropriate seed from which to extend.

Accounting for Variation. First attempts to include variation into the reference data structure were made by augmenting the local alignment method to consider alternative walks during the extend step [56, 57]. This approach has since been extended from the linear reference case to graph references. To represent non-reference variation of multiple references during the seeding stage, HISAT2 uses generalized compressed suffix arrays [58] to index walks in an augmented reference sequence, forming a local genome graph [59]. VG [47] uses a similar technique [60] to index variation graphs representing a population of references.

BrownieAligner, another recent work developed for local alignment of sequences to *de Bruijn* graph representations of genomic variation, features an optimal extension phase using a branch-and-bound-based early cutoff, while employing a heuristic maximal-exact-match approach for seeding [61].

Optimal Alignment. Current optimal alignment algorithms reach the impractical $O(nm)$ runtime that has been shown to be a lower bound for the worst-case edit distance computation [6]. In this light, approaches for improving the efficiency of optimal alignment have taken advantage of specialized features of modern CPUs to improve the practical runtime of the Smith-Waterman dynamic programming (DP) algorithm [62] considering all possible starting nodes. These use modern SIMD instructions (e.g. VG [47] and PaSGAL [63]) or reformulations of edit distance computation to allow for bit-parallel computations in GraphAligner[1] [64]. Many of these, however, are designed only for specific types of genome graphs, such as *de Bruijn* graphs [61, 65, 66] and variation graphs [47]. A compromise often made when aligning sequences to cyclic graphs using algorithms reliant on directed acyclic graphs involves the computationally expensive "DAG-ification" of graph regions [47, 67].

A* algorithm. We aim to guarantee optimal alignment while optimizing the average runtime to not reach its worst-case complexity. While Dijkstra is an algorithm that explores graph nodes in the order of their distance from the start, A* is a generalization of Dijkstra that also accounts for their distance from the target. A* prioritizes the exploration of nodes that seem to be closer to the target nodes. This way, A* can sometimes dramatically improve on the performance of Dijkstra while remaining optimal.

There has been one attempt to apply A* for optimal alignment [68] which uses a heuristic function that accounts only for the length of the remaining query sequence to be aligned. However, it does not significantly outperform Dijkstra (in fact, it is equivalent for a zero matching cost). In contrast, the heuristic function we introduce is more informative and consistently outperforms Dijkstra.

3.1.2 *Main Contributions*

We introduce a novel approach, called AStarix, for optimal sequence-to-graph alignment based on A*. As with any A* instantiation, the core diffi-

1 We refer as BitParallel to to the bit-parallel DP algorithm implemented in GraphAligner tool [64].

culty lies in developing an accurate domain-specific heuristic which is fast to compute. We design a heuristic that accounts for the content of the upcoming query letters to be aligned, which more effectively guides the search. Our proposed heuristic has two advantages: (i) it is correctness-preserving, that is, it preserves the fact that AStarix finds the best alignment, yet (ii) it is practically effective in that the algorithm performs a near-optimal number of steps. Overall, this heuristic enables AStarix to compute the best alignment while also scaling to larger reference graph sizes when compared to existing state-of-the-art optimal aligners.

Our main contributions[2] include:

1. **AStarix.** An algorithm for optimal sequence-to-graph alignment based on a novel instantiation of $A^\star$ with an accurate domain-specific heuristic that accounts for the upcoming query letters to be aligned (§3.3).
2. **Algorithmic optimizations.** To ensure that AStarix is practical, we introduce a number of algorithmic optimizations which increase performance and decrease memory footprint (§3.4). We also prove that all optimizations are correctness-preserving.
3. **Thorough experimental evaluation of AStarix.** We demonstrate that AStarix is up to 2 orders of magnitude faster than other optimal aligners on various reference graphs (§3.11).

3.2 TASK DESCRIPTION: ALIGNMENT TO REFERENCE GRAPHS

We now describe the task of aligning a query to a reference graph. To this end, we (i) introduce the task of optimal alignment on a *reference graph*, (ii) formalize this task in terms of an *edit graph*, and (iii) introduce an alternative formulation in terms of an *alignment graph*, which is the basis for shortest path formulations of the optimal alignment. Fig. 3.1 summarizes these different graph types.

Reference Graph. We encode the collection of references to which we want to align in a reference graph, which captures genomic variation that a linear reference cannot express [46, 47]. We formalize a reference graph as a tuple $G_\mathrm{r} = (V_\mathrm{r}, E_\mathrm{r})$ of nodes V_r and directed, labeled edges $E_\mathrm{r} \subseteq V_\mathrm{r} \times V_\mathrm{r} \times \Sigma$, where the alphabet $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ represents the four different nucleotides.

2 The appendix with algorithms and evalution details is included in the full version of this paper: https://www.biorxiv.org/content/10.1101/2020.01.22.915496v1

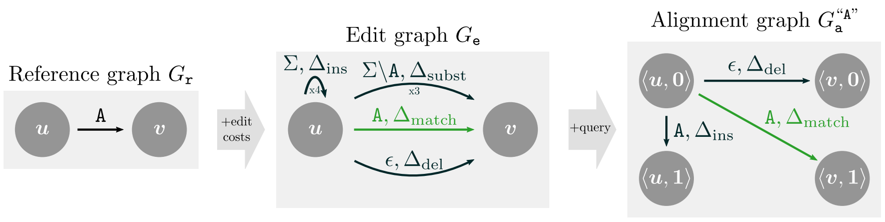


FIGURE 3.1: Starting from the reference graph (left), we can construct the edit graph (middle) and the alignment graph G_a^q for query $q =$ “A” (right). Edges are annotated with labels and/or costs, where sets of labels represent multiple edges, one for each letter in the set (indicated by “x3” and “x4”).

Note that in contrast to sequence graphs [69], we label edges instead of nodes.

Path, Spelling. Any path $\pi = (e_1, \ldots, e_k)$ in G_r induces a *spelling* $\sigma(\pi) \in \Sigma^*$ defined by $\sigma(e_1) \cdots \sigma(e_k)$, where $\sigma(e_i)$ is the label of edge e_i and $\Sigma^* := \bigcup_{k \in \mathbb{N}} \Sigma^k$. We note that our approach naturally handles cyclic walks and does not require cycle unrolling, a feature shared with BITPARALLEL [64] and BROWNIEALIGNER [61] but missing from VG [47], PASGAL [63] and V-ALIGN [67].

Alignment on Reference Graph. An *alignment* of *query* $q \in \Sigma^*$ to a reference graph $G_r = (V_r, E_r)$ consists of (i) a path π in G_r and (ii) a sequence of edit operations (matches, substitutions, insertions, deletions) transforming $\sigma(\pi)$ to q.

Optimal Alignment, Edit Distance. Each edit operation is associated with a real-valued cost (Δ_{match}, Δ_{subst}, Δ_{ins}, and Δ_{del}, respectively). An optimal alignment minimizes the total cost of the edit operations converting $\sigma(\pi)$ to q. For optimal alignments, this total cost is equal to the edit distance between $\sigma(\pi)$ and q, i.e., the cheapest sequence of edit operations transforming $\sigma(\pi)$ into q.

We make the (standard) assumption that $0 \leq \Delta_{match} \leq \Delta_{subst}, \Delta_{ins}, \Delta_{del}$, which will be a prerequisite for the correctness of our approach.

Edit Graph. Instead of representing alignments as pairs of (i) paths in the reference graph and (ii) sequences of edit operations on these paths, we introduce *edit graphs* whose paths intrinsically capture both. This way, we can formally define an alignment more conveniently as a path in an edit graph.

Formally, an *edit graph* $G_{\mathsf{e}} := (V_{\mathsf{e}}, E_{\mathsf{e}})$ has directed, labeled edges $E_{\mathsf{e}} \subseteq V_{\mathsf{e}} \times V_{\mathsf{e}} \times \Sigma_{\epsilon} \times \mathbb{R}_{\geq 0}$ with associated costs that account for edits. Here, $\Sigma_{\epsilon} := \Sigma \cup \{\epsilon\}$ extends the alphabet Σ by ϵ to account for deleted characters (see Fig. 3.1). The edit and reference graphs consist of the same vertices, i.e., $V_{\mathsf{e}} = V_{\mathsf{r}}$. However, E_{e} contains more edges than E_{r} to account for edits. Concretely, for each edge $(u, v, \ell) \in E_{\mathsf{r}}$, E_{e} contains edges to account for (i) matches, by an edge $(u, v, \ell, \Delta_{\text{match}})$, (ii) substitutions, by edges $(u, v, \ell', \Delta_{\text{subst}})$ for each $\ell' \in \Sigma \backslash \ell$, (iii) deletions, by an edge $(u, v, \epsilon, \Delta_{\text{del}})$, and (iv) insertions, by edges $(u, u, \ell', \Delta_{\text{ins}})$ for each $\ell' \in \Sigma$. The spelling $\sigma(\pi) \in \Sigma^*$ of a path $\pi \in G_{\mathsf{e}}$ is defined analogously to reference graphs, except that deleted letters (represented by ϵ) are ignored. The cost $\operatorname{cost} \pi$ of a path $\pi \in G_{\mathsf{e}}$ is the sum of all its edge costs.

Alignment on Edit Graph. An *alignment* of query q to G_{r} is a path π in G_{e} spelling q, i.e., $q = \sigma(\pi)$. An *optimal alignment* is an alignment of minimal cost.

Alignment Graph. To find an optimal alignment of q to the edit graph G_{e} using shortest path finding algorithms, we must ensure that only paths spelling q are considered. To this end, we introduce an alternative but equivalent formulation of alignments in terms of an *alignment graph* $G_{\mathsf{a}}^q = (V_{\mathsf{a}}^q, E_{\mathsf{a}}^q)$.

Here, each *state* $\langle v, i \rangle \in V_{\mathsf{a}}^q$ consists of a vertex $v \in V_{\mathsf{e}}$ and a query position $i \in \{0, \ldots, |q|\}$ (equivalent to [69]). Traversing a state $\langle v, i \rangle \in V_{\mathsf{a}}^q$ represents the alignment of the first i query characters ending at node v. In particular, query position $i = 0$ indicates that we have not yet matched any letters from the query. We note that the alignment graph explicitly depends on the query q. In particular, the example alignment graph $G_{\mathsf{a}}^{\text{``A''}}$ in Fig. 3.1 lacks substitution edges from $G_{\mathsf{e}}$, as their labels (`C`, `G`, `T`) do not match the query $q = \text{``}A\text{''}$.

We construct the alignment graph G_{a}^q to guarantee that any walk from a source $\langle u, 0 \rangle$ to a state $\langle v, i \rangle$ corresponds to an alignment of the first i letters of query q to G_{r}. As a consequence, there is a one-to-one correspondence between alignments π_{e} of q to G_{e} and paths $\pi_{\mathsf{a}}^q \in G_{\mathsf{a}}^q$ from sources $S := V_{\mathsf{r}} \times \{0\}$ to targets $T := V_{\mathsf{r}} \times \{|q|\}$, with $\operatorname{cost} \pi_{\mathsf{r}} = \operatorname{cost} \pi_{\mathsf{a}}^q$. To find the best alignment in G_{e}, only paths in G_{a}^q (walks without repeating nodes) can be considered, since repeating a node in G_{a}^q cannot lead to a lower cost ($\Delta_{\text{del}} \geq 0$) for the same state.

The edges $E_\mathsf{a}^q \subseteq V_\mathsf{a}^q \times V_\mathsf{a}^q \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ are built based on the edges in E_e, except that the former (i) keep track of the position in the query i, and (ii) only contain empty edges or edges whose label matches the next query letter:

$$(u, v, \ell, w) \in E_\mathsf{e} \implies (\langle u, i\rangle, \langle v, i+1\rangle, \ell, w) \in E_\mathsf{a}^q \quad \text{for } 0 \leq i < |q| \text{ with } q[i] = \ell \tag{3.1}$$

$$(u, v, \epsilon, w) \in E_\mathsf{e} \implies (\langle u, i\rangle, \langle v, i \quad \rangle, \epsilon, w) \in E_\mathsf{a}^q \quad \text{for } 0 \leq i < |q| \tag{3.2}$$

Here, assuming 0-indexing, $q[i]$ is the next letter to be matched after matching i letters. Then, Eq. (3.1) represents matches, substitutions, and insertions (which advance the position in the query by 1), while Eq. (3.2) represents deletions (which do not advance the position in the query).

Dynamic Construction. As the size of the alignment graph is $O(|G_\mathsf{r}| \cdot |q|)$, it is expensive to build it fully for every new query. Therefore, our implementation constructs the alignment graph G_a^q on-the-fly: the outgoing edges of a node are only generated on demand and are freed from memory after alignment.

3.3 ASTARIX: FINDING OPTIMAL ALIGNMENTS USING A*

In this section, we first introduce the general A* algorithm for finding shortest paths, and the notion of an optimistic heuristic, a sufficient condition for instantiations of A* to be correct (i.e., to indeed find shortest paths). Then we instantiate A* with our domain-specific heuristic that accounts for upcoming subsequences to be aligned, and prove that this heuristic is optimistic.

