ASTARIX: Fast and Optimal Sequence-to-Graph Alignment

Pesho Ivanov, Benjamin Bichsel, Harun Mustafa,
André Kahles, Gunnar Rätsch, and Martin Vechev

Department of Computer Science,
ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Abstract. 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

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 [35,4]. 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 [7,25,9].

Both for sequence-to-sequence alignment and sequence-to-graph alignment, heuristics are employed to keep alignment tractable [2,21,9], 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) [7],

in complex but rare genotypes arising from somatic-subclones in tumor sequencing data [10], or in the presence of frequent sequencing errors [29]. 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 [5] 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 [3,16].

1.1 Related Work

Seed-and-Extend. Since optimal alignment is often intractable, many aligners use heuristics, most commonly the *seed-and-extend* paradigm [2,21,22]. 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 [30,17]. 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 [33] to index walks in an augmented reference sequence, forming a local genome graph [19]. VG [9] uses a similar technique [32] 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 [11].

Optimal Alignment. Current optimal sequence-to-graph alignment algorithms reach their worst-case $\mathcal{O}(nm)$ runtime [16]. 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 [34] considering all possible starting nodes. These use modern SIMD instructions (e.g. VG [9] and PaSGAL [15]) or reformulations of edit distance computation to allow for bit-parallel

computations in GraphAligner[1] [27]. Many of these, however, are designed only for specific types of genome graphs, such as *de Bruijn* graphs [24,11,23] and variation graphs [9]. 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 [18,9].

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 [8] 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.

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 difficulty 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* with an accurate domain-specific heuristic that accounts for the upcoming query letters to be aligned (§3).
2. **Algorithmic optimizations.** To ensure that AStarix is practical, we introduce a number of algorithmic optimizations which increase performance and decrease memory footprint (§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 (§5).

[1] We refer as BitParallel to to the bit-parallel DP algorithm implemented in GraphAligner tool [27].

[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

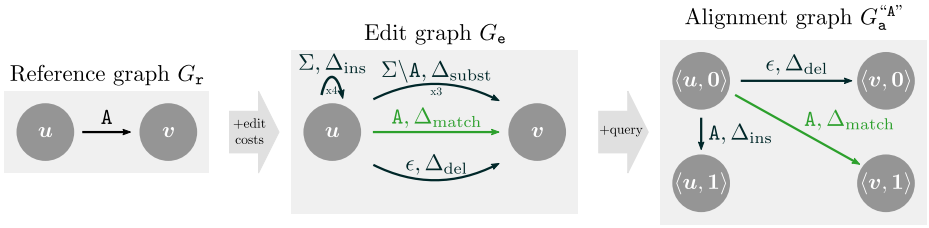

Fig. 1: Starting from the reference graph (left), we can construct the edit graph (middle) and the alignment graph $G_\mathtt{a}^q$ for query q = “A” (right). Edges are annotated with labels and/or costs, where sets of labels represent multiple edges, one for each letter in the set (indicated by “x3” and “x4”).

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. 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 [25,9]. We formalize a reference graph as a tuple $G_\mathtt{r} = (V_\mathtt{r}, E_\mathtt{r})$ of nodes $V_\mathtt{r}$ and directed, labeled edges $E_\mathtt{r} \subseteq V_\mathtt{r} \times V_\mathtt{r} \times \Sigma$, where the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ represents the four different nucleotides. Note that in contrast to sequence graphs [28], we label edges instead of nodes.

Path, Spelling. Any path $\pi = (e_1, \ldots, e_k)$ in $G_\mathtt{r}$ 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 [27] and BrownieAligner [11] but missing from VG [9], PaSGAL [15] and V-ALIGN [18].

Alignment on Reference Graph. An *alignment* of *query* $q \in \Sigma^*$ to a reference graph $G_\mathtt{r} = (V_\mathtt{r}, E_\mathtt{r})$ consists of (i) a path π in $G_\mathtt{r}$ 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 $0 \leq \Delta_{\text{match}} \leq \Delta_{\text{subst}}, \Delta_{\text{ins}}, \Delta_{\text{del}}$, 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* $G_{\mathtt{e}} := (V_{\mathtt{e}}, E_{\mathtt{e}})$ has directed, labeled edges $E_{\mathtt{e}} \subseteq V_{\mathtt{e}} \times V_{\mathtt{e}} \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ 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. 1). The edit and reference graphs consist of the same vertices, i.e., $V_{\mathtt{e}} = V_{\mathtt{r}}$. However, $E_{\mathtt{e}}$ contains more edges than $E_{\mathtt{r}}$ to account for edits. Concretely, for each edge $(u, v, \ell) \in E_{\mathtt{r}}$, $E_{\mathtt{e}}$ 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 $\pi \in G_{\mathtt{e}}$ is defined analogously to reference graphs, except that deleted letters (represented by ϵ) are ignored. The cost $\text{cost}(\pi)$ of a path $\pi \in G_{\mathtt{e}}$ is the sum of all its edge costs.

