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
Master in Computer Science,
Shumen University

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

Pesho Ivanov, *Optimal sequence alignment using A**, 2022
Cover image by Yavor Milanov

DOI: 10.3929/ethz-a-

To my high-school teachers
Svilen Rusev, who introduced me to science, and
Biserka Yovcheva, who equipped me to research it.

ABSTRACT

Sequence alignment is the task of finding similarities between sequences. It has been a central building block in molecular biology since DNA, RNA and protein sequences were first obtained half a century ago. Sequence alignment is applied to research and medicine, with applications in evolutionary biology, genome assembly, oncology and many others. The analyses of the growing amounts of genomic data require algorithms with high accuracy and speed. Moreover, the ongoing transition from single genome references to pangenomes (representative for a whole population) motivates novel alignment algorithms.

We consider the two problems: *semi-global* alignment of a set of DNA reads to a pangenome graph reference (possibly containing cycles), and *global* (end-to-end) alignment of two sequences. Recent theoretical results conclude that strongly subquadratic algorithms are unlikely to exist for the worst case. Moreover, the runtime and memory of existing optimal algorithms scale quadratically even for similar sequences. An open problem is to develop an algorithm with linear-like empirical scaling on inputs where the errors are linear in n. In the present thesis we introduce an approach that aims to solve this problem in order to develop practical optimal alignment algorithms.

Modern optimal aligners perform uninformed search – they neglect information from the not-yet-aligned parts of the sequences. We exploit this information in a principled framework where an alignment with minimal edit distance is equivalent to a shortest path in an alignment graph, and the remaining information about the sequences is captured in a heuristic function that estimates the length of a remaining shortest path. The classic shortest path algorithm A* uses such a heuristic to direct the search, and yet it finds provably shortest paths given that the heuristic is *admissible*, i.e. it always return a lower bound on the edit distance of the remaining suffixes. Curiously, previous attempts to apply A* to multiple sequence alignment (MSA), did not result in practical algorithms even for pairwise alignment. In the present thesis we investigate how to do that. In the shortest path formulation, we (i) suggest a novel highly-informed admissible heuristic, (ii) design efficient algorithms and data structures for computing the heuristic, (iii) prove their optimality, (iv) implement the presented algorithms, and (v) compare their performance and scaling to other optimal algorithms. Our approach is provably optimal according to edit distance, its runtime empirically scales subquadratically (and sometimes even near-linearly) with the output size, which translates to orders of magnitude of speedup compared to state-of-the-art optimal algorithms.

Throughout this thesis, we demonstrate how to encompass various dimensions of the input complexity while preserving the speed and optimality. First, we demonstrate how to apply a trie index to scale the alignment runtime sublinearly with the reference size. Then, we introduce a novel seed heuristic for A* which enables aligning of long sequences (up to 100 Mbp) near-linearly with their length. Last, we extend the seed heuristic to a general chaining seed heuristic that encompases inexact matching, match chaining, and gap costs to raise the tolerated error rate (to 30% for synthetic data and 10% for real data). Interestingly, our seed heuristic challenges the omnipresent seed-(chain)-extend paradigm to sequence alignment which aims to connect long and high-quality seed matches. We, instead, use the information of the lack of matches to dismiss suboptimal alignments, which leads to optimizing seeds for being short and not having many matches.

These first steps to scalable optimal alignment using A* give rise to a number of research directions: approaching other alignment types, more general optimization metrics, relaxed optimality guarantees, performance analyses, improved heuristics, applications outside of biology, and more performant algorithms and implementations.