3.3.1 *Background: General A* algorithm*

Given a weighted graph $G = (V, E)$ with $E \subseteq V \times V \times \mathbb{R}_{\geq 0}$, the A* algorithm (abbreviated as A*) searches for the shortest path from sources $S \subseteq V$ to targets $T \subseteq V$. It is an extension of Dijkstra's algorithm that additionally leverages a *heuristic function* $h \colon V \to \mathbb{R}_{\geq 0}$ to decide which paths to explore first. If $h(u) \equiv 0$, A* is equivalent to Dijkstra's algorithm. We provide an implementation of A* and Dijkstra in §3.14.1, but do not assume knowledge of either algorithm in the following. At a high level, A* maintains the set of all *explored* states, initialized with the set of sources S. Then, A* iteratively *expands* the explored state with lowest estimated cost by exploring all its

Algorithm 1 AStarix including heuristic function.

```
 1: $G_r$: Reference graph                                   ▷ Global variables
 2: $d$: Upcoming sequence length

 3: function AStarix($q$: Query)
 4:     $G_a^q \leftarrow$ DefineAlignmentGraph($G_r, q$)        ▷ Following §3.9.1
 5:     $S \leftarrow \{\langle v,i\rangle \in V_a^q \mid i = 0\}$        ▷ Sources: no letter matched
 6:     $T \leftarrow \{\langle v,i\rangle \in V_a^q \mid i = |q|\}$      ▷ Targets: all letters matched
 7:     return $A^\star(G_a^q, S, T,$ Heuristic$)$           ▷ $A^\star$ provided in §3.14.1

 8: function Heuristic($\langle u,i\rangle$ : State)     ▷ Heuristic: Cost of upcoming
    sequence
 9:     $d' \leftarrow \min(d, |q| - i)$            ▷ Actual length of upcoming sequence
10:     $s \leftarrow q[i : i + d']$   ▷ Upcoming sequence (next $d$ letters after current)
11:     return $h(u, s)$                  ▷ Cost of aligning $s$ to $G_e$ starting from $u$

12: function $h(u, s)$                          ▷ Cost of aligning $s$ starting from $u$
13:     return RecursiveAlign($u, s, 0.0, \infty$)   ▷ Simple branch-and-bound
```

neighbors, until it finds a target. Here, the cost for node u is estimated by the distance from source, called $g(u)$, plus the estimate from the heuristic $h(u)$.

Heuristic Function. The heuristic function $h(u)$ estimates the cost $h^*(u)$ of a shortest path in G from u to a target $t \in T$. Intuitively, a good heuristic correlates well with the distance from u to t.

To ensure that $A^\star$ indeed finds the shortest path, h should be *optimistic*:

Definition 6 (Optimistic heuristic). *A heuristic h is optimistic if it provides a lower bound on the distance to the closest target: $\forall u.h(u) \leq h^*(u)$.*

While any optimistic h ensures that $A^\star$ finds optimal alignments [70], the specific choice of h is critical for performance. In particular, decreasing the error $\delta(u) = h^*(u) - h(u)$ can only improve the performance of $A^\star$ [70]. Thus, a key contribution of ours is a domain-specific heuristic h.

3.3.2 AStarix: *Instantiating A**

Algorithm 1 shows an unoptimized version of AStarix and its heuristic function. AStarix expects a reference graph (Algorithm 1) and a query (Algorithm 1) as input, and returns an optimal alignment (Algorithm 1) by searching for a shortest path from S to T in the alignment graph G_a^q. It is parameterized by hyper-parameters (d in Algorithm 1, more in §3.4) and edit costs (implicitly provided).

The function Heuristic (Algorithms 1–1) computes a lower bound on the remaining cost of a best alignment: the minimum cost $h(u,s)$ of aligning the *upcoming sequence* s (where $|s| \leq d$) starting from node u. Importantly, s is limited to the next $d' \leq d$ letters of q, starting from query position i. Thus, computing $h(u,s)$ is substantially cheaper than aligning all remaining letters of q.

To compute $h(u,s)$ we leverage a simple branch-and-bound algorithm, provided in §3.7.2. In the following, for convenience, we refer to the heuristic as h (which is parameterized by (u,s)) instead of Heuristic (which is parameterized by $\langle u,i\rangle$). Further, we say that h is optimistic if $h(u,s)$ is a lower bound on the cost for aligning all remaining letters (i.e., $q[i : |q|]$) starting from node u (note that s is a prefix of $q[i : |q|]$).

Theorem 6. *h is optimistic.*

Proof. h only considers the next d' letters of q instead of all remaining letters. Since all costs are non-negative, the theorem follows. □ □

Benefit of A* Heuristic over Dijkstra. Fig. 3.2 shows the benefit of using our heuristic function compared to Dijkstra. Here, Dijkstra expands states based on their distance g from the origin nodes $\langle u,0\rangle$ and $\langle v,0\rangle$. Hence, depending on tie-breaking, Dijkstra may expand all states with $h \leq 1$, as shown in Fig. 3.2. By contrast, A* chooses the next state to expand by the sum of the distance from the origin g and the heuristic h, expanding only states with $g + h \leq 1$.

Memoization. Recall that the return value of h in Algorithm 1 only depends on u and the upcoming sequence s (which in turn depends on i and d). Thus, $h(u,s)$ can be reused for different positions across different queries in $O(1)$ time, if it was computed for a previous query.

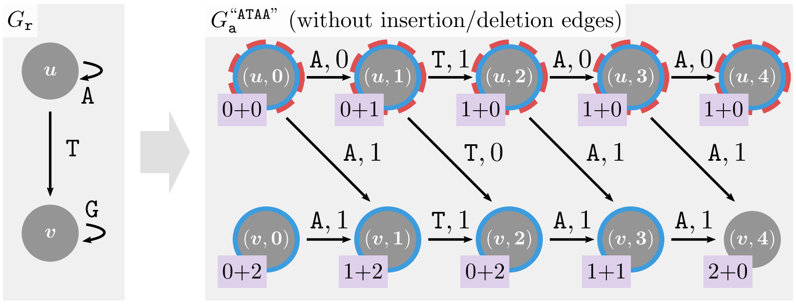


FIGURE 3.2: The benefit of using our heuristic over Dijkstra. Alignment graph $G_a^{\text{“ATAA”}}$ (right) is based on reference graph G_r (left), but omits insertion and deletion edges for simplicity. The pink boxes $g + h$ indicate the distance from the sources $S = \{\langle u, 0\rangle, \langle v, 0\rangle\}$ (in g) and the cost of aligning the next $d = 2$ letters (in h). Dijkstra (resp. A*) expands states circled in blue (resp. dashed red).

3.4 ASTARIX ALGORITHM: OPTIMIZATIONS

We now discuss several optimizations we developed to speed up ASTARIX while preserving its optimality. These optimizations reduce preprocessing and alignment runtime as well as memory footprint (in particular for memoization).

3.4.1 *Reducing Semi-global to Local Alignment Using a Trie*

To find an optimal alignment, we generally need to consider all reference graph nodes $u \in G_r$ as possible starting nodes. Thus, optimal aligners PASGAL [63] and BITPARALLEL [64] brute-force through all possible starting nodes $u \in G_r$.

To more efficiently handle arbitrary starting positions for alignments, we extend the reference graph with a trie (referred to as *suffix tree* in [68]) to effectively align from all possible starting nodes *simultaneously*.

Single Starting State. In the trie approach, abstraction nodes are added to the graph, each of which corresponds to a set of nodes in G_r that correspond to the same prefix. In the following, we formalize this approach.

Concretely, we extend G_r by a *trie of depth D*, resulting in graph $G_r^+ = (V_r^+, E_r^+)$. Our goal is that all paths in G_r that have length D and end in $v \in V_r$ correspond to paths in G_r^+ starting from a single source ϵ to $v \in V_r^+$, where ϵ represents the empty string. This correspondence ensures that it suffices to consider only paths in G_r^+ starting from the source ϵ. In particular, each alignment on G_r^+ can be translated into an alignment on G_r (we omit this translation here).

Fig. 3.10 shows an example trie. To construct it, we first associate with every node $v \in V_r$ the set $\mathcal{S}_v$ of its D-mers (orange boxes in Fig. 3.10): spells of paths ending in v and of length D. Our goal is then to use paths in the trie to spell these D-mers.

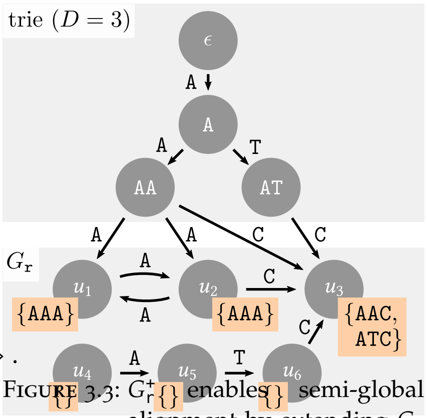


FIGURE 3.3: G_r^+ enables semi-global alignment by extending G_r with a trie.

Second, we construct the trie nodes from all prefixes of these D-mers:

$$V_r^+ := V_r \cup \bigcup_{v \in V_r} \left\{ s[0:i] \;\middle|\; \begin{array}{c} s \in \mathcal{S}_v, \\ 0 \le i < D \end{array} \right\}.$$

Third, we add edges within the trie, which ensure that paths from ϵ to any trie node s spell s. Formally, whenever $s{\cdot}\ell \in V_r^+$, we add an edge $(s, s{\cdot}\ell, \ell)$ to E_r^+, where "$\cdot$" denotes string concatenation. Finally, we add edges between the trie and the reference graph, which ensure that any D-mer of any node $v \in V_r$ can be spelled by a walk from ϵ to v. Formally, if $s{\cdot}\ell \in \mathcal{S}_v$, then $(s, v, \ell) \in E_r^+$.

Importantly, extending G_r to G_r^+ is compatible with the construction of the edit graph G_e, the construction of the alignment graph and all other optimizations. In particular, when searching for a shortest path in the alignment graph constructed from G_r^+, it suffices to only consider starting node $\langle \epsilon, 0 \rangle$.

Reducing Size of Trie. We can reduce the size of the trie by removing specific trie nodes. In particular, we iteratively remove each trie leaf node $s \cdot \ell \in V_r^+$ with a unique outgoing edge $(s \cdot \ell, v, \ell')$ to a reference graph node $v \in V_r$. To compensate for removing node $s \cdot \ell$, we introduce a new edge (s, u, ℓ) to a node $u \in V_r$ with an edge (u, v, ℓ') (such a node must exist according to the construction of G_r^+). For example, in Fig. 3.10, we (i) remove node AT including its edges $(\texttt{A}, \texttt{AT}, \texttt{T})$ and $(\texttt{AT}, u_3, \texttt{C})$, but (ii) introduce an edge $(\texttt{A}, u_2, T)$.

This optimization is lossless, as the D-mer $s \cdot \ell \cdot \ell' \in \mathcal{S}_v$ can still be spelled by the path from ϵ to s, extended by (s, u, ℓ) and (u, v, ℓ').

3.4.2 *Greedy Match Optimization*

We also employ an optimization originally developed for computing the edit distance between two strings [39, 71], but which has also been used in the context of string to graph alignment [68]. We omit the correctness proof of this optimization, which is already covered in [71], and only explain the intuition behind it.

Suppose there is only one outgoing edge $e = (u, v, \ell) \in E_\mathsf{r}$ from a node $u \in V_\mathsf{r}$. Suppose also that while aligning a query q, we explore state $\langle u, i\rangle$ for which the next query letter $q[i]$ matches the label ℓ. In this case, we do not need to consider the edit outgoing edges, because any edit at this point can be postponed without additional cost, as $\Delta_\text{match} \leq \min(\Delta_\text{subst}, \Delta_\text{ins}, \Delta_\text{del})$. Thus, we can greedily explore state $\langle v, i+1\rangle$, aligning $q[i+1]$ to e by using the edge $(\langle u, i\rangle, \langle v, i+1\rangle, \ell, \Delta_\text{match})$ before continuing with the A* search. We note that this optimization is only applicable when aligning in non-branching regions of the reference graph. In particular, it is not applicable for most trie nodes (§3.4.1).

3.4.3 *Speeding Up Evaluation of Heuristic*

In the following, we show how to reduce the runtime of evaluating the heuristic $h(u, s)$, by introducing two separate optimizations that compose naturally.

Capping Cost. We cap $h(u, s)$ at c, replacing it by $h_c(u, s) := \min(h(u, s), c)$. To achieve this, we allow RecursiveAlign to ignore paths costing more than c. For large enough c, this speeds up computation without significantly decreasing the benefit of the heuristic, since nodes associated with a high heuristic value are typically not explored anyways. We investigate the effect of c in §3.7.3.

Theorem 7. *h_c is optimistic.*

Proof. We have $h_c(u, s) \leq h(u, s)$ and that $h(u, s)$ is optimistic (**??** 6). □ □

Capping Depth. We reduce the number of nodes that need to be considered by $h(u, s)$. To this end, we define a modified heuristic $h_d(u, s)$ that only considers nodes $R_u \subseteq V_\mathsf{e}$ at distance at most d from u (cp. Algorithm 1 in Algorithm 1): $R_u := \{v \in V_\mathsf{r} \mid \exists \text{ path } \pi \in G_\mathsf{e} \text{ from } u \text{ to } v \text{ with } |\pi| \leq d\}$.

If an alignment of s reaches the boundary of R_u, defined as

$$B(R_u) := \{v \in R_u \mid \exists (v, v', \ell) \in E_{\mathsf{e}} \text{ with } v' \notin R_u\},$$

it is allowed to only spell a prefix of s, and the remaining unaligned letters of s are considered aligned with zero cost:

$$h_d(u, s) := \min_{\pi \in \Pi} \operatorname{cost} \pi, \text{ where}$$
$$\Pi := \left\{ \pi \in G_{\mathsf{r}} \;\middle|\; \mathit{start}(\pi) = u, \sigma(\pi) = s \vee \big(\mathit{end}(\pi) \in B(R_u) \wedge \exists i. \sigma(\pi) = s[1..i]\big) \right\}$$

Theorem 8. *h_d is optimistic.*

Proof. It suffices to show $h_d(u, s) \leq h(u, s)$ since $h(u, s)$ is optimistic. In the case where all of s is aligned, $h_d(u, s) = h(u, s)$. Otherwise, the unaligned letters of s are not penalized, so $h_d(u, s) \leq h(u, s)$. □ □

3.4.4 *Partitioning Nodes into Equivalence Classes*

We have shown in §3.3.2 how to reuse an already computed $h(u, s)$ for repeating s across different queries and query positions. In the following, we additionally aim to reuse $h(u, s)$ across different nodes u, so that $h(u, s)$ does not need to be computed for all nodes u. Intuitively, we want to assign two nodes u and v to the same equivalence class when the *graph region* considered by $h(u, s)$ is equivalent to the graph region considered by $h(v, s)$, up to renaming of nodes.