Alignment on Edit Graph. An *alignment* of query q to $G_{\mathtt{r}}$ is a path π in $G_{\mathtt{e}}$ 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 $G_{\mathtt{e}}$ 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* $G^q_{\mathtt{a}} = (V^q_{\mathtt{a}}, E^q_{\mathtt{a}})$.

Here, each *state* $\langle v, i \rangle \in V^q_{\mathtt{a}}$ consists of a vertex $v \in V_{\mathtt{e}}$ and a query position $i \in \{0, \ldots, |q|\}$ (equivalent to [28]). Traversing a state $\langle v, i \rangle \in V^q_{\mathtt{a}}$ 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 $G^{\text{“}\mathtt{A}\text{”}}_{\mathtt{a}}$ in Fig. 1 lacks substitution edges from $G_{\mathtt{e}}$, as their labels (`C`, `G`, `T`) do not match the query $q =$ “A”.

We construct the alignment graph $G^q_{\mathtt{a}}$ 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 $G_{\mathtt{r}}$. As a consequence, there is a one-to-one correspondence between alignments $\pi_{\mathtt{e}}$ of q to $G_{\mathtt{e}}$ and paths $\pi^q_{\mathtt{a}} \in G^q_{\mathtt{a}}$ from sources $S := V_{\mathtt{r}} \times \{0\}$ to targets $T := V_{\mathtt{r}} \times \{|q|\}$, with $\text{cost}(\pi_{\mathtt{r}}) = \text{cost}(\pi^q_{\mathtt{a}})$. To find the best alignment in $G_{\mathtt{e}}$, only paths in $G^q_{\mathtt{a}}$ (walks without repeating nodes) can be considered, since repeating a node in $G^q_{\mathtt{a}}$ cannot lead to a lower cost ($\Delta_{\text{del}} \geq 0$) for the same state.

The edges $E^q_{\mathtt{a}} \subseteq V^q_{\mathtt{a}} \times V^q_{\mathtt{a}} \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ are built based on the edges in $E_{\mathtt{e}}$, 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:

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

$$(u, v, \epsilon, w) \in E_{\mathtt{e}} \implies (\langle u, i \rangle, \langle v, i \quad \rangle, \epsilon, w) \in E^q_{\mathtt{a}} \quad \text{for } 0 \leq i < |q| \tag{2}$$

Here, assuming 0-indexing, $q[i]$ is the next letter to be matched after matching i letters. Then, Eq. (1) represents matches, substitutions, and insertions (which

Algorithm 1 AStarix including heuristic function.

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

function AStarix(q: Query)
    G_a^q ← DefineAlignmentGraph(G_r, q)                          ▷ Following §2
    S ← {⟨v, i⟩ ∈ V_a^q | i = 0}                                  ▷ Sources: no letter matched
    T ← {⟨v, i⟩ ∈ V_a^q | i = |q|}                                ▷ Targets: all letters matched
    return A*(G_a^q, S, T, Heuristic)                             ▷ A* provided in App. A.1

function Heuristic(⟨u, i⟩: State)                                 ▷ Heuristic: Cost of upcoming sequence
    d' ← min(d, |q| − i)                                          ▷ Actual length of upcoming sequence
    s ← q[i : i + d']                                             ▷ Upcoming sequence (next d letters after current)
    return h(u, s)                                                ▷ Cost of aligning s to G_e starting from u