ZUSAMMENFASSUNG

Beim Sequenzabgleich geht es darum, Ähnlichkeiten zwischen Sequenzen zu finden. Sie ist Sie ist ein zentraler Baustein der Molekularbiologie, seit vor einem halben Jahrhundert erstmals DNA-, RNA- und Protein Sequenzen vor einem halben Jahrhundert erstmals gewonnen wurden. Sequenzabgleich wird angewandt der Forschung und der Medizin, mit Anwendungen in der Evolutionsbiologie, der Genom Genomzusammensetzung, Onkologie und vielen anderen Bereichen. Die Analyse der wachsenden Mengen an genomischer Daten erfordert Algorithmen mit hoher Genauigkeit und Geschwindigkeit. Außerdem ist der Übergang von Einzelgenomreferenzen zu Pangenomen (repräsentativ für eine ganze für eine ganze Population) motiviert neuartige Algorithmen zum Alignment.

Wir betrachten die beiden Probleme: *semi-global* Alignment eines Satzes von DNA-Reads an eine Pangenom-Graphenreferenz (die möglicherweise Zyklen enthält), und *global* (Ende-zu-Ende) Abgleich zweier Sequenzen. Jüngste theoretische Ergebnisse zeigen dass stark subquadratische Algorithmen für den schlimmsten Fall unwahrscheinlich sind. Außerdem skalieren die Laufzeit und der Speicherplatz der vorhandenen optimalen Algorithmen selbst für ähnliche Sequenzen quadratisch. Ein offenes Problem ist die Entwicklung eines Algorithmus mit linear-ähnlicher empirischer Skalierung für Eingaben zu entwickeln, bei denen die Fehler linear in n sind. In der vorliegenden Arbeit stellen wir einen Ansatz vor, der darauf abzielt dieses Problem zu lösen, um praktische optimale Alignment-Algorithmen zu entwickeln.

Moderne optimale Aligner führen eine uninformierte Suche durch - sie vernachlässigen Informationen aus den noch nicht alignierten Teilen der Sequenzen. Wir nutzen diese Informationen in einem einem prinzipiellen Rahmen, in dem ein Alignment mit minimalem Editierabstand einem kürzesten Pfad in einem Alignment-Graphen entspricht, und die verbleibende Informationen über die Sequenzen werden in einer heuristischen Funktion erfasst, die die die Länge eines verbleibenden kürzesten Pfades schätzt. Der klassische kürzeste Weg Algorithmus A* verwendet eine solche Heuristik, um die Suche zu lenken, und er findet dennoch nachweislich kürzeste Pfade, wenn die Heuristik *zulässig* ist, d.h. er immer eine untere Schranke für die Edit-Distanz der verbleibenden Suffixe liefert. Seltsamerweise haben frühere Versuche, A* auf das Multiple Sequence Alignment (MSA) anzuwenden, nicht einmal für paarweises Alignment zu praktischen Algorithmen geführt. In dieser vorliegenden Arbeit untersuchen wir, wie dies möglich ist. In der Formulierung des kürzesten Weges, schlagen wir (i) eine neue,

hochinformierte zulässige Heuristik vor, (ii) entwerfen effiziente Algorithmen und Datenstrukturen für die Berechnung der Heuristik, (iii) beweisen ihre Optimalität, (iv) implementieren die vorgestellten Algorithmen und (v) vergleichen ihre Leistung und Skalierung mit anderen optimalen Algorithmen. Unser Ansatz ist nachweislich optimal nach der Edit-Distanz, seine Laufzeit skaliert empirisch subquadratisch (und manchmal sogar nahezu linear) mit der Ausgabegröße, was was zu einer Beschleunigung um Größenordnungen im Vergleich zu optimalen Algorithmen.