Thus, $h(u, s) = h(v, s)$ if u and v are from the same equivalence class. Therefore, we can (arbitrarily) choose a representative node $r \in V_{\mathsf{r}}$ for every equivalence class, and evaluate $h(r, s)$ instead of $h(u, s)$, where r is the representative of the equivalence class of u. To look up representative nodes in $O(1)$, we define a helper array *repr* with $\mathit{repr}[u] = r$.

Identifying Equivalence Classes. To identify the nodes belonging to the same equivalence class, we assume the optimization from §3.4.3, i.e., that our heuristic only considers nodes up to a distance d from u. Moreover, for performance reasons, our implementation detects only the equivalence classes of nodes u with a single outgoing path of length at least d. In this case, u and u' are in the same equivalence class if their outgoing paths spell the same sequence. In contrast, we leave nodes with forking paths in separate equivalence classes.

Note that for smaller d, the number of equivalence classes gets smaller, the reuse of the heuristic gets higher, and the memoization table has a lower memory footprint. At the same time, however, the heuristic $h_d(u,s)$ is less informative.

3.5 EVALUATION

In this section we present a thorough experimental evaluation[3] of AStarix on simulated Illumina reads. Our evaluation demonstrates that:

1. AStarix is faster than Dijkstra because the heuristic reduces the number of explored states by an order of magnitude.
2. The runtime of AStarix scales better than state-of-the-art optimal aligners with increasing graph size, on a variety of reference graphs.

3.5.1 *Implementation of* AStarix *and Dijkstra*

Our AStarix implementation uses an adjacency list graph data structure to represent the reference and the trie in a unified way, representing each letter by a separate edge object. To represent the reverse complementary walks in G_{r}, the vertices are doubled, connected in the opposite direction, and labeled with complementary nucleotides ($\mathtt{A} \leftrightarrow \mathtt{T}$, $\mathtt{C} \leftrightarrow \mathtt{G}$). We do not limit the number of memoized heuristic function values (§3.3.2), but note we could do so by resetting the memoization table periodically. Our implementation of Dijkstra reuses the same AStarix codebase except the use of a heuristic function (i.e., with $h \equiv 0$).

We apply all described optimizations to AStarix and Dijkstra, except §3.4.3 and §3.4.4 which are applicable only to AStarix.

While the optimality of AStarix is not affected by its parameters, its performance is (see §3.7.3 for analysis). To compare with other aligners, we use values $d = 5$, $c = 5$, $D = \lfloor \log_\Sigma |G_{\mathsf{r}}| \rfloor$.

3.5.2 *Compared Aligners:* PaSGAL *and* BitParallel

We compare the performance of AStarix to that of two state-of-the-art optimal aligners: PaSGAL and BitParallel, with their default parameters. We do not compare to the exact aligner of VG as (i) its optimal alignment is intended for testing purposes only, (ii) it does not provide an interface for aligning a set of reads, and (iii) it has been consistently outperformed by PaSGAL [63].

3 https://github.com/eth-sri/astarix/tree/RECOMB2020_experiments

PaSGAL is compiled with AVX2 SIMD support. The resulting alignments are not expected to match exactly between the local aligner PaSGAL and the semi-global aligners (AStarix and BitParallel) as they solve different tasks with different edit costs. Nevertheless, in analogy with the evaluations of PaSGAL [63], it is still meaningful to compare performance, assuming that the dynamic programming approach of PaSGAL can be adapted to semi-global alignment with similar performance.

Both BitParallel and PaSGAL reach their worst-case runtime complexity independent of the edit costs $\Delta = (\Delta_{\text{match}}, \Delta_{\text{subst}}, \Delta_{\text{ins}}, \Delta_{\text{del}})$. PaSGAL is evaluated using its default costs $\Delta = (-1, 1, 1, 1)$ and BitParallel is evaluated using the only supported costs $\Delta = (0, 1, 1, 1)$.

3.5.3 *Setting*

All evaluations were executed singled-threaded on an Intel Core i7-6700 CPU running at 3.40GHz.

Reference Graphs and Reads. We designed three experiments utilizing three different reference graphs (in Table 3.1). The first is a linear graph without variation based on the *E. coli* reference genome (strain: K-12 substr. MG1655, ASM584v2 [72]). The other two are variation graphs taken from the PaSGAL evaluations [63]: they are based on the Leukocyte Receptor Complex (LRC, with 1 099 856 nodes and 1 144 498 edges), and the Major Histocompatibility Complex (MHC1, with 5 138 362 nodes and 5 318 019 edges). We note that we do not evaluate on de Brujin graphs, since PaSGAL does not support cyclic graphs.

For the *E. coli* dataset we used the ART tool [73] to simulate an Illumina single-end read set with 10 000 reads of length 100. For the LCR and MHC1 datasets, we sampled 20 000 single-end reads of length 100 from the already generated sets in [63] using the Mason2 [74] simulator.

For Dijkstra and AStarix, the runtime complexity depends not only on the data size, but also on the data content, including edit costs. More accurate heuristics lead to better $A^\star$ performance [75], which is why we evaluate AStarix with costs corresponding more closely to Illumina error profiles: $\Delta = (0, 1, 5, 5)$.

Metrics. As all aligners evaluated in this work are provably optimal, we are mostly interested in their performance. To study the end-to-end performance of the optimal aligners, we use the Snakemake [76] pipeline framework to measure the execution time of every aligner (including the time

		Runtime and Memory			
Genome graph	**Size**	AStarix	Dijkstra	PaSGAL	BitParal
E. coli (linear)	~4.7 Mbp	33 sec	73 sec	3 272 sec	4 906 s
		0.66 GB	0.66 GB	0.55 GB	0.43
LCR (variant)	~1 Mbp	437 sec	940 sec	1 614 sec	SegFa
		1.12 GB	1.09 GB	0.30 GB	
MHC1 (variant)	~5 Mbp	1 282 sec	1 588 sec	>7 200 sec	SegFa
		4.35 GB	1.21 GB	0.87 GB	

Table 3.1: Performance of optimal aligners for different reference graphs.

spent on reading and indexing the reference graph input and outputting the resulting alignments). We note that the alignment phase dominates for all tools and experiments.

To judge the potential of heuristic functions, we measure not only the runtime but also the number of states explored by AStarix and Dijkstra. This number reflects the quality of the heuristic function rather than the speed of computation of the heuristic, the implementation and the system parameters.

3.5.4 *Comparison of Optimal Aligners*

Different Reference Graphs. Table 3.1 shows the performance of optimal aligners across various references. On all references, AStarix is consistently faster than Dijkstra, which is consistently faster than PaSGAL and BitParallel. The memory usage of Dijkstra is within a factor of 3 compared to PaSGAL and BitParallel. Due to the heuristic memoization, the memory usage of AStarix can grow several times compared to Dijkstra.

Scaling with Reference Graph Size. Fig. 3.4 compares the performance of existing optimal aligners. BitParallel and PaSGAL always explore all states, thus their average-case reaches the worst-case complexity of $O(|G_{\mathsf{a}}^{q}|) = O(m \cdot G_{\mathsf{r}})$. Due to the trie indexing, the runtime of AStarix and Dijkstra scales in the reference size with a polynomial of power around 0.2 versus the expected linear dependency of BitParallel and PaSGAL.

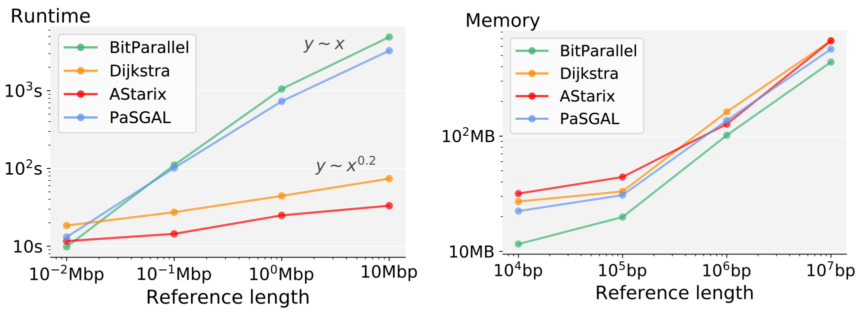


FIGURE 3.4: Comparison of overall runtime and memory usage of optimal aligners with increasing prefixes of E. coli as references.

The heuristic function of ASTARIX demonstrates a 2-fold speed-up over Dijkstra. This is possible due to the highly branching trie structure, which allows skipping the explicit exploration for the majority of starting nodes.

3.5.5 *A* Speedup*

To measure the speedup caused by the heuristic function, we compare the number of not only the expanded, but also of explored states (the latter number is never smaller, see §3.3.1 and the example in Fig. 3.2) between ASTARIX and Dijkstra on the MHC1 dataset.

Fig. 3.5 demonstrates the benefit of the heuristic function in terms of both alignment time and number of explored states. Most importantly, ASTARIX scales much better with increasing number of errors in the read, compared to Dijkstra. More specifically, the number of states explored by Dijkstra, as a function of alignment cost, grows exponentially with a base of around 10, whereas the base for ASTARIX is around 3 (the empirical complexity is estimated as a best exponential fit $exploredStates \sim a \cdot score^b$).

The horizontal black line in Fig. 3.5 denotes the total number of states $|G_r| \cdot |q|$, which is always explored by BITPARALLEL and PASGAL. On the other hand, any aligner must explore at least $m = |q|$ states, which we show as a horizontal dashed line. This lower bound is determined by the fact that at least the states on a best alignment need to be explored.

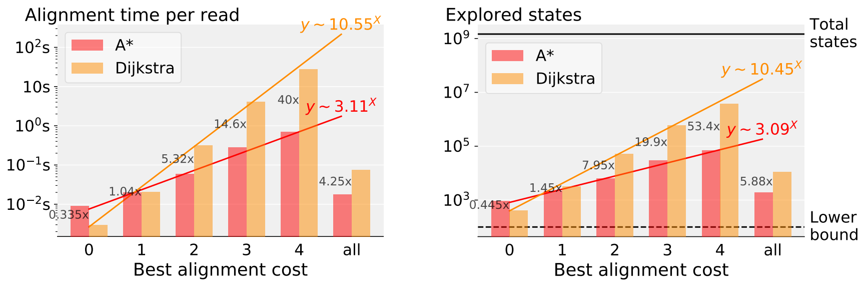


Figure 3.5: Comparison of A⋆ and Dijkstra in terms of mean alignment runtime per read and mean explored states depending on the best alignment cost on MHC1.

3.6 CONCLUSION

We presented AStarix, an A⋆ algorithm to find optimal alignments, based on a domain-specific heuristic and enhanced by multiple algorithmic optimizations. Importantly, our approach allows for both cyclic and acyclic graphs including variation and de Bruijn graphs.

We demonstrated that AStarix scales exponentially better than Dijkstra with increasing (but small) number of errors in the reads. Moreover, for short reads, both AStarix and Dijkstra scale better and outperform current state-of-the-art optimal aligners with increasing genome graph size. Nevertheless, scaling optimal alignment of long reads on big graphs remains an open problem.

We expect that AStarix can be scaled further, to both (i) bigger graphs and (ii) longer and noisier reads. Scaling AStarix may require a combination of (i) the development of more clever heuristic functions (by leveraging existing work on A⋆ and edit distance) and (ii) algorithmic optimizations. We note that if desired, a (sub-optimal) seeding step could speed up AStarix by pre-filtering the starting positions, analogously to other practical aligners.

3.7 APPENDIX

3.7.1 *Generic Algorithms: A⋆ and Dijkstra*

Algorithm 2 shows a generic implementation of the A⋆ algorithm, roughly following [70]. We do not implement the reconstruction of the best alignment in order to simplify the presentation. The procedure BACKTRACKPATH traces the best alignment back to the *source*, based on remembered edges used to optimize f for each alignment state. Algorithm 2 also shows a simple implementation of Dijkstra in terms of A⋆.

Algorithm 2 A⋆ algorithm (generalizes Dijkstra)

```
1:  function A⋆($G$: Graph, $S$: Sources, $T$: Targets, $h$: Heuristic function)
2:      $f \leftarrow Map(default = \infty)$: Nodes $\rightarrow \mathbb{R}_{\geq 0}$      ▷ Map nodes from $G$ to priorities
3:      $Q \leftarrow MinPriorityQueue(priority = f)$      ▷ Priorities according to $f$
4:      for all $s \in S$ do
5:          $f[s] \leftarrow 0.0$
6:          $Q.push(s)$                                   ▷ Initially, explore all $s \in S$
7:      while $Q \neq \emptyset$ do
8:          $curr \leftarrow Q.pop()$          ▷ Get state with minimal $f$ to be expanded
9:          if $curr \in T$ then
10:             return BACKTRACKPATH($curr$)      ▷ Reconstruct a path to $curr$ (omitted)
11:         for all $(curr, next, cost) \in G.outgoingEdges(curr)$ do
12:             $\hat{f}_{next} \leftarrow f[curr] + cost + h(next)$ ▷ Candidate value for $f[next]$
13:             if $\hat{f}_{next} < f[next]$ then
14:                 $f[next] \leftarrow \hat{f}_{next}$
15:                 $Q.push(next)$                        ▷ Explore state $next$
16:     assert $False$                ▷ Cannot happen if $T$ is reachable from $S$

17: function DIJKSTRA($G$: $Graph$, $S$: $Sources$, $T$: $Targets$)
18:     $h(v) \leftarrow 0.0$                             ▷ Constant-zero function $h$
19:     A⋆$(G, S, T, h)$
```

Algorithm 3 Recursive alignment used by Heuristic in Algorithm 1.

```
1: function RecursiveAlign($u, s, curr, best$)          ▷ Return value is $\leq best$
2:     if $curr \geq best$ then
3:         return $best$                         ▷ Branch and bound: bounding
4:     if $s = \epsilon$ then                                ▷ Reached a target
5:         return $curr$
6:     for all $(u, v, \ell, w) \in E_e$ where $\ell \in \{s[0], \epsilon\}$ do
7:         $suff = s[1:]$ if $\ell \neq \epsilon$ else $s$
8:         $best =$ RecursiveAlign($u, suff, curr + w, best$)
9:     return $best$
```

3.7.2 *Recursive Alignment Algorithm*

Algorithm 3 shows our implementation of RecursiveAlign, used in Algorithm 1 to evaluate h. RecursiveAlign is a simple branch-and-bound algorithm that recursively looks for the cheapest alignment of s starting from u, and does not follow paths whose cost exceeds *best*, the best path found so far.