function h(u, s)                                                  ▷ Cost of aligning s starting from u
    return RecursiveAlign(u, s, 0.0, ∞)                           ▷ Simple branch-and-bound
```

advance the position in the query by 1), while Eq. (2) represents deletions (which do not advance the position in the query).

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

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.

3.1 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 App. A.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 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* indeed finds the shortest path, h should be *optimistic*:

Definition 1 (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* finds optimal alignments [6, Res. 3], 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* [6, Res. 6]. Thus, a key contribution of ours is a domain-specific heuristic h.

3.2 AStarix: Instantiating A*

Algorithm 1 shows an unoptimized version of AStarix and its heuristic function. AStarix expects a reference graph (Line 1) and a query (Line 3) as input, and returns an optimal alignment (Line 7) by searching for a shortest path from S to T in the alignment graph G^q_{a}. It is parameterized by hyper-parameters (d in Line 2, more in §4) and edit costs (implicitly provided).

The function Heuristic (Lines 8–11) 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 App. A.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 1. *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. 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. 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 $g+h \leq 1$.

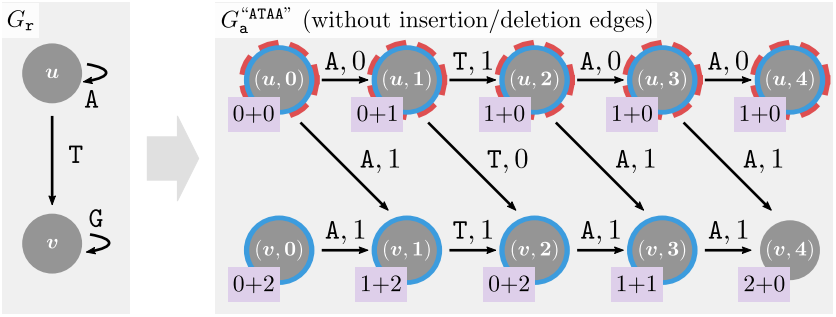

Fig. 2: The benefit of using our heuristic over DIJKSTRA. Alignment graph $G_{\mathtt{a}}^{\text{"ATAA"}}$ (right) is based on reference graph $G_{\mathtt{r}}$ (left), but omits insertion and deletion edges for simplicity. The pink boxes $g+h$ indicate the distance from the sources $S = \{\langle u, 0\rangle, \langle v, 0\rangle\}$ (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).

Memoization. Recall that the return value of h in Line 8 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 $\mathcal{O}(1)$ time, if it was computed for a previous query.

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).

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 $u \in G_{\mathtt{r}}$ as possible starting nodes. Thus, optimal aligners PASGAL [15] and BITPARALLEL [27] brute-force through all possible starting nodes $u \in G_{\mathtt{r}}$.

To more efficiently handle arbitrary starting positions for alignments, we extend the reference graph with a trie (referred to as *suffix tree* in [8]) 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 $G_{\mathtt{r}}$ that correspond to the same prefix. In the following, we formalize this approach.

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

Fig. 3 shows an example trie. To construct it, we first associate with every node $v \in V_{\mathtt{r}}$ the set $\mathcal{S}_v$ of its D-mers (orange boxes in Fig. 3): 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.

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

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

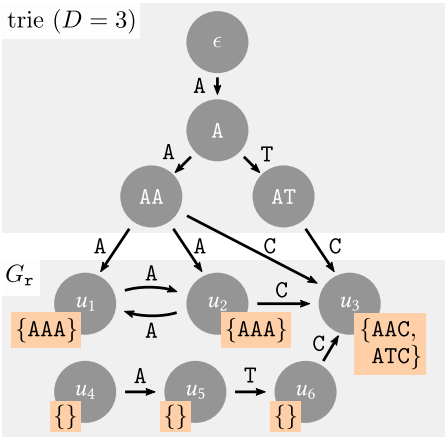

Fig. 3: $G^+_{\mathtt{r}}$ enables semi-global alignment by extending $G_{\mathtt{r}}$ with a trie.

Third, we add edges within the trie, which ensure that paths from ϵ to any trie node s spell s. Formally, whenever $s{\cdot}\ell \in V^+_{\mathtt{r}}$, we add an edge $(s, s{\cdot}\ell, \ell)$ to $E^+_{\mathtt{r}}$, 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 $v \in V_{\mathtt{r}}$ can be spelled by a walk from ϵ to v. Formally, if $s{\cdot}\ell \in \mathcal{S}_v$, then $(s, v, \ell) \in E^+_{\mathtt{r}}$.