In dieser Arbeit zeigen wir, wie man verschiedene Dimensionen der der Eingabekomplexität unter Beibehaltung der Geschwindigkeit und Optimalität. Zunächst zeigen wir demonstrieren wir, wie man einen Trie-Index anwendet, um die Laufzeit des Alignments sublinear mit der Referenzgröße skaliert. Dann führen wir eine neue Seed-Heuristik für A* ein, die das Alignment von langen Sequenzen (bis zu 100 Mbp) nahezu linear mit deren Länge ermöglicht. Schließlich erweitern wir die Seed-Heuristik zu einer allgemeinen Chaining-Seed-Heuristik die ungenaue Übereinstimmung, Verkettung von Übereinstimmungen und Lückenkosten einschließt, um die tolerierte Fehlerrate zu erhöhen (auf 30% für synthetische Daten und 10% für reale Daten). Interessanterweise stellt unsere Seed-Heuristik das allgegenwärtige Seed-(Chain)-Extend Paradigma des Sequenzalignments heraus, das darauf abzielt, lange und hochwertige Seed Übereinstimmungen. Wir nutzen stattdessen die Information, dass es keine Übereinstimmungen gibt, um um suboptimale Alignments auszuschließen, was dazu führt, dass die Seeds daraufhin optimiert werden, kurz zu sein und nicht viele Übereinstimmungen haben.

Diese ersten Schritte zu einem skalierbaren optimalen Alignment mit A* geben Anlass zu einer Reihe von Forschungsrichtungen: Annäherung an andere Alignment-Typen, allgemeinere Optimierungsmetriken, entspannte Optimalitätsgarantien, Leistungsanalysen, verbesserte Heuristiken, Anwendungen außerhalb der Biologie und leistungsfähigere Algorithmen und Implementierungen.

Sequenzalignment ist der Prozess der Erkennung von Ähnlichkeiten zwischen Sequenzen. Seit vor einem halben Jahrhundert zum ersten Mal genetische Sequenzen sequenziert wurden, ment ist eine grundlegende Aufgabe in der Molekularbiologie mit Anwendungen in der Evolutionsbiologie, Genomassemblierung, Variationserkennung und andere. Ein laufender Übergang von uns- Die Umstellung von Genomen auf die Verwendung von Pangenomen motiviert zum Überdenken der klassischen Ausrichtung Algorithmen. Die Vielfalt der Anwendungen kombiniert mit der wachsenden Menge an genetische Daten motivieren die Entwicklung schneller und genauer Ausrichtungsalgorithmen.

Existierende Ausrichtungsalgorithmen sind entweder optimal, aber quadratisch oder schnell, aber geeignet. nahe. Diese Arbeit prosiert einen eleganten Ansatz zur

Ausrichtung basierend auf dem A* Algorithmus, der sowohl heuristisch schnell als auch nachweislich optimal ist. Es wurde gezeigt Diese Ausrichtung ist in stark subquadratischer Zeit im Allgemeinen wahrscheinlich nicht lösbar Fall. Das Ziel, das wir in dieser Arbeit verfolgen, ist die Anwendung des A* Ansatz als möglichst viele Arten von Daten und bleibt dabei schnell und optimal. Wir betrachten zwei Arten von Alignment: semi-global, um einen DNA-Satz zu kartieren Sequenzen zu einer Pangenom-Referenz; und global zur Berechnung der Bearbeitungsentfernung

Abstand zwischen zwei Folgen. Um verschiedene Datendimensionen handhaben zu können, verwenden wir schlagen mehrere Techniken vor und untersuchen empirisch ihre Laufzeitskalierung: ein Versuch Index ermöglicht sublineare Skalierung mit der Referenzgröße, Seed-Heuristik ermöglicht nahezu lineare Skalierung mit Sequenzlänge und ungenaue Seed-Matching und Match-Verkettung Skalierung auf hohe Fehlerraten ermöglichen. Aufgrund der überlegenen Skalierung des A* sich nähern, Unsere prototypischen Implementierungen laufen um Größenordnungen schneller als bestehende optimale Anflüge auch bei langen Fehlsequenzen.

Wir sehen eine Vielzahl zukünftiger Richtungen für die Weiterentwicklung von A* für Sequenz Ausrichtung, einschließlich anderer Ausrichtungsarten, Verallgemeinerung der Bearbeitungsentfernungsmetrik, Lockerung der Optimalitätsgarantie, theoretische Analysen der Leistung und Effizientere Implementierungen für den Produktionseinsatz.