3.7.3 *Parameter Estimation*

We now evaluate the influence of different parameter choices (c, d, D) on runtime and memory usage.

Fig. 3.6 demonstrates the benefit of using a trie with the size reduction optimization (end of §3.4.1): increasing the trie depth D speeds up aligning but requires more memory. Selecting the trie depth based on the graph size $D = \lfloor \log_\Sigma |G_r| \rfloor$ provides a reasonable trade-off between alignment time and memory.

§3.7.3 shows the joint effect of c and d. It demonstrates that having a long reach (d) that covers at least some errors ($c > 0$) is a reasonable strategy for choosing d and c.

3.7.4 *Versions, commands, parameters for running all evaluated approaches*

In the following, we provide details on how we executed the approaches discussed in §3.11:

PaSGAL

Obtained from	https://github.com/ParBLiSS/PaSGAL (Commit `50ad80c`)
Command	`PaSGAL -q reads.fq -r graph.vg -m vg -o output -t 1`

BitParallel

Obtained from	https://github.com/maickrau/GraphAligner/tree/WabiExperiments (Commit `241565c`)
Command	`Aligner -f reads.fq -g graph.gfa >output`

AStarix

Obtained from	https://github.com/eth-sri/astarix/tree/recomb2020
Command	`astarix align-optimal -f reads.fq -g graph.gfa >output`

Dijkstra

Obtained from	https://github.com/eth-sri/astarix/tree/recomb2020
Command	`astarix align-optimal -f reads.fq -g graph.gfa -a dijkstra >output`

3.7.5 *Notations*

Table 3.4 summarizes the notational conventions used in this work.

Object	Notation
Queries	$Q = \{q_i \| q_i \in \Sigma^m\}$
Query	$q \in Q$
Length	$m := \|q\| \in \mathbb{N}$
Position in query	$q[i] \in \Sigma, i \in \{1, \ldots, m\}$
Reference graph	$G_\mathsf{r} = (V_\mathsf{r}, E_\mathsf{r})$
Size	$\|G_\mathsf{r}\| := \|V_\mathsf{r}\| + \|E_\mathsf{r}\| \in \mathbb{N}$
Nodes	$u, v \in V_\mathsf{r}, n := \|V_\mathsf{r}\| \in \mathbb{N}$
Edges	$e \in E_\mathsf{r} := V_\mathsf{r} \times V_\mathsf{r} \times \Sigma$
Edge letter	$\ell \in \Sigma$
Reference graph with a trie	$G_\mathsf{r}^+ = (V_\mathsf{r}^+, E_\mathsf{r}^+)$
Trie depth	$D \in \mathbb{N}_{>0}$
Edit graph	$G_\mathsf{e} = (V_\mathsf{e}, E_\mathsf{e})$
Edit costs	$0 \leq \Delta_\text{match} \leq \Delta_\text{subst}, \Delta_\text{ins}, \Delta_\text{del}$
Alignment	$\pi \in E_\mathsf{e}^*$ and $\sigma(\pi) = q$
Optimal alignment	$\dot{\pi} \in E_\mathsf{e}^*$
Alignment cost	$\text{cost}\,\pi \in \mathbb{R}_{\geq 0}$
Alignment graph	$G_\mathsf{a}^q = (V_\mathsf{a}^q, E_\mathsf{a}^q)$
Size	$N := \|V_\mathsf{a}^q\| \in \mathbb{N}$
State	$\langle u, i \rangle \in V_\mathsf{a}^q := V \times \{0, \ldots, m\}$
Edges	$(\langle u, i \rangle, \langle v, j \rangle, \ell, w) \in E_\mathsf{a}^q \subseteq V_\mathsf{a}^q \times V_\mathsf{a}^q \times \Sigma_\epsilon \times$
Edge cost	$w \in \mathbb{R}_{\geq 0}$
In all graphs	$G(V, E) \in \{G_\mathsf{r}, G_\mathsf{e}, G_\mathsf{a}^q\}$
Walk	$\pi \in G : \pi \in E^*$
Walk spelling	$\sigma(\pi) \in \Sigma^*$
Walk begin and end nodes	$begin(\pi), end(\pi) \in V$
Path	a walk without repeating nodes
AStarix	$A^\star(G, S, T, h)$
Graph	$G = (V, E)$
Nodes	$u, v \in V$
Edges	$e \in E \subseteq V \times V \times \mathbb{R}_{\geq 0}$
Source states	$S \subseteq V$
Target states	$T \subseteq V$
Upcoming sequence	$s \in \Sigma^k$
Cost cap	$c \in \mathbb{R}_{\geq 0}$

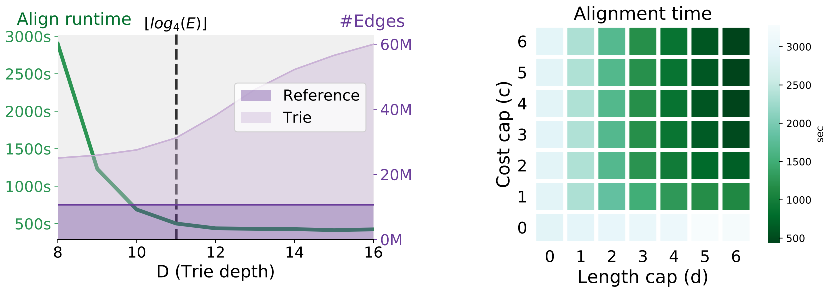


FIGURE 3.6: Left: Effect of D on performance of ASTARIX (MHC1 experiment). The dashed line shows our choice of D. Right: Runtime of ASTARIX depending on d and c (MHC1 experiment).

We present a novel A* *seed heuristic* that enables fast and optimal sequence-to-graph alignment, guaranteed to minimize the edit distance of the alignment assuming non-negative edit costs.

We phrase optimal alignment as a shortest path problem and solve it by instantiating the A* algorithm with our seed heuristic. The seed heuristic first extracts non-overlapping substrings (*seeds*) from the read, finds exact seed *matches* in the reference, marks preceding reference positions by *crumbs*, and uses the crumbs to direct the A* search. The key idea is to punish paths for the absence of foreseeable seed matches. We prove admissibility of the seed heuristic, thus guaranteeing alignment optimality.

Our implementation extends the free and open source aligner and demonstrates that the seed heuristic outperforms all state-of-the-art optimal aligners including GraphAligner, Vargas, PaSGAL, and the prefix heuristic previously employed by AStarix. Specifically, we achieve a consistent speedup of >60× on both short Illumina reads and long HiFi reads (up to 25kbp), on both the *E. coli* linear reference genome (1Mbp) and the MHC variant graph (5Mbp). Our speedup is enabled by the seed heuristic consistently skipping >99.99% of the table cells that optimal aligners based on dynamic programming compute.

AStarix aligner and evaluations: https://github.com/eth-sri/astarix
Full paper: https://www.biorxiv.org/content/10.1101/2021.11.05.467453

Genome graph, Optimal alignment, Semi-global alignment, Edit distance, Shortest path, Long reads, A* algorithm, Seed heuristic

3.8 INTRODUCTION

Alignment of reads to a reference genome is an essential and early step in most bioinformatics pipelines. While linear references have been used traditionally, an increasing interest is directed towards graph references capable of representing biological variation [47]. Specifically, a *sequence-to-graph* alignment is a base-to-base correspondence between a given read and a walk in the graph. As sequencing errors and biological variation result in inexact read alignments, edit distance is the most common metric that alignment algorithms optimize in order to find the most probable read origin in the reference.

Suboptimal alignment. In the last decades, approximate and alignment-free methods satisfied the demand for faster algorithms which process huge volumes of genetic data [7]. *Seed-and-extend* is arguably the most popular paradigm in read alignment [48, 49, 55]. First, substrings (called *seeds* or *kmers*) of the read are extracted, then aligned to the reference, and finally prospective matching locations are *extended* on both sides to align the full read.

While such a heuristic may produce acceptable alignments in many cases, it fundamentally does not provide quality guarantees, resulting in suboptimal alignment accuracy. In contrast, we demonstrate in this work that seeds can benefit optimal alignment as well.

Key challenges in optimal alignment. Finding optimal alignments is desirable but expensive in the worst case, requiring $O(Nm)$ time [77], for graph size N and read length m. Unfortunately, most optimal sequence-to-graph aligners rely on dynamic programming (DP) and always reach this worst-case asymptotic runtime. Such aligners include Vargas [78], PaSGAL [63], GraphAligner [64], HGA [79], and VG [47], which use bit-level optimizations and parallelization to increase their throughput.

In contrast, AStarix [23] follows the promising direction of using a heuristic to avoid worst-case runtime on realistic data. To this end, AStarix rephrases the task of alignment as a shortest-path problem in an *alignment graph* extended by a *trie index*, and solves it using the A* algorithm instantiated with a problem-specific prefix heuristic. Importantly, its choice of heuristic only affects performance, not optimality. Unlike DP-based algorithms, this prefix heuristic allows scaling sublinearly with the reference size, substantially increasing performance on large genomes. However, it can only efficiently align reads of limited length.

This work: Optimal alignment for short and long reads. In this work, we address the key challenge of scaling to long HiFi reads, while retaining the superior scaling of AStarix in the size of the reference graph. To this end, we instantiate the A* algorithm with a novel seed heuristic, which outperforms existing optimal aligners on both short and long HiFi reads. Specifically, the seed heuristic utilizes information from the whole read to narrowly direct the A* search by placing *crumbs* on graph nodes which lead up to a *seed match*, i.e., an exact match of a substring of the read.

Overall, the contributions presented in this work are:

1. A novel A* seed heuristic that exploits information from the whole read to quickly align it to a general graphs reference.

2. An optimality proof showing that the seed heuristic always finds an alignment with minimal edit distance.

3. An implementation of the seed heuristic as part of the AStarix aligner.
4. An extensive evaluation of our approach, showing that we align both short Illumina reads and long HiFi reads to both linear and graph references $\geq 60\times$ faster than existing optimal aligners.
5. A demonstration of superior empirical runtime scaling in the reference size N: $N^{0.46}$ on Illumina reads and $N^{0.11}$ on HiFi reads.

3.9 PREREQUISITES

We define the task of alignment as a shortest path problem (§3.9.1) to be solved using the A* algorithm (§3.9.2).

3.9.1 *Problem statement: Alignment as shortest path*

In the following, we formalize the task of optimally aligning a read to a reference graph in terms of finding a shortest path in an *alignment graph*. Our discussion closely follows [23] and is in line with [69].

Reference graph. A reference graph $G_\mathsf{r} = (V_\mathsf{r}, E_\mathsf{r})$ encodes a collection of references to be considered when aligning a read. Its directed edges $E_\mathsf{r} \subseteq V_\mathsf{r} \times V_\mathsf{r} \times \Sigma$ are labeled by nucleotide letters from $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, hence any walk π_r in G_r spells a string $\sigma(\pi_\mathsf{r}) \in \Sigma^*$.

An alignment of a read $q \in \Sigma^*$ to a reference graph G_r consists of (i) a walk π_r in G_r and (ii) a sequence of edits (matches, substitutions, deletions, and insertions) that transform $\sigma(\pi_\mathsf{r})$ to q. An alignment is *optimal* if it minimizes the sum of edit costs for a given real-valued cost model $\Delta = (\Delta_\mathrm{match}, \Delta_\mathrm{subst}, \Delta_\mathrm{del}, \Delta_\mathrm{ins})$. Throughout this work, we assume that edit costs are non-negative—a pre-requisite for the correctness of A*. Further, we assume that $\Delta_\mathrm{match} \leq \Delta_\mathrm{subst}, \Delta_\mathrm{ins}, \Delta_\mathrm{del}$—a prerequisite for the correctness of our heuristic.

We note that our approach naturally works for cyclic reference graphs.

Alignment graph. In order to formalize optimal alignment as a shortest path finding problem, we rely on an *alignment graph* $G_\mathsf{a}^q = (V_\mathsf{a}^q, E_\mathsf{a}^q)$. Its nodes V_a^q are *states* of the form $\langle v, i \rangle$, where $v \in V_\mathsf{r}$ is a node in the reference graph and $i \in \{0, \ldots, |q|\}$ corresponds to a position in the read q. Its edges E_a^q are selected such that any path ${}^q_\mathsf{a}\pi$ in G_a^q from $\langle u, 0 \rangle$ to $\langle v, i \rangle$ corresponds to an alignment of the first i letters of q to G_r. Further, the edges are weighted, which allows us to define an *optimal alignment* of a read $q \in \Sigma^*$ as a shortest

$$
\begin{aligned}
&(\langle u,i\rangle,\langle v,i+1\rangle,q[i],\Delta_{\text{match}}) \in E_{\mathsf{a}}^{q} && \text{if } (u,v,\ell) \in E_{\mathsf{r}}, \ell = q[i] && \text{(match)}\\
&(\langle u,i\rangle,\langle v,i+1\rangle,q[i],\Delta_{\text{subst}}) \in E_{\mathsf{a}}^{q} && \text{if } (u,v,\ell) \in E_{\mathsf{r}}, \ell \neq q[i] && \text{(substitution)}\\
&(\langle u,i\rangle,\langle v,i\rangle,\varepsilon,\Delta_{\text{del}}) \in E_{\mathsf{a}}^{q} && \text{if } (u,v,\ell) \in E_{\mathsf{r}} && \text{(deletion)}\\
&(\langle u,i\rangle,\langle u,i+1\rangle,q[i],\Delta_{\text{ins}}) \in E_{\mathsf{a}}^{q} && && \text{(insertion),}
\end{aligned}
$$

FIGURE 3.7: Formal definition of alignment graph edges $E_{\mathsf{a}}^{q} \subseteq V_{\mathsf{a}}^{q} \times V_{\mathsf{a}}^{q} \times \Sigma_{\varepsilon} \times \mathbb{R}_{\geq 0}$. Here, $u,v \in V_{\mathsf{r}}$, $0 \leq i < |q|$, $\ell \in \Sigma$, and ε represents the empty string, indicating that letter ℓ was deleted.

path ${}^{q}_{\mathsf{a}}\pi$ in G_{a}^{q} from $\langle u,0\rangle$ to $\langle v,|q|\rangle$, for any $u,v \in V_{\mathsf{r}}$. Fig. 3.7 formally defines the edges E_{a}^{q}.