Importantly, extending $G_{\mathtt{r}}$ to $G^+_{\mathtt{r}}$ is compatible with the construction of the edit graph $G_{\mathtt{e}}$, the construction of the alignment graph and all other optimizations. In particular, when searching for a shortest path in the alignment graph constructed from $G^+_{\mathtt{r}}$, 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 $s{\cdot}\ell \in V^+_{\mathtt{r}}$ with a unique outgoing edge $(s{\cdot}\ell, v, \ell')$ to a reference graph node $v \in V_{\mathtt{r}}$. To compensate for removing node $s \cdot \ell$, we introduce a new edge (s, u, ℓ) to a node $u \in V_{\mathtt{r}}$ with an edge (u, v, ℓ') (such a node must exist according to the construction of $G^+_{\mathtt{r}}$). For example, in Fig. 3, we (i) remove node `AT` including its edges $(\mathtt{A}, \mathtt{AT}, \mathtt{T})$ and $(\mathtt{AT}, u_3, \mathtt{C})$, but (ii) introduce an edge $(\mathtt{A}, u_2, T)$.

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').

4.2 Greedy Match Optimization

We also employ an optimization originally developed for computing the edit distance between two strings [31,1], but which has also been used in the context of string to graph alignment [8]. We omit the correctness proof of this optimization, which is already covered in [31], and only explain the intuition behind it.

Suppose there is only one outgoing edge $e = (u, v, \ell) \in E_{\mathtt{r}}$ from a node $u \in V_{\mathtt{r}}$. 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 (§4.1).

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 App. A.3.

Theorem 2. *h_c is optimistic.*

Proof. We have $h_c(u, s) \leq h(u, s)$ and that $h(u, s)$ is optimistic (Theorem 1). □

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 $R_u \subseteq V_{\mathtt{e}}$ at distance at most d from u (cp. Line 2 in Algorithm 1): $R_u := \{v \in V_{\mathtt{r}} \mid \exists \text{ path } \pi \in G_{\mathtt{e}} \text{ from } u \text{ to } v \text{ with } |\pi| \leq d\}$.

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

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

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

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

Theorem 3. *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 $h_d(u, s) \leq h(u, s)$. □

4.4 Partitioning Nodes into Equivalence Classes

We have shown in §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 $r \in V_{\mathtt{r}}$ 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 $\mathcal{O}(1)$, we define a helper array *repr* with $repr[u] = r$.

Identifying Equivalence Classes. To identify the nodes belonging to the same equivalence class, we assume the optimization from §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.

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.

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 $G_{\mathtt{r}}$, 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 (§3.2), 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 §4.3 and §4.4 which are applicable only to AStarix.

While the optimality of AStarix is not affected by its parameters, its performance is (see App. A.3 for analysis). To compare with other aligners, we use values $d = 5$, $c = 5$, $D = \lfloor \log_{\Sigma} |G_{\mathtt{r}}| \rfloor$.

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

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 [15].

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 [15], 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 $\Delta = (0, 1, 1, 1)$.