THESIS PUBLICATIONS

1. Ivanov, P., Bichsel, B., Mustafa, H., Kahles, A., Rätsch, G. & Vechev, M. ***AStarix: Fast and Optimal Sequence-to-Graph Alignment***[1] in *RECOMB* (2020).
2. Ivanov, P., Bichsel, B. & Vechev, M. ***Fast and Optimal Sequence-to-Graph Alignment Guided by Seeds*** in *RECOMB* (2022).
3. Groot Koerkamp†, R. & Ivanov†, P. **Exact global alignment using A* with seed heuristic and match pruning**[2]. *submitted to Oxford Bioinformatics* (2023).

OTHER WORKS DURING THE PHD PROGRAM

1. Ivanov, P. & Vechev, M. ***CellDive: Gene-level analysis of lymphocyte single-cell RNA data based on a clonality specification language***[3] GitHub repository. 2018.
2. 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).
3. Ivanov, P. ***minSH: Minimal re-implementation of A* with seed heuristic***[4] GitHub repository. 2023.

1 https://github.com/eth-sri/astarix
2 https://github.com/RagnarGrootKoerkamp/astar-pairwise-aligner
†These authors contributed equally to this work.
3 https://github.com/pesho-ivanov/celldive
4 https://github.com/pesho-ivanov/minSH

ACKNOWLEDGEMENTS

I am lucky to have such supportive parents as Ivo and Svetla who fully dedicated themselves to my development, introduced me to their professions of engineering and medicine, and supported my immense freedom in life, and sow my childhood dream for science.

I am greatful for the opportunities to study, teach and socialize with teachers, mentors fellow students in numerous schools and Summer camps, most notably the A&B informatics school in Shumen (Bulgaria) by Biserka Yovcheva, Summer computer school in Russia by Victor Matyukhin, physics preparations for IYPT in Shumen by Svilen Rusev, Yandex ShAD bioinformatics evening school in Moscow by Mikhail Gelfand (mentored by Konstantin Severinov), Algorithmic Biology Lab in St.Petersburg by Pavel Pevzner (mentored by Sergey Nurk).

I owe my first academic steps the (then called) Software Reliability Lab in ETH Zurich by Martin Vechev who supported my academic freedom for combining biology and informatics and guided the principled approach to formal correctness. I am especially thankful to my colleagues Benjamin Bichsel (Beni) and Dimitar K. Dimitrov (Mitko) for their pedantic attitude towards correctness and consistency, so much needed in the biological sciences.

I am grateful to my coauthors Harun Mustafa, André Kahles and Gunnar Rätsch for directing my A* work towards practical results. I am also greatful to Yun-Tsan Chang, Desislava Ignatova and Emmanuella Guenova from the Dermatology Clinic in University of Zurich for collaborating on a cross-disciplinary project on single-cell RNA data from lymphoma patients, despite the lack of publishable results.

I am pleased to see that Ragnar Groot Koerkamp accepted my invitation to join the research direction of A* for sequence alignment. It is a pleasure for me to cooperate on a shared passion with a like-minded person. I am also greatful to Iskren Chernev and Mykola Akulov who dedicated their time and effort to developing tools and visualizations towards further developing the A* for alignment. Additionally, I an greatful to Benjamin Bichsel, Maximilian Mordig and André Kahles for valuable comments on drafts, and to Sergey Nurk for his help with the human data.

I also thank Pavol Bielik for helping me accept certain sides of academia, Gagandeep Singh for giving me a good example for a future idea of using machine learning for performance optimization while retaining formal correctness, Fiorella Meyer for supporting the administrative side of my doctoral program, and Arshavir Ter-Gabrielyan for his peer support towards preparing this thesis. I thank Yavor Milanov

who introduced me to his disrespect of art and made the beautiful thesis cover, Pencho Yordanov who convinced me that new and expensive data is not the only path towards good science, and my virtual friends who helped me develop skills for successful collaboration. I am greatful to the practice of Vipassana meditation for the calm concentration resulting in imagining the *seed heuristic* that is the base for this thesis. I thank the anonymous paper reviewers for their valuable feedback, confirming the sanity of my ideas, and for being accepting to new academicians like me.