3.9.2 *A⋆ algorithm for finding a shortest path*

The A⋆ algorithm is a shortest path algorithm that generalizes Dijkstra's algorithm by directing the search towards the target. Given a weighted graph $G = (V,E)$, the A⋆ algorithm finds a shortest path from sources $S \subseteq V$ to targets $T \subseteq V$. To prioritize paths that lead to a target, it relies on an admissible heuristic function $h\colon V \to \mathbb{R}_{\geq 0}$, where $h(v)$ estimates the remaining length of the shortest path from a given node $v \in V$ to a target $t \in T$.

Algorithm. In a nutshell, the A⋆ algorithm maintains a set of *explored* nodes, initialized by all possible starting nodes S. It then iteratively *expands* the explored state v with lowest estimated total cost $f(v)$ by exploring all its neighbors. Here, $f(v) := g(v) + h(v)$, where $g(v)$ is the distance from $s \in S$ to v, and $h(v)$ is the estimated distance from v to $t \in T$. When the A⋆ algorithm expands a target node $t \in T$, it reconstructs the path leading to t and returns it. For completeness, §3.14.1 provides an implementation of A⋆.

Admissibility. The A⋆ algorithm is guaranteed to find a shortest path if its heuristic h provides a lower bound on the distance to the closest target, often referred to as h being *admissible* or optimistic.

Further, the performance of the A⋆ algorithm relies critically on the choice of h. Specifically, it is crucial to have low estimates for the optimal paths but also to have high estimates for suboptimal paths.

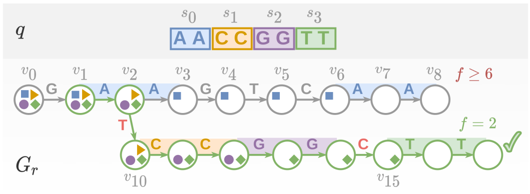


FIGURE 3.8: A toy overview example using the seed heuristic to align a read q to a reference graph G_r. The read is split into four colored seeds, where their corresponding crumbs are shown inside reference graph nodes as symbols with matching color. The optimal alignment is highlighted as a green path ending with a tick (✓) and includes one substitution (T→A) and one deletion (C).

Discussion. To summarize, we use the A* algorithm to find a shortest path from $\langle u, 0\rangle$ to $\langle v, |q|\rangle$ in G_a^q. To guarantee optimality, its heuristic function $h\langle v, i\rangle$ must provide a lower bound on the shortest distance from state $\langle v, i\rangle$ to a terminal state of the form $\langle w, |q|\rangle$. Equivalently, $h\langle v, i\rangle$ should lower bound the minimal cost of aligning $q[i{:}]$ to G_r starting from v, where $q[i{:}]$ denotes the suffix of q starting at position i (0-indexed). The key challenge is thus finding a heuristic that is not only admissible but also yields favorable performance.

3.10 SEED HEURISTIC

We instantiate the A* algorithm with a novel, domain-specific *seed heuristic* which allows to quickly align reads to a general reference graph. We first intuitively explain the seed heuristic and showcase it on a simple example (§3.10.1). Then, we formally define the heuristic and prove its admissibility (§3.10.2). Finally, we adapt our approach to rely on a trie, which leads to a critical speedup (§3.10.3).

3.10.1 *Overview*

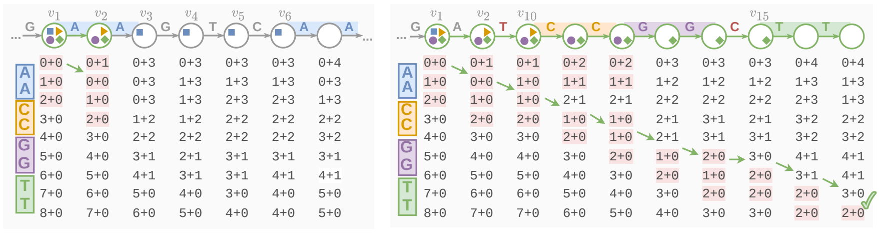
Fig. 3.8 showcases the seed heuristic on an overview example. It shows a read q to be aligned to a reference graph G_r. Our goal is to find an optimal alignment starting from an arbitrary node $v \in G_r$. For simplicity of the exposition, we assume unit edit costs $\Delta = (0, 1, 1, 1)$, which we generalize in §3.10.2.

Intuition. The A* search requires us to provide a lower bound of the remaining path cost from a state $\langle v, i \rangle$ to a target state. Clearly, to align the whole query, each of the remaining seeds (i.e. at or after position i in q) has to be eventually aligned. The intuition underlying the seed heuristic is to punish the state for the absence of any foreseeable match of each remaining seed. Notice that the order of the seeds is not directly taken into account.

In order to quickly check if a seed s can lead to a match, we follow a procedure similar to the one used by Hansel and Gretel who were placing breadcrumbs to find their trail back home. Before aligning a query, we will precompute all *crumbs* from all seeds so that not finding a crumb for a seed s on node v indicates that seed s could not be matched exactly before the query is fully aligned continuing from v. This way, assuming that a shortest path includes many seed matches, the crumbs will direct the A* search along with it.

If a crumb from an expected seed is missing in node v, its corresponding seed s could not possibly be aligned exactly and this will incur the cost of at least one substitution, insertion, or deletion. Assuming unit edit costs, $h\langle v, i \rangle$ yields a lower bound on the cost for aligning $q[i{:}]$ starting from v by simply returning the number of missing expected crumbs in v.

Crumbs precomputation example. Fig. 3.8 shows four seeds as colored sections of length 2 each, and represents their corresponding crumbs as ■, ▶, ● and ◆, respectively. Four crumbs are expected if we start at v_2, but ■ is missing, so $h\langle v_2, 0 \rangle = 1$. Analogously, if we reach v_2 after aligning one letter from the read, we expect 3 crumbs (except ■), and we find them all in v_2, so $h\langle v_2, 1 \rangle = 0$. To precompute the crumbs for each seed, we first find all positions in G_r from which the seed aligns exactly. Fig. 3.8 shows these exact matches as colored sections of G_r. Then, from each match we traverse G_r backwards and add crumbs to nodes that could lead to these matches. For example, because seed `CC` can be matched starting in node v_{10}, crumbs ▶ are placed on all nodes leading up to v_{10}. Similarly, seed `AA` has two exact matches, one starting in node v_0 and one starting in node v_6. However, we only add crumbs ■ to nodes v_0, v_1, and v_3–v_6, but not to node v_2. This is because v_2 is (i) strictly after the beginning of the match of `AA` at v_1 and (ii) too far before the match of `AA` at v_6. Specifically, any alignment starting from node v_2 and still matching `AA` at v_6 would induce an overall cost of 4 (it would require deleting the 4 letters A, G, T, and C). Even without a crumb ■ on v_2, our heuristic provides a lower bound on



	v_1	v_2	v_3	v_4	v_5	v_6	
	0+0	0+1	0+3	0+3	0+3	0+3	0+4
A	1+0	0+0	0+3	1+3	1+3	1+3	0+3
A	2+0	1+0	0+3	1+3	2+3	2+3	1+3
C	3+0	2+0	1+2	1+2	2+2	2+2	2+2
C	4+0	3+0	2+2	2+2	2+2	2+2	3+2
G	5+0	4+0	3+1	2+1	3+1	3+1	3+1
G	6+0	5+0	4+1	3+1	3+1	4+1	4+1
T	7+0	6+0	5+0	4+0	3+0	4+0	5+0
T	8+0	7+0	6+0	5+0	4+0	4+0	5+0

	v_1	v_2	v_{10}					v_{15}		
	0+0	0+1	0+1	0+2	0+2	0+3	0+3	0+3	0+4	0+4
A	1+0	0+0	1+0	1+1	1+1	1+2	1+2	1+2	1+3	1+3
A	2+0	1+0	1+0	2+1	2+1	2+2	2+2	2+2	2+3	1+3
C	3+0	2+0	2+0	1+0	1+0	2+1	3+1	2+1	3+2	2+2
C	4+0	3+0	3+0	2+0	1+0	2+1	3+1	3+1	3+2	3+2
G	5+0	4+0	4+0	3+0	2+0	1+0	2+0	3+0	4+1	4+1
G	6+0	5+0	5+0	4+0	3+0	2+0	1+0	2+0	3+1	4+1
T	7+0	6+0	6+0	5+0	4+0	3+0	2+0	2+0	2+0	3+0
T	8+0	7+0	7+0	6+0	5+0	4+0	3+0	3+0	2+0	2+0

FIGURE 3.9: Exploration of G_a^q, searching for a shortest path from the first to the last row using the seed heuristic. The table entry in the i^{th} row (zero indexed) below node v shows $g\langle v,i\rangle + h\langle v,i\rangle$, where $g\langle v,i\rangle$ is the shortest distance from any starting state $\langle u,0\rangle$ to $\langle v,i\rangle$. States that may[4] be expanded by the A* algorithm are highlighted in pink, and the rest of the states are shown for completeness even though they are never expanded. The shortest path corresponding to the best alignment is shown with green arrows ($\rightarrow$).

the cost of such an alignment: it never estimates a cost of more than 4, the number of seeds.

Guiding the search example. Fig. 3.9 demonstrates how $h\langle v,i\rangle$ guides the A* algorithm towards the shortest path by showing which states may be expanded when using the seed heuristic. Specifically, the unique optimal alignment in Fig. 3.8 starts from node v_1, continues to v_2, and then proceeds through node v_{10} (instead of v_3).

While the seed heuristic initially explores all states of the form $\langle v,0\rangle$ (we discuss in §3.10.3 how to avoid this by using a trie), it skips expanding any state that involves nodes v_3–v_8. This improvement is possible because all these explored states are penalized by the seed heuristic by at least 3, while the shortest path of cost 2 will be found before considering states on nodes v_3–v_8. Here, the heuristic function accurately predicts that expanding v_{10} may eventually lead to an exact alignment of seeds `CC`, `GG` and `TT`, while expanding v_3 may not lead to an alignment of either seed. In particular, the seed heuristic is not misled by the short-term benefit of correctly matching A in v_2, and instead provides a long-term recommendation based on the whole read. Thus, even though the walk to v_3 aligns exactly the first two letters of q, A* does not expand v_3 because the seed heuristic guarantees that the future cost will be at least 3.

4 Depending on how the A* algorithm handles tie-braking, different sets of states could be explored. For simplicity, we show all states that *could potentially* be explored.

3.10.2 *Formal definition*

Next, we formally define the seed heuristic function $h\langle v,i\rangle$. Overall, we want to ensure that $h\langle v,i\rangle$ is admissible, i.e., that it is a lower bound on the cost of a shortest path from $\langle v,i\rangle$ to some $\langle w,|q|\rangle$ in G_{a}^{q}.

Seeds. We split read $q \in \Sigma^*$ into a set $\mathcal{S}$ of non-overlapping seeds $s_0,\dots,s_{|\mathcal{S}|-1} \in \Sigma^*$. For simplicity, in this work we ensure that all seeds have the same length and are consecutive, i.e., we split q into substrings $s_0 \cdot s_1 \cdots s_{|\mathcal{S}|-1} \cdot t$, where all s_j are seeds of length k and we ignore the suffix t of q, which is shorter than k. We note that our approach could be trivially generalized to seeds of different lengths or non-consecutive seeds as long as they do not overlap. An interesting future work item is investigating how different choices of seeds affect the performance of our approach, and selecting seeds accordingly.

Matches. For each seed $s \in \mathcal{S}$, we locate all nodes $u \in M(s)$ in the reference graph that can be the start of an exact match of s:

$$M(s) := \{u \in V_{\mathsf{r}} \mid \exists \text{walk } \pi \text{ starting from } u \in G_{\mathsf{r}} \text{ and spelling } \sigma(\pi) = s\}.$$

To compute $M(s)$ efficiently, we leverage the trie introduced in §3.10.3.

Crumbs. For seed s_j starting at position i in q, we place crumbs on all nodes $u \in V_{\mathsf{r}}$ which can reach a node $v \in M(s_j)$ using less than $i + n_{\mathrm{del}}$ edges:

$$\begin{aligned} C(s) &:= \{u \in V_{\mathsf{r}} \mid \exists v \in M(s)\colon \mathit{dist}(u,v) < i + n_{\mathrm{del}}\}, \\ &\quad \text{where } \mathit{dist}(u,v) \text{ is the length of a shortest walk from } u \text{ to } v. \end{aligned}$$

Later in this section, we will select n_{del} to ensure that if an alignment uses more than n_{del} deletions, its cost must be so high that the heuristic function is trivially admissible.

To compute $C(s)$ efficiently, we can traverse the reference graph backwards from each $v \in M(s)$ by a backward breadth-first-search (BFS).