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 1). The first is a linear graph without variation based on the *E. coli* reference genome (strain: K-12 substr. MG1655, ASM584v2 [13]). The other two are variation graphs taken from the PaSGAL evaluations [15]: 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 [14] 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 [15] using the Mason2 [12] 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* performance [26], 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 [20] pipeline framework to measure the execution time of every aligner (including the time 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.

Table 1: Performance of optimal aligners for different reference graphs.

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

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.

5.4 Comparison of Optimal Aligners

Different Reference Graphs. Table 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. 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 $\mathcal{O}(|G^q_{\mathtt{a}}|) = \mathcal{O}(m{\cdot}G_{\mathtt{r}})$. 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.

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.

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 §3.1 and the example in Fig. 2) between AStarix and Dijkstra on the MHC1 dataset.

Fig. 5 demonstrates the benefit of the heuristic function in terms of both alignment time and number of explored states. Most importantly, AStarix

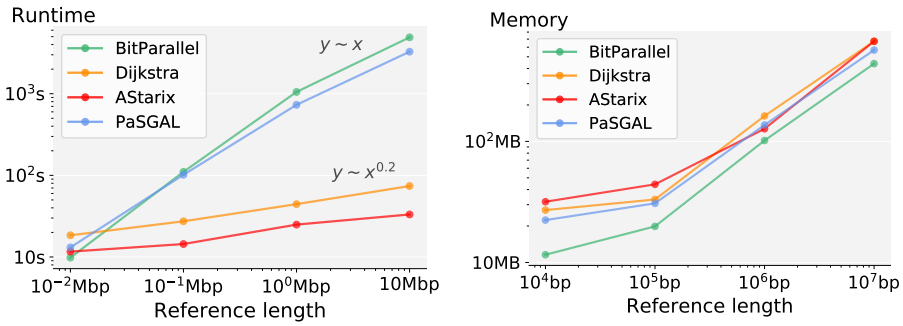

Fig. 4: Comparison of overall runtime and memory usage of optimal aligners with increasing prefixes of E. coli as references.

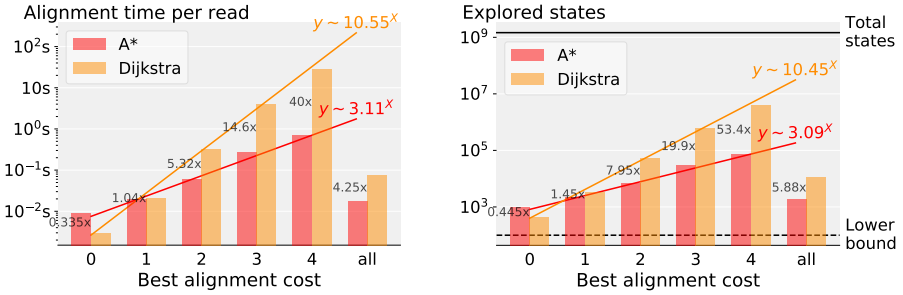

Fig. 5: Comparison of A* and Dijkstra in terms of mean alignment runtime per read and mean explored states depending on the best alignment cost on MHC1.

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. 5 denotes the total number of states $|G_{\mathtt{r}}| \cdot |q|$, 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.

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.

References

1. Allison, L.: Lazy dynamic-programming can be eager. Information Processing Letters (1992)
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology (1990)
3. Antipov, D., Korobeynikov, A., McLean, J.S., Pevzner, P.A.: hybridSPAdes: an algorithm for hybrid assembly of short and long reads. Bioinformatics (Oxford, England) (2016)
4. 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. G3 (Bethesda, Md.) (2015)
5. Buhler, S., Sanchez-Mazas, A.: HLA DNA sequence variation among human populations: molecular signatures of demographic and selective events. PloS One (2011)
6. Dechter, R., Pearl, J.: Generalized best-first search strategies and the optimality of A*. Journal of the ACM (1985)
7. 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)
8. Dox, G., Fostier, J.: Efficient algorithms for pairwise sequence alignment on graphs. Master's thesis, Ghent university (2018)
9. 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)
10. 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. Genome Biology (2011)
11. Heydari, M., Miclotte, G., Van de Peer, Y., Fostier, J.: BrownieAligner: accurate alignment of Illumina sequencing data to de Bruijn graphs. BMC Bioinformatics (2018)
12. Holtgrewe, M.: Mason – A Read Simulator for Second Generation Sequencing Data. Tech. Report FU Berlin (2010), http://publications.imp.fu-berlin.de/962/
13. 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)

14. Huang, W., Li, L., Myers, J.R., Marth, G.T.: ART: a next-generation sequencing read simulator. Bioinformatics (Oxford, England) (2012)
15. Jain, C., Misra, S., Zhang, H., Dilthey, A., Aluru, S.: Accelerating Sequence Alignment to Graphs. In: International Parallel and Distributed Processing Symposium (IPDPS) (2019), iSSN: 1530-2075