I thank the countless friends, teachers, students, academics, past scientists and philosophers, who patiently listened and discussed, shared and taught me their perspectives, allowing me to better understand and present my research. Additionally, I gladly accepted the broad influence of all virtual friends, fellow travellers and couchsurfers, hippies, vegans, meditators, paragliding pilots, motorcyclists, wikipedia contributors, activists, psychonauts, and lovers, for sharing their passions and building up the creative environment around me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION

FREQUENTLY USED TERMS

edit distance	minimal cost to edit one sequence to another
alignment	correspondence between letters of two sequences
• global	fully aligning two sequences to each other
• semi-global	fully aligning a query somewhere in a reference
• reads alignment	multiple semi-global alignments to the same reference
dynamic programming	optimization technique by reusing partial results
• explored state	pushed to the priority queue
• expanded state	popped from the priority queue
Dijkstra's algorithm	uninfomred shortest path algorithm
A* algorithm	generalization of Dijkstra, informed by a heuristic
• admissible heuristic	lower bound on the remaining shortest distance
graph reference	a compressed representation of multiple references
trie index	tree structure with short reference subsequences
seed heuristic	our proposed heuristic function for A*
• chaining	requiring that all matches are consequitive
• inexact matching	allowing a number of errors (1) when matching a seed
• potential	maximal value of the heuristic
performance	runtime and memory footprint
scaling	change (not necessarily asymptotic) with bigger input

FREQUENTLY USED SYMBOLS

$\Sigma = \{A, C, G, T\}$	alphabet
$A, B \in \Sigma^*$	sequences
$q \in \Sigma^m$	query/read
$\Delta_{\text{match}}, \Delta_{\text{subst}}, \Delta_{\text{ins}}, \Delta_{\text{del}} \geq 0$	edit costs for match, insertion, deletion and substitution of a single letter
$ed(A, B)$	edit distance
$G_{\mathtt{r}} = (V_{\mathtt{r}}, E_{\mathtt{r}})$	reference graph
$\langle u, i \rangle$	state (subalignment of A)
$G = (V, E)$	alignment graph on states V
D	trie depth
g	best distance from start
$h \colon V \to \mathbb{R}_{\geq 0}$	heuristic function
$f = g + h$	estimated total distance
$h\langle u, i \rangle$	seed heuristic
$s \in \mathcal{S}$	seeds
k	seed length
r	seed potential
$P\langle i, j \rangle = r \cdot \lvert\mathcal{S}_{\geq i}\rvert$	state potential
$m \in \mathcal{M}_s$	matches of seed s
$\mathrm{c_m}(m) \in [0, r)$	cost of match m
$\gamma(m, m')$	chaining cost
$c_{\text{gap}}(\langle i, j \rangle, \langle i', j' \rangle) := \lvert(i' - i) - (j' - j)\rvert$	gap cost
$c_{\text{seed}}(u, v) = r \cdot \mathcal{S}_{u \ldots v}$	seed cost
$c_{\text{gs}} = \max(c_{\text{gap}}, c_{\text{seed}})$	gap-seed cost
	crumbs on nodes for match m

INTRODUCTION

In this chapter we will intuitvely describe the alignment problems of interest, their relation to genomics, the algorithms and instruments that have been developed through the last 60 years. Then we will pinpoint the motivation for this thesis, as well as its scientific contributions.

PROBLEM STATEMENT

Sequence alignment. One of the simplest operations on sequences is to compare them. More precisely, we consider the problem of *pairwise sequence alignment*: given two sequences, one can be aligned to the other letter-by-letter. Depending on the parts of the sequences that are being aligned, we differentiate types of alignment (Fig. 0.1) and focus on global and semi-global.