Heuristic. Let $\mathcal{S}_{\geq i}$ be the set of seeds that start at or after position i of the read, formally defined by $\mathcal{S}_{\geq i} := \{s_j \mid \lceil i/k \rceil \leq j < |\mathcal{S}|\}$. This allows us to define the number of expected but missing crumbs in state $\langle v,i\rangle$ as $\mathit{misses}\langle v,i\rangle := \big|\{s \in \mathcal{S}_{\geq i} \mid v \notin C(s)\}\big|$. Finally, we define the seed heuristic as

$$h\langle v,i\rangle = (|q| - i) \cdot \Delta_{\mathrm{match}} + \mathit{misses}\langle v,i\rangle \cdot \delta_{\mathit{min}}, \tag{3.3}$$

$$\text{for } \delta_{\mathit{min}} = \min(\Delta_{\mathrm{subst}} - \Delta_{\mathrm{match}}, \Delta_{\mathrm{del}}, \Delta_{\mathrm{ins}} - \Delta_{\mathrm{match}}), \tag{3.4}$$

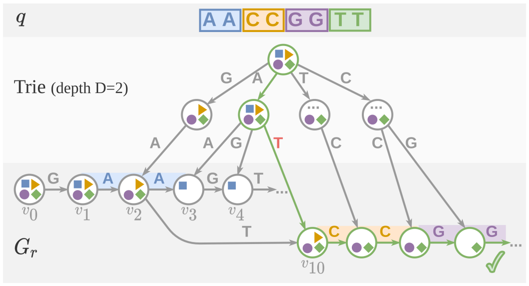


FIGURE 3.10: Reference graph from Fig. 3.8, extended by a trie of depth $D = 2$. For simplicity, the reverse-complement reference graph and parts marked by "..." are omitted.

Intuitively, Eq. (3.3) reflects that the cost of aligning each remaining letter from $q[i{:}]$ is at least Δ_{match}. In addition, every inexact alignment of a seed induces an additional cost of at least δ_{min}. Specifically, every substitution costs Δ_{subst} but requires one less match; every deletion costs Δ_{del}; and every insertion costs Δ_{ins} but also requires one less match.

We note that $h\langle v, i\rangle$ implicitly also depends on the reference graph G_{r}, the read q, the set of seeds, and the edit costs Δ.

In order for an alignment with at least n_{del} deletions to have a cost so high that the heuristic function is trivially admissible, we ensure $n_{\text{del}} \cdot \Delta_{\text{del}} \geq h\langle v, i\rangle$ by defining

$$n_{\text{del}} := \left\lceil \frac{|q| \cdot \Delta_{\text{match}} + |\mathcal{S}| \cdot \delta_{min}}{\Delta_{\text{del}}} \right\rceil . \tag{3.5}$$

In **??** 9, we show that $h\langle v, i\rangle$ is admissible, ensuring that our heuristic yields optimal alignments.

Theorem 9 (Admissibility)**.** *The seed heuristic $h\langle v, i\rangle$ is admissible.*

Proof. We provide a proof for **??** 9 in §3.14.2. □ □ □

3.10.3 *Trie index*

Considering all nodes $v \in V_{\text{r}}$ as possible starting points for the alignment means that the A* algorithm would explore all states of the form $\langle v, 0\rangle$, which immediately induces a high overhead of $|V_{\text{r}}|$. In line with previous works [23, 68], we avoid this overhead by complementing the reference graph with a trie index to produce a new graph $G_{\mathrm{r}}^{+} = (V_{\mathrm{r}}^{+}, E_{\mathrm{r}}^{+})$, where V_{r}^{+} is the union of the reference graph nodes V_{r} and the new trie vertices, and E_{r}^{+} is the union of E_{r}, the trie edges, and edges connecting the trie leafs with reference nodes. Note that constructing this trie index is a one-time pre-processing step that can be reused for multiple queries.

Since we want to also support aligning reverse-complement reads by starting from the trie *root*, we build the trie not only from the original reference graph and also from its reverse-complement.

Intuition. Fig. 3.10 extends the reference graph G_{r} from Fig. 3.8 with a trie. Here, any path in the reference graph uniquely corresponds to a path starting from the trie *root* (the top-most node in Fig. 3.10). Thus, in order to find an optimal alignment, it suffices to consider paths starting from the trie *root*, by using state $\langle root, 0\rangle$ as the only source for the A* algorithm. Note that if the reference graph branches frequently, the number of paths with length D may rise exponentially, leading to an exponential number of trie leaves. To counteract this exponential growth, we can select D logarithmically small, as $\log_4 N$.

For a more thorough introduction to the trie and its construction, see [23]. Importantly, our placement of crumbs (§3.10.2) generalizes directly to reference graphs extended with a trie (see also Fig. 3.10).

Reusing the trie to find seed matches. As a second usage of the trie, we can also exploit it to efficiently locate all matches $M(s)$ of a given seed s. In order to find all nodes where a seed match begins, we align (without errors) $\bar{s}$, the reverse-complement of s. To this end, we follow all paths spelling $\bar{s}$ starting from the *root*—the final nodes of these paths then correspond to nodes in $M(s)$. We ensure that the seed length $|s|$ is not shorter than the trie depth D, so that matching all letters in $\bar{s}$ ensures that we eventually transition from a trie leaf to the reference graph.

Optimization: skip crumbs on the trie. Generally, we aim to place as few crumbs as possible, in order to both reduce precomputation time and avoid misleading the A* algorithm by unnecessary crumbs. In the following, we introduce an optimization to avoid placing crumbs on trie nodes that are "too close" to the match of their corresponding seed so they cannot lead to an optimal alignment.

Specifically, when traversing the reference graph backwards to place crumbs for a match of seed s starting at node w, we may "climb" from a reference graph node u to a trie node u' backwards through an edge that otherwise leads from the trie to the reference. Assuming s starts at position

i in the read, we have already established that we can only consider nodes u that can reach w with less than $i + n_{\text{del}}$ edges (see §3.10.2). Here, we observe that it is sufficient to only climb into the trie from nodes u that can reach w using more than $i - n_{\text{ins}} - D$ edges, for

$$n_{\text{ins}} := \left\lceil \frac{|q| \cdot \Delta_{\text{match}} + |\mathcal{S}| \cdot \delta_{min}}{\Delta_{\text{ins}}} \right\rceil . \tag{3.6}$$

We define n_{ins} analogously to n_{del} to ensure that n_{ins} insertions will induce a cost that is always higher than $h\langle u, i\rangle$. We note that we can only avoid climbing into the trie if all paths from u to w are too short, in particular the longest one.

The following **??** 5 shows that this optimization preserves optimality.

Lemma 5 (Admissibility when skipping crumbs)**.** *The seed heuristic remains admissible when crumbs are skipped in the trie.*

Proof. We provide a proof for **??** 5 in §3.14.2. □ □ □

In order to efficiently identify all nodes u that can reach w by using more than $i - D - n_{\text{ins}}$ edges (among all nodes at a backward-distance at most $i + n_{\text{del}}$ from w), we use topological sorting: considering only nodes at a backward-distance at most $i + n_{\text{del}}$ from w, the length of a longest path from a node v to w is (i) ∞ if v lies on a cycle and (ii) computable from nodes closer to w otherwise.

3.11 EVALUATIONS

In the following, we demonstrate that our approach aligns faster than existing optimal aligners due to its superior scaling. Specifically, we address the following research questions:

Q1 What speedup can the seed heuristic achieve?
Q2 How does the seed heuristic scale with reference size?
Q3 How does the seed heuristic scale with read length?

The modes of operation which we analyze include both short (Illumina) and long (HiFi) reads to be aligned on on both linear and graph references.

3.11.1 *Seed heuristic implementation*

Both the seed heuristic and the prefix heuristic reuse the same free and open source C++ codebase of the AStarix aligner [23]. It includes a simple implementation of a graph and trie data structure which is not optimized for memory usage. In order to easily align reverse complement reads, the reverse complement of the graph is stored alongside its straight version. The shortest path algorithm only constructs explored states explicitly, so most states remain implicitly defined and do not cause computational burden.

Both heuristics benefit from a default optimization in AStarix called *greedy matching* [23, Section 4.2] which skips adding a state to the A* queue when only one edge is outgoing from a state and the upcoming read and reference letters match.

3.11.2 *Setting*

All experiments were executed on a commodity machine with an Intel Core i7-6700 CPU @ 3.40GHz processor, using a memory limit of 20 GB and a single thread. We note that while multiple tools support parallelization when aligning a single read, all tools can be trivially parallelized to align multiple reads in parallel.

Compared aligners. We compare the novel seed heuristic to prefix heuristic (both heuristics are implemented in AStarix), GraphAligner, PaSGAL, and Vargas. We provide the versions and commands used to run all aligners and read simulators in §3.14.3. We note that we do not compare to VG [47] and HGA [79] since the optimal alignment functionality of VG is meant for debugging purposes and has been shown to be inferior to other aligners [79, Tab. 4], and HGA makes use of parallelization and a GPU but has been shown to be superseded in the single CPU regime [79, Fig. 9]. PaSGAL and Vargas are compiled with AVX2 support. We execute the prefix heuristic with the default lookup depth $d = 5$.

Seed heuristic parameters. The choice of parameters for the seed heuristic influences its performance. Increasing the trie depth increases its number of nodes, but decreases the average out-degree of its leaves. We set the trie depth for all experiments to $D = 14 \approx \log_4 N$.

Shorter seeds are more likely to be matched perfectly by an optimal alignment, as they contain less letters that could be subject to edits. Thus, shorter seeds can tolerate higher error rate, but at the cost of slower precomputation due to a higher total number of matches, and slower alignment due to more off-track matches. In our experiments, we use seed lengths of $k = 25$ for Illumina reads and $k = 150$ for HiFi reads.

Data. We aligned reads to two different reference graphs: a linear *E. coli* genome (strain: K-12 substr. MG1655, ASM584v2) [72], with length of 4 641 652bp (approx. 4.7Mbp), and a variant graph with the Major Histocompatibility Complex (MHC), of size 5 318 019bp (approx. 5Mbp), taken from the evaluations of PaSGAL [63]. Additionally, we extracted a path from MHC in order to create a linear reference MHC-linear of length 4 956 648bp which covers approx. 93% of the original graph. Because of input graph format restrictions, we execute GraphAligner, Vargas and PaSGAL only on linear references in FASTA format (*E. coli* and the MHC-linear), while we execute the seed heuristic and the prefix heuristic on the original references (*E. coli* and MHC). This yields an underestimation of the speedup of the seed heuristic, as we expect the performance on MHC-linear to be strictly better than on the whole MHC graph.

To generate both short Illumina and long HiFi reads, we relied on two tools. We generated short single-end 200bp Illumina MSv3 reads using ART simulator [80]. We generated long HiFi reads using the script randomreads.sh[5] with sequencing lengths 5–25kbp and error rates 0.3%, which are typical for HiFi reads.

Edit costs. We execute AStarix with edit costs typical to the corresponding sequencing technology: $\Delta = (0, 1, 5, 5)$ for Illumina reads and $\Delta = (0, 1, 1, 1)$ for HiFi reads. As the performance of DP-based tools is independent of edit costs, we are using the respective default edits costs when executing GraphAligner, PaSGAL and Vargas.

Metrics. We report the performance of aligners in terms of runtime per one thousand aligned base pairs [s/kbp]. Since we measured runtime end-to-end (including loading reference graphs and reads from disk, and building the trie index for AStarix), we ensured that alignment time dominates the total runtime by providing sufficiently many reads to each aligner. In order to prevent excessive runtimes for slower tools, we used a different number of reads for each tool and explicitly report them for each experiment.

Since shortest path approaches skip considerable parts of the computation performed by aligners based on dynamic programming, the commonly used Giga Cell Updates Per Second (GCUPS) metric is not adequate for measuring performance in our context.

We measured used memory by `max_rss` (Maximum Resident Set Size) from Snakemake[6].

5 https://github.com/BioInfoTools/BBMap/blob/master/sh/randomreads.sh
6 https://snakemake.readthedocs.io/en/stable/

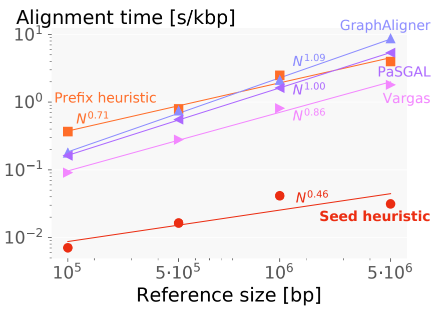
	Illumina		HiFi		
Tool	*E. coli*	MHC	*E. coli*	MHC	
Seeds heuristic	0.019	0.041	0.001	0.002	s/kbp
(this work)	2.4	2.6	2.4	1.7	GB (max use
	99.9996	99.9981	99.9989	99.9984	% skipped st
Prefix heuristic	269x	180x	n/a	n/a	x slowdown
	7.7	9.6	>20	>20	
	99.9501	99.9501	n/a	n/a	
GraphAligner	424x	212x	118x	64x	
	0.2	0.2	3.6	3.4	
Vargas	133x	67x	1 413x	705x	
	<0.1	<0.1	7.3	7.3	
PaSGAL	263x	130x	1 367x	736x	
	0.6	0.6	0.6	0.6	

Table 3.3: Runtime and memory comparison of optimal aligners. Simulated Illumina and HiFi reads are aligned to linear *E. coli* and graph MHC references. The runtime of the seed heuristic is expressed as absolute time per aligned kbp, while the other aligners are compared to the seed heuristic at a fold change. Additionally, the fraction of explored states is shown for the seed heuristic and the prefix heuristic.

We do not report accuracy or number of unaligned reads, as all evaluated tools align all reads with guaranteed optimality according to edit distance. We note that Vargas reports a warning that some of its alignments are not optimal—we ignore this warning and focus on its performance.

3.11.3 *Q1: Speedup of the seed heuristic*

Table 3.3 shows that the seed heuristic achieves a speedup of at least 60 times compared to all considered aligners, across all regimes of operation: both Illumina and HiFi reads aligned on *E. coli* and MHC references.



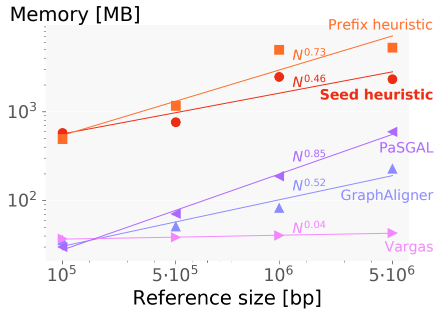


FIGURE 3.11: Performance degradation with reference size for **Illumina reads**. Log-log plots of total alignment time (left) and memory usage (right) show the scaling difference between aligners.