16. Jain, C., Zhang, H., Gao, Y., Aluru, S.: On the Complexity of Sequence to Graph Alignment. In: Research in Computational Molecular Biology. Cham (2019)
17. Jean, G., Kahles, A., Sreedharan, V.T., De Bona, F., Rätsch, G.: RNA-Seq read alignments with PALMapper. Current Protocols in Bioinformatics (2010)
18. Kavya, V.N.S., Tayal, K., Srinivasan, R., Sivadasan, N.: Sequence Alignment on Directed Graphs. Journal of Computational Biology (2019)
19. 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)
20. Köster, J., Rahmann, S.: Snakemake–a scalable bioinformatics workflow engine. Bioinformatics (Oxford, England) (2012)
21. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature Methods (2012)
22. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics (Oxford, England) (2009)
23. Limasset, A., Flot, J.F., Peterlongo, P.: Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs. Bioinformatics (2019), btz102
24. Liu, B., Guo, H., Brudno, M., Wang, Y.: deBGA: read alignment with de Bruijn graph-based seed and extension. Bioinformatics (Oxford, England) (2016)
25. Paten, B., Novak, A.M., Eizenga, J.M., Garrison, E.: Genome graphs and the evolution of genome inference. Genome Research (2017)
26. Pearl, J.: On the Discovery and Generation of Certain Heuristics. AI Magazine (1983)
27. Rautiainen, M., Mäkinen, V., Marschall, T.: Bit-parallel sequence-to-graph alignment. Bioinformatics (2019)
28. Rautiainen, M., Marschall, T.: Aligning sequences to general graphs in $O(V+mE)$ time. preprint (2017)
29. Salmela, L., Rivals, E.: LoRDEC: accurate and efficient long read error correction. Bioinformatics (Oxford, England) (2014)
30. Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O., Weigel, D.: Simultaneous alignment of short reads against multiple genomes. Genome Biology (2009)
31. Sellers, P.H.: An algorithm for the distance between two finite sequences. Journal of Combinatorial Theory (1974)
32. Sirén, J.: Indexing Variation Graphs. In: 2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX) (2017)
33. Sirén, J., Välimäki, N., Mäkinen, V.: Indexing Graphs for Path Queries with Applications in Genome Research. IEEE/ACM transactions on computational biology and bioinformatics (TCBB) (2014)
34. Smith, T.F., Waterman, M.S.: Comparison of biosequences. Advances in Applied Mathematics (1981)
35. 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)

A Appendix

A.1 Generic Algorithms: A* and Dijkstra

Algorithm 2 shows a generic implementation of the A* algorithm, roughly following [6]. 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)

function A*(G: Graph, S: Sources, T: Targets, h: Heuristic function)
  $f \leftarrow Map(default = \infty)$: Nodes $\to \mathbb{R}_{\geq 0}$ ▷ Map nodes from G to priorities
  $Q \leftarrow MinPriorityQueue(priority = f)$ ▷ Priorities according to f
  for all $s \in S$ **do**
    $f[s] \leftarrow 0.0$
    $Q.push(s)$ ▷ Initially, explore all $s \in S$
  while $Q \neq \emptyset$ **do**
    $curr \leftarrow Q.pop()$ ▷ Get state with minimal f to be expanded
    if $curr \in T$ **then**
      return BacktrackPath($curr$) ▷ Reconstruct a path to $curr$ (omitted)
    for all $(curr, next, cost) \in G.outgoingEdges(curr)$ **do**
      $\hat{f}_{next} \leftarrow f[curr] + cost + h(next)$ ▷ Candidate value for $f[next]$
      if $\hat{f}_{next} < f[next]$ **then**
        $f[next] \leftarrow \hat{f}_{next}$
        $Q.push(next)$ ▷ Explore state $next$
  assert *False* ▷ Cannot happen if T is reachable from S

function Dijkstra(G: *Graph*, S: *Sources*, T: *Targets*)
  $h(v) \leftarrow 0.0$ ▷ Constant-zero function h
  A$^\star(G, S, T, h)$

Algorithm 3 Recursive alignment used by Heuristic in Algorithm 1.