In the Illumina experiments, the seed heuristic is given 100k reads, while the other tools are given 1000 reads. In the HiFi experiments, the seed heuristic is given reads that cover the reference 10 times, and the other tools are given reads of coverage 0.1.

The key reason for the speedup of the seed heuristic is that on all four experiments, it skips $\geq 99.99\%$ of the Nm states computed by the DP approaches of GraphAligner, PaSGAL, and Vargas. This fraction accounts for both the explored states during the A* algorithm, and the number of crumbs added to nodes during precomputation for each read.

The prefix heuristic exceeded the available memory on HiFi reads, as it is not designed for long reads.

3.11.4 *Q2: Scaling with reference size*

In order to study the scaling of the aligners in terms of the reference size, we extracted prefixes of increasing length from MHC-linear. We then generated reads from each prefix, and ran all tools on all prefixes with the corresponding reads.

Illumina reads. Fig. 3.11 shows the runtime scaling and memory usage for Illumina reads. The seed heuristic was provided with 10k reads, while other tools were provided with 1k reads. The runtime of GraphAligner, PaSGAL and Vargas grow approximately linearly with the reference length, whereas the runtime of the seed heuristic grows roughly with $\sqrt[2]{N}$, where N is the reference size. Even on relatively small graphs like MHC, the

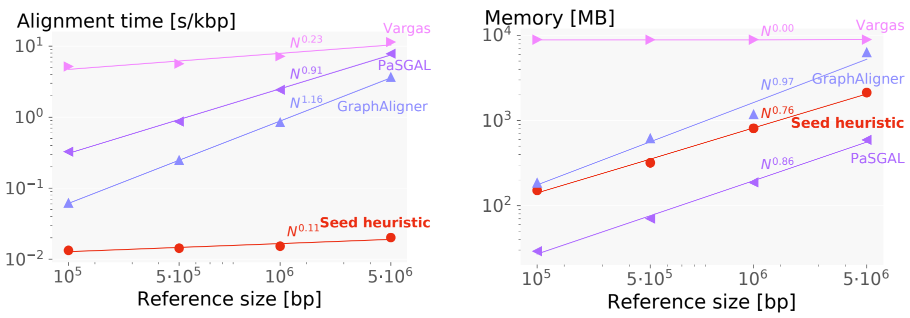


FIGURE 3.12: Performance degradation with reference size for **HiFi reads**. Log-log plots of total alignment time (left) and memory usage (right) show the scaling difference between aligners. Linear best fits correspond to polynomials of varying degree.

speedup of the seed heuristic reaches 200 times. Note that the scaling of the prefix heuristic is substantially worse than the seed heuristic since the 200bp reads are outside of its operational capabilities.

HiFi reads. Fig. 3.12 shows the runtime scaling and memory usage for HiFi reads. The respective total lengths of all aligned reads are 5Mbp for the seed heuristic, 500kbp for GRAPHALIGNER, and 100kbp for VARGAS and PASGAL. We do not show the prefix heuristic, since it explores too many states and runs out of memory. Crucially, we observe that the runtime of the seed heuristic is almost independent of the reference size, growing as $N^{0.11}$. We believe this improved trend compared to short reads is because the seed heuristic obtains better guidance on long reads, as it can leverage information from the whole read.

For both Illumina and HiFi reads, we observe near-linear scaling for PASGAL and GRAPHALIGNER as expected from the theoretical $O(Nm)$ runtime of the DP approaches. We conjecture that the runtime of VARGAS for long reads is dominated by the dependence from the read length, which is why on HiFi reads we observe better than linear runtime dependency on N but very large runtime. The current alignment bottleneck of ASTARIX-SEEDS is its memory usage, which is distributed between remembering crumbs and holding a queue of explored states.

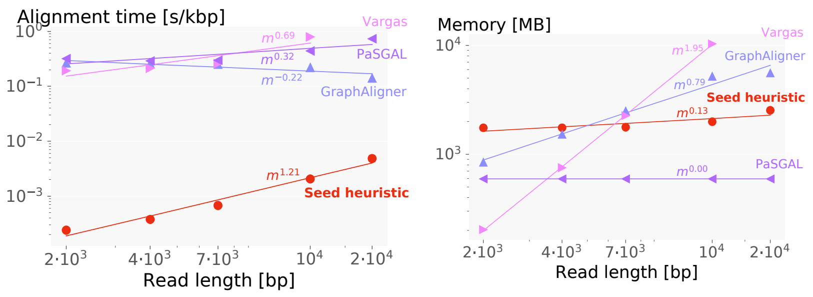


FIGURE 3.13: Performance degradation with HiFi read length. Log-log plots of total alignment time (left) and memory usage (right) show the scaling difference between aligners.

3.11.5 *Q3: Scaling with read length*

Fig. 3.13 shows the runtime and memory scaling with increasing length of aligned HiFi reads on MHC reference. Here we used reads with a total length of 100Mbp for the seed heuristic and 2Mbp for all other aligners.

The scaling of the seed heuristic in terms of read length is slightly worse than that of other aligners. However, this is compensated by its superior scaling in terms of reference size (see §3.11.4), leading to an overall better absolute runtime. We note that the memory usage of the seed heuristic does not heavily depend on the read length and for reads longer than 10kbp, it is superior to GRAPHALIGNER and VARGAS.

3.12 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for the valuable feedback. The first author is grateful for the Vipassana meditation which inspired the idea to punish paths for the absence of foreseeable seed matches. We also thank Paul Medvedev and Ragnar Groot Koerkamp for noting inaccuracies in the presentation of the algorithm.

3.13 CONCLUSION

We have presented an optimal read aligner based on the A* algorithm instantiated with a novel seed heuristic which guides the search by preferring crumbs on nodes that lead towards optimal alignments even for long reads.

The memory usage is currently limiting the application of AStarix for bigger references due to the size of the trie index. A remaining challenge is designing a heuristic function able to handle not only long but also noisier reads, such as the uncorrected PacBio reads that may reach 20% of mistakes. Possible improvements of the seed heuristic may include inexact matching of seeds, careful choice of seed positions, and accounting for the seeds order.

3.14 APPENDIX

Algorithm 4 A⋆ algorithm

```
 1: function A⋆($G$: Graph, $S$: Sources, $T$: Targets, $h$: Heuristic function)
 2:     $g \leftarrow Map$: (Nodes $\rightarrow \mathbb{R}_{\geq 0}$)     ▷ Shortest paths lengths to explored nodes
 3:     $f \leftarrow Map$: (Nodes $\rightarrow \mathbb{R}_{\geq 0}$)     ▷ $f(u) = g(u) + h(u)$
 4:     $Q \leftarrow MinPriorityQueue(priority = f)$     ▷ Priorities according to $f$
 5:     for all $s \in S$ do
 6:         $g[s] \leftarrow 0.0$, $f[s] \leftarrow 0.0$
 7:         $Q.push(s)$     ▷ Initially, explore all $s \in S$
 8:     while $Q \neq \emptyset$ do
 9:         $curr \leftarrow Q.pop()$     ▷ Get state with minimal $f$ to be expanded
10:         if $curr \in T$ then
11:             return BacktrackPath($curr$)     ▷ Reconstruct a walk to $curr$
12:         for all $(curr, next, cost) \in G.outgoingEdges(curr)$ do
13:             $g_{next} \leftarrow g[curr] + cost$
14:             $\hat{f}_{next} \leftarrow g_{next} + h(next)$     ▷ Candidate value for $f[next]$
15:             if $\hat{f}_{next} < f[next]$ then
16:                 $g[next] \leftarrow g_{next}$
17:                 $f[next] \leftarrow \hat{f}_{next}$
18:                 $Q.push(next)$     ▷ Explore state $next$
19:     assert $False$     ▷ Cannot happen if $T$ is reachable from $S$
```

3.14.1 *A⋆ Algorithm*

Algorithm 4 shows an implementation of the A⋆ algorithm, taken from [23]. We omit the implementation of BacktrackPath for simplicity.

3.14.2 *Proofs*

In the following, we provide proofs for **??** 9 and **??** 5, restated here for convenience.

Theorem 9 (Admissibility). *The seed heuristic $h\langle v,i\rangle$ is admissible.*

Proof. Let A be an optimal alignment of $q[i{:}]$ starting from $v \in G_{\mathrm{r}}$. We will prove that the cost of A is at least $h\langle v,i\rangle$.

If A contains at least n_{del} deletions, its cost is at least $n_{\mathrm{del}} \cdot \Delta_{\mathrm{del}}$, which is at least $|q| \cdot \Delta_{\mathrm{match}} + |\mathcal{S}| \cdot \delta_{min}$ by plugging in n_{del} from Eq. (3.5). This is an upper bound for $h\langle v,i\rangle$, which we observe after maximizing $h\langle v,i\rangle$ by substituting $i = 1$ and $misses = |\mathcal{S}|$ into Eq. (3.3), which concludes the proof in this case.

Otherwise, A contains less than n_{del} deletions. If we interpret A as a path in G_{a}^{q}, we first observe that A must spell $q[i{:}]$. Thus, A must in particular also contain all seeds $s_j \in \mathcal{S}_{\geq i}$ as substrings. We then split A into *subalignments* $A_{-1}, A_0, \ldots, A_p$, selected such that $A_0, \ldots, A_{p-1}$ spell the seeds $s_j \in \mathcal{S}_{\geq i}$, and A_{-1} and A_p spell the prefix and suffix of $q[i{:}]$ which do not cover any full seed.

This ensures that we can compute a lower bound on the cost of A as follows:

$$
\begin{aligned}
\mathrm{cost}(A) &= \sum_{j=-1}^{p} \mathrm{cost}(A_k) && (3.7)\\
&\geq \sum_{j=-1}^{p} |\sigma(A_k)| \cdot \Delta_{\mathrm{match}} + \sum_{j=0}^{p-1} \left\{ \begin{array}{ll} 0 & \text{if } v \in C\Big(s_{\lceil i/k \rceil + j}\Big) \\ \delta_{min} & \text{if } v \notin C\Big(s_{\lceil i/k \rceil + j}\Big) \end{array} \right\} && (3.8)\\
&= (|q| - i) \cdot \Delta_{\mathrm{match}} + \Big|\{v \notin C(s) \mid s \in \mathcal{S}_{\geq i}\}\Big| \cdot \delta_{min} && (3.9)\\
&= h\langle v,i\rangle && (3.10)
\end{aligned}
$$

Here, Eq. (3.7) follows from our decomposition of A. If we ignore the right-hand side in Eq. (3.8) (right of "+"), the inequality follows because matching all letters is the cheapest method to align any string. The right-hand side follows from a more precise analysis for subalignments A_k that spell a seed $s_{\lceil i/k \rceil + j}$ without a corresponding crumb in v. The absence of such a crumb indicates that no exact match of $s_{\lceil i/k \rceil + j}$ in G_{r} can be reached within less than $i + n_{\mathrm{del}}$ steps from v. However, because A contains less than n_{del} deletions, A_k must start within less than $i + n_{\mathrm{del}}$ steps from v. Thus, A_k does not align $s_{\lceil i/k \rceil + j}$ exactly, meaning that it introduces a cost of at least δ_{min}.

Eq. (3.9) follows from observing that $A_{-1}, \ldots, A_p$ have a total length of $|q| - i$, and observing that the right-hand sum adds up δ_{min} for every expected but missing crumb. Finally, Eq. (3.10) follows from our definition of $h\langle v, i\rangle$, concluding the proof. □ □ □

Lemma 5 (Admissibility when skipping crumbs). *The seed heuristic remains admissible when crumbs are skipped in the trie.*

Proof. Consider a reference graph with a match of seed s starting in node w. Now, consider a node v that cannot reach w using more than $i - D - n_{\text{ins}}$ edges. We can then show that a trie node v' with a path to v does not require a crumb for the match of s in node w.

Specifically, any path from *root* through nodes v' and v to node w has total length greater or equal $i - n_{\text{ins}}$. Thus, matching s at w requires at least n_{ins} insertions. Hence, the cost of such a path is at least $n_{\text{ins}} \cdot \Delta_{\text{ins}} = |q| \cdot \Delta_{\text{match}} + n \cdot \delta_{min}$. Observing that this is an upper bound for $h\langle v, i\rangle$ concludes the proof. □ □ □

3.14.3 *Versions, commands, parameters for running all evaluated approaches*

In the following, we provide details on how we executed the newest versions of the tools discussed in §3.11:

Executing AStarix.
Obtained from https://github.com/eth-sri/astarix

Seed heuristic

Command
```
astarix align-optimal -D 14 -a astar-seeds
-seeds_len l -f reads.fq -g graph.gfa >output
```

Prefix heuristic

Command
```
astarix align-optimal -D 14 -a astar-prefix -d 5 -f
reads.fq -g graph.gfa >output
```

For aligning Illumina reads, `astarix` is used with additional `-M 0 -S 1 -G 5` and for HiFi reads with `-M 0 -S 1 -G 1` which better match the error rate profiles for these technologies.

Executing other tools.

Vargas

Obtained from	https://github.com/langmead-lab/vargas (v0.2, commit `b1ad5d9`)
Command	`vargas align -g graph.gdef -U reads.fq -ete`
Comment	`-ete` stands for end to end alignment; default is 1 thread

PaSGAL

Obtained from	https://github.com/ParBLiSS/PaSGAL (commit `9948629`)
Command	`PaSGAL -q reads.fq -r graph.vg -m vg -o output -t 1`
Comment	Compiled with AVX2.

GraphAligner

Obtained from	https://github.com/maickrau/GraphAligner (v1.0.13, commit `02c8e26`)
Command	`GraphAligner -seeds-first-full-rows 64 -b 10000 -t 1 -f reads.fq -g graph.gfa -a alignments.gaf >output` (commit `9948629`)
Comment	`-seeds-first-full-rows` forces the search from all possible reference positions instead of using seeds; `-b 10000` sets a high alignment bandwidth; these two parameters are necessary for an optimal alignment according to the author and developer of the tool.

Simulating reads.

Illumina

```
art_illumina -ss MSv3 -sam -i graph.fasta -c N -l
200 -o dir -rnd_seed 42
```

HiFi

```
randomreads.sh -Xmx1g build=1 ow=t seed=1
ref=graph.fa illuminanames=t addslash=t pacbio=t
pbmin=0.003 pbmax=0.003 paired=f gaussianlength=t
minlength=5000 midlength=13000 maxlen=25000
out=reads.fq
```

Comment `BBMapcoverage`, https://github.com/BioInfoTools/BBMap/blob/master/sh/randomreads.sh (commit: a9ceda0)

3.14.4 *Notations*

Table 3.4 summarizes the notational conventions used in this work.

Object	Notation
Queries	$Q = \{q_i \mid q_i \in \Sigma^m\}$
Read	$q \in Q$
Length	$m := \lvert q \rvert \in \mathbb{N}$
Position in read	$q[i] \in \Sigma, i \in \{0, \ldots, m-1\}$
Reference graph	$G_\mathsf{r} = (V_\mathsf{r}, E_\mathsf{r})$
Size	$\lvert G_\mathsf{r} \rvert := \lvert V_\mathsf{r} \rvert + \lvert E_\mathsf{r} \rvert \in \mathbb{N}$
Nodes	$u, v \in V_\mathsf{r}$
Number of nodes	$N := \lvert V_\mathsf{r} \rvert \in \mathbb{N}$
Edges	$e \in E_\mathsf{r} := V_\mathsf{r} \times V_\mathsf{r} \times \Sigma$
Edge letter	$\ell \in \Sigma$
Reference graph with a trie	$G_\mathsf{r}^+ = (V_\mathsf{r}^+, E_\mathsf{r}^+)$
Trie depth	$D \in \mathbb{N}_{>0}$
Alignment graph	$G_\mathsf{a}^q = (V_\mathsf{a}^q, E_\mathsf{a}^q)$
State	$\langle u, i \rangle \in V_\mathsf{a}^q := V \times \{0, \ldots, m\}$
Edges	$(\langle u, i \rangle, \langle v, j \rangle, \ell, w) \in E_\mathsf{a}^q \subseteq V_\mathsf{a}^q \times V_\mathsf{a}^q \times \Sigma_\varepsilon \times \mathbb{R}_{\geq 0}, \Sigma$
Edge cost	$w \in \mathbb{R}_{\geq 0}$
Alignment	$\pi \in E_\mathsf{e}^*$ and $\sigma(\pi) = q$
Alignment cost	cost $\pi \in \mathbb{R}_{\geq 0}$
Seed heuristic	$h\langle u, i \rangle$
State	$\langle u, i \rangle$
Seed length	k
Maximum number of deletions	n_{del}
Maximum number of insertions	n_{ins}
In all graphs	$G = (V, E) \in \{G_\mathsf{r}, G_\mathsf{e}, G_\mathsf{a}^q\}$
Walk	$\pi \in G : \pi \in E^*$
Walk spelling	$\sigma(\pi) \in \Sigma^*$
Path	A walk without repeating nodes
A*	$A^\star(G, S, T, h)$
Graph	$G = (V, E)$
Nodes	$u, v \in V$
Edges	$e \in E \subseteq V \times V \times \mathbb{R}_{\geq 0}$
Source states	$S \subseteq V$
Target states	$T \subseteq V$
Heuristic function	$h \colon V \to \mathbb{R}_{\geq 0}$

4 SUMMARY

I dream my painting and I paint my dream.
— Vincent van Gogh

Summary here.

A
APPENDIX

Here be dragons.

BIBLIOGRAPHY

1. Medvedev, P. Theoretical analysis of edit distance algorithms: an applied perspective. *arXiv preprint:2204.09535* (2022).
2. Ivanov, P., Bichsel, B. & Vechev, M. *Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds* in *RECOMB 2022* (2022).
3. Needleman, S. B. & Wunsch, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology* **48**, 443 (1970).
4. Navarro, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR)* **33**, 31 (2001).
5. Prjibelski, A. D., Korobeynikov, A. I. & Lapidus, A. L. in *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics* 292 (2018).
6. Backurs, A. & Indyk, P. *Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)* in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing* (2015), 51.
7. Kucherov, G. Evolution of biosequence search algorithms: a brief survey. *Bioinformatics* **35**, 3547 (2019).
8. Vintsyuk, T. K. Speech discrimination by dynamic programming. *Cybernetics* **4**, 52 (1968).
9. Sankoff, D. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences* **69**, 4 (1972).
10. Sellers, P. H. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics* **26**, 787 (1974).
11. Wagner, R. A. & Fischer, M. J. The string-to-string correction problem. *Journal of the ACM (JACM)* **21**, 168 (1974).
12. Reinert, K., Dadi, T. H., Ehrhardt, M., Hauswedell, H., Mehringer, S., Rahn, R., Kim, J., Pockrandt, C., Winkler, J., Siragusa, E., *et al.* The SeqAn C++ template library for efficient sequence analysis: a resource for programmers. *Journal of biotechnology* **261**, 157 (2017).
13. Daily, J. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics* **17**, 1 (2016).

14. Ukkonen, E. Algorithms for approximate string matching. *Information and control* **64**, 100 (1985).

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

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

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

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

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

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

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

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

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

24. Benson, G., Levy, A. & Shalom, R. *Longest Common Subsequence in k-length substrings* 2014.

25. Levenshtein, V. I. *et al. Binary codes capable of correcting deletions, insertions, and reversals* in *Soviet physics doklady* **10** (1966), 707.

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

27. Benson, G., Levy, A., Maimoni, S., Noifeld, D. & Shalom, B. R. LCSk: a refined similarity measure. *Theoretical Computer Science* **638**, 11 (2016).

28. Poole, D. L. & Mackworth, A. K. *Artificial Intelligence: Foundations of Computational Agents* Second (Cambridge University Press, 2017).

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

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

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

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

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

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

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

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

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

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

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

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

41. 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.* Sequencing of human genomes with nanopore technology. *Nature communications* **10**, 1 (2019).

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

43. Stevenson, K. R., Coolon, J. D. & Wittkopp, P. J. Sources of bias in measures of allele-specific expression derived from RNA-seq data aligned to a single reference genome. *BMC Genomics* (2013).

44. Brandt, D. Y. C., Aguiar, V. R. C., Bitarello, B. D., Nunes, K., Goudet, J. & Meyer, D. Mapping Bias Overestimates Reference Allele Frequencies at the HLA Genes in the 1000 Genomes Project Phase I Data. eng. *G3 (Bethesda, Md.)* (2015).

45. Dilthey, A., Cox, C., Iqbal, Z., Nelson, M. R. & McVean, G. Improved genome inference in the MHC using a population reference graph. *Nature Genetics* (2015).

46. Paten, B., Novak, A. M., Eizenga, J. M. & Garrison, E. Genome graphs and the evolution of genome inference. *Genome Research* (2017).

47. Garrison, E., Sirén, J., Novak, A. M., Hickey, G., Eizenga, J. M., Dawson, E. T., Jones, W., Garg, S., Markello, C., Lin, M. F., Paten, B. & Durbin, R. Variation graph toolkit improves read mapping by representing genetic variation in the reference. *Nature Biotechnology* (2018).

48. Altschul, S. F., Gish, W., Miller, W., Myers, E. W. & Lipman, D. J. Basic local alignment search tool. eng. *Journal of Molecular Biology* (1990).

49. Langmead, B. & Salzberg, S. L. Fast gapped-read alignment with Bowtie 2. *Nature Methods* (2012).

50. Harismendy, O., Schwab, R. B., Bao, L., Olson, J., Rozenzhak, S., Kotsopoulos, S. K., Pond, S., Crain, B., Chee, M. S., Messer, K., Link, D. R. & Frazer, K. A. Detection of low prevalence somatic mutations in solid tumors with ultra-deep targeted sequencing. eng. *Genome Biology* (2011).

51. Salmela, L. & Rivals, E. LoRDEC: accurate and efficient long read error correction. eng. *Bioinformatics (Oxford, England)* (2014).

52. Buhler, S. & Sanchez-Mazas, A. HLA DNA sequence variation among human populations: molecular signatures of demographic and selective events. eng. *PloS One* (2011).

53. Antipov, D., Korobeynikov, A., McLean, J. S. & Pevzner, P. A. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. eng. *Bioinformatics (Oxford, England)* (2016).

54. Jain, C., Zhang, H., Gao, Y. & Aluru, S. *On the Complexity of Sequence to Graph Alignment* in *Research in Computational Molecular Biology* (Cham, 2019).

55. Li, H. & Durbin, R. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics (Oxford, England)* (2009).

56. Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O. & Weigel, D. Simultaneous alignment of short reads against multiple genomes. eng. *Genome Biology* (2009).

57. Jean, G., Kahles, A., Sreedharan, V. T., De Bona, F. & Rätsch, G. RNA-Seq read alignments with PALMapper. eng. *Current Protocols in Bioinformatics* (2010).

58. Sirén, J., Välimäki, N. & Mäkinen, V. Indexing Graphs for Path Queries with Applications in Genome Research. eng. *IEEE/ACM transactions on computational biology and bioinformatics (TCBB)* (2014).

59. Kim, D., Paggi, J. M., Park, C., Bennett, C. & Salzberg, S. L. Graph-based genome alignment and genotyping with HISAT2 and HISAT-genotype. *Nature Biotechnology* (2019).

60. Sirén, J. in *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (2017).

61. Heydari, M., Miclotte, G., Van de Peer, Y. & Fostier, J. BrownieAligner: accurate alignment of Illumina sequencing data to de Bruijn graphs. *BMC Bioinformatics* (2018).

62. Smith, T. F. & Waterman, M. S. Comparison of biosequences. *Advances in Applied Mathematics* (1981).

63. Jain, C., Misra, S., Zhang, H., Dilthey, A. & Aluru, S. *Accelerating Sequence Alignment to Graphs* in *International Parallel and Distributed Processing Symposium (IPDPS)* ISSN: 1530-2075 (2019).

64. Rautiainen, M., Mäkinen, V. & Marschall, T. Bit-parallel sequence-to-graph alignment. *Bioinformatics* (2019).

65. Liu, B., Guo, H., Brudno, M. & Wang, Y. deBGA: read alignment with de Bruijn graph-based seed and extension. eng. *Bioinformatics (Oxford, England)* (2016).

66. Limasset, A., Flot, J.-F. & Peterlongo, P. Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. *Bioinformatics*. btz102 (2019).

67. Kavya, V. N. S., Tayal, K., Srinivasan, R. & Sivadasan, N. Sequence Alignment on Directed Graphs. eng. *Journal of Computational Biology* (2019).

68. Dox, G. & Fostier, J. *Efficient algorithms for pairwise sequence alignment on graphs* MA thesis (Ghent university, 2018).

69. Rautiainen, M. & Marschall, T. *Aligning sequences to general graphs in O(V+mE) time* preprint (2017).

70. Dechter, R. & Pearl, J. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* (1985).

71. Sellers, P. H. An algorithm for the distance between two finite sequences. *Journal of Combinatorial Theory* (1974).

72. Howe, K. L., Contreras-Moreira, B., De Silva, N., Maslen, G., Akanni, W., Allen, J., Alvarez-Jarreta, J., Barba, M., Bolser, D. M., Cambell, L., *et al.* Ensembl Genomes 2020–enabling non-vertebrate genomic research. *Nucleic Acids Research* (2020).

73. Huang, W., Li, L., Myers, J. R. & Marth, G. T. ART: a next-generation sequencing read simulator. eng. *Bioinformatics (Oxford, England)* (2012).

74. Holtgrewe, M. Mason – A Read Simulator for Second Generation Sequencing Data. *Tech. Report FU Berlin* (2010).

75. Pearl, J. On the Discovery and Generation of Certain Heuristics. *AI Mag.* (1983).

76. Köster, J. & Rahmann, S. Snakemake–a scalable bioinformatics workflow engine. eng. *Bioinformatics (Oxford, England)* (2012).

77. Equi, M., Grossi, R., Mäkinen, V., Tomescu, A., *et al. On the complexity of string matching for graphs* in *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)* (2019).

78. Darby, C. A., Gaddipati, R., Schatz, M. C. & Langmead, B. Vargas: heuristic-free alignment for assessing linear and graph read aligners. *Bioinformatics* **36**, 3712 (2020).

79. Feng, Z. & Luo, Q. *Accelerating Sequence-to-Graph Alignment on Heterogeneous Processors* in *50th International Conference on Parallel Processing* (2021), 1.

80. Huang, W., Li, L., Myers, J. R. & Marth, G. T. ART: a next-generation sequencing read simulator. *Bioinformatics* **28**, 593 (2011).

CURRICULUM VITAE

PERSONAL DATA

Name	Pesho Ivanov
Date of Birth	May 29, 1989
Place of Birth	Shumen, Bulgaria
Citizen of	Bulgaria

EDUCATION

1896 – 1900	Eidgenössisches Polytechnikum, Zürich, Switzerland *Final degree:* Diploma
1895 – 1896	Aargauische Kantonsschule (grammar school) Aarau, Switzerland *Final degree:* Matura (university entrance diploma)
– July 1894	Luitpold-Gymnasium (grammar school) Munich, Germany

EMPLOYMENT

June 1902 –	Technical Expert, III Class *Federal Office for Intellectual Property*, Bern, Switzerland

PUBLICATIONS

1. He, J., Ivanov, P., Tsankov, P., Raychev, V. & Vechev, M. *Debin: Predicting debug information in stripped binaries* in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), 1667.
2. Ivanov, P., Bichsel, B., Mustafa, H., Kahles, A., Rätsch, G. & Vechev, M. T. *AStarix: Fast and Optimal Sequence-to-Graph Alignment* in *RECOMB 2020* (2020).
3. Ivanov, P., Bichsel, B. & Vechev, M. *Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds* in *RECOMB 2022* (2022).
4. Groot Koerkamp, R. & Ivanov, P. Exact global alignment using A* with seed heuristic and match pruning. *bioRxiv* (2022).