```
1: function RecursiveAlign(u, s, curr, best)                ▷ Return value is ≤ best
2:     if curr ≥ best then
3:         return best                                      ▷ Branch and bound: bounding
4:     if s = ϵ then                                        ▷ Reached a target
5:         return curr
6:     for all (u, v, ℓ, w) ∈ E_e where ℓ ∈ {s[0], ϵ} do
7:         suff = s[1 :] if ℓ ≠ ϵ else s
8:         best = RecursiveAlign(u, suff, curr + w, best)
9:     return best
```

A.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.

A.3 Parameter Estimation

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

Fig. 6 demonstrates the benefit of using a trie with the size reduction optimization (end of §4.1): increasing the trie depth D speeds up aligning but requires more memory. Selecting the trie depth based on the graph size $D = \lfloor \log_\Sigma |G_\mathbf{r}| \rfloor$ provides a reasonable trade-off between alignment time and memory.

Fig. 7 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.

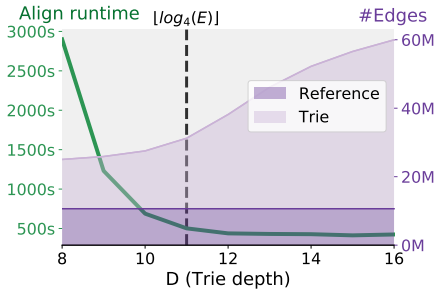

Fig. 6: Effect of D on performance of AStarix (MHC1 experiment). The dashed line shows our choice of D.

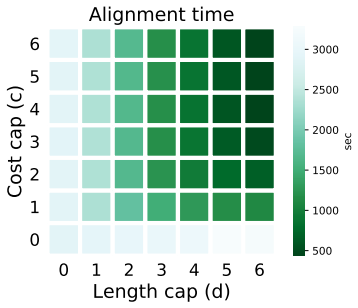

Fig. 7: Runtime of AStarix depending on d and c (MHC1 experiment).

A.4 Versions, commands, parameters for running all evaluated approaches

In the following, we provide details on how we executed the approaches discussed in §5:

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`

A.5 Notations

Table 2 summarizes the notational conventions used in this work.

Table 2: Notational conventions.

Object	Notation
Queries	$Q = \{q_i \mid q_i \in \Sigma^m\}$
Query	$q \in Q$
Length	$m := \lvert q \rvert \in \mathbb{N}$
Position in query	$q[i] \in \Sigma$, $i \in \{1, \dots, m\}$
Reference graph	$G_\mathtt{r} = (V_\mathtt{r}, E_\mathtt{r})$
Size	$\lvert G_\mathtt{r} \rvert := \lvert V_\mathtt{r} \rvert + \lvert E_\mathtt{r} \rvert \in \mathbb{N}$
Nodes	$u, v \in V_\mathtt{r}$, $n := \lvert V_\mathtt{r} \rvert \in \mathbb{N}$
Edges	$e \in E_\mathtt{r} := V_\mathtt{r} \times V_\mathtt{r} \times \Sigma$
Edge letter	$\ell \in \Sigma$
Reference graph with a trie	$G_\mathtt{r}^+ = (V_\mathtt{r}^+, E_\mathtt{r}^+)$
Trie depth	$D \in \mathbb{N}_{>0}$
Edit graph	$G_\mathtt{e} = (V_\mathtt{e}, E_\mathtt{e})$
Edit costs	$0 \le \Delta_\text{match} \le \Delta_\text{subst}, \Delta_\text{ins}, \Delta_\text{del}$
Alignment	$\pi \in E_\mathtt{e}^*$ and $\sigma(\pi) = q$
Optimal alignment	$\dot{\pi} \in E_\mathtt{e}^*$
Alignment cost	$\text{cost}(\pi) \in \mathbb{R}_{\ge 0}$
Alignment graph	$G_\mathtt{a}^q = (V_\mathtt{a}^q, E_\mathtt{a}^q)$
Size	$N := \lvert V_\mathtt{a}^q \rvert \in \mathbb{N}$
State	$\langle u, i \rangle \in V_\mathtt{a}^q := V \times \{0, \dots, m\}$
Edges	$(\langle u, i \rangle, \langle v, j \rangle, \ell, w) \in E_\mathtt{a}^q \subseteq V_\mathtt{a}^q \times V_\mathtt{a}^q \times \Sigma_\epsilon \times \mathbb{R}_{\ge 0}$
Edge cost	$w \in \mathbb{R}_{\ge 0}$
In all graphs	$G(V, E) \in \{G_\mathtt{r}, G_\mathtt{e}, G_\mathtt{a}^q\}$
Walk	$\pi \in G : \pi \in E^*$
Walk spelling	$\sigma(\pi) \in \Sigma^*$
Walk begin and end nodes	$\mathit{begin}(\pi), \mathit{end}(\pi) \in V$
Path	a walk without repeating nodes
AStarix	$A^\star(G, S, T, h)$
Graph	$G = (V, E)$
Nodes	$u, v \in V$
Edges	$e \in E \subseteq V \times V \times \mathbb{R}_{\ge 0}$
Source states	$S \subseteq V$
Target states	$T \subseteq V$
Upcoming sequence	$s \in \Sigma^k$
Cost cap	$c \in \mathbb{R}_{\ge 0}$
Depth cap	$d = \lvert s \rvert$
Heuristic function	$h(u, s) \colon V \to \mathbb{R}_{[0;\infty]}, \lvert s \rvert \le d$
Minimum cost to a target	$h^*(u, s)$
Cost of the cheapest path from u to v	$k(u, v), u, v \in G$
Optimistic	$h(u, s) \le \min_\pi \text{cost}(\pi), \pi \colon \pi$ starts from $u, \sigma(\pi) = s$
Explored state	A state pushed to the queue of Algorithm 2
Expanded state	A state popped from the queue of Algorithm 2