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 suefficient but finding several best (top-K) alignments. In the context of read alignment, a set of reads is aligned to the same reference sequence so an indexing procedure is often useful for the performance.

Minimizing edits. It would have been relatively easy to find alignments if the sequences did not differ from each other. Nevertheless, the real data may be subject to such processes as biological evolution and sequencing technologies, which result in differences between compared sequences. Commonly, the most probable explaination of these differences is to be reconstructed. If we assume an *error model* which repeatedly applies a random single-letter edit (substitutions, insertions and deletions), then the most probable sequence of edits would be one with a minimal number of edits (possibly weighting different types of edits with different costs). Throughout this thesis we consider this most widespread objective—it provides a tradeoff between expressivity and computability—it reasonably estimates the real world sequences while disregarding any memory in the error model.

Global alignment. The simplest type of alignment is to find a sequence of edits that convert one whole sequence A to another sequence B. Note that converting A to B corresponds to the reverse process of converting B to A (e.g. insertions correspond to deletions). If we ignore the order of applying the edits, the result can

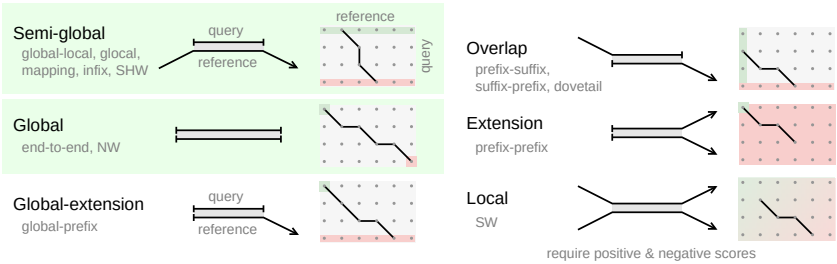


FIGURE 0.1: Names, alignment and paths of main alignment types. Synonims of the alignment names are shown in grey. The alignments are simplistically shows without edits. Paths corresponding to the alignments are shown as paths on an alignment graph. All paths start at green nodes and finish at red nodes. With green background are shown the two alignment types considered in this thesis.

be equivalently represented simpler as an *alignmnet*: each letter from A has either not been edited (so it corresponded to a matching letter in B), or has been substituted for another letter (so it corresponded to a mismatching letter in B), or has been deleted (which can be represented as mismatching it with an inserted fictive *gap symbol* in B). Additionally, letters could have been added to A (represented as a gap in A mismatching a letter in B). Note that the number of edits is equal to the number of mismatches in the corresponding alignment. The minimal number of edits necessary to transform A to B is known as *Levenshtein distance* when each edit operation is equally costly, and more generally as *edit distance*, when the edit operations are weighted depending on their type. Clearly, finding an alignment with a minimal number of edits implies computing the edit distance.

Semi-global alignment. Semi-global sequence alignment (also called approximate/fuzzy string search outside computational biology) is the alignment of a whole *query* sequence to a continuous region (subsequence) of a *reference* sequence. This region is unspecified, so semi-global alignment is a more complex problem than global alignment. We target semi-global alignment by again minimizing edit distance. Finding an alignment implies also finding a reference region where the query aligns, which is known as *mapping* and is sufficient for some applications.

Semi-global alignments on the same reference. If multiple queries have to be aligned on the same reference, powerful optimizations are applicable.

Sequence-to-graph alignment. A set of reference sequences can be compactly represented as paths in a graph (possbly cyclic). *Sequence-to-graph* alignment

is a semi-global alignment between a query sequence and a path in a reference graph [79].

Alignment as shortest path. An alignment of two sequences is a letter-to-letter correspondence between the two sequences (using gap letters for insertions and deletions). Considering the corresponding letters consecutively, each pair of letters proceeds to a longer prefix of A (in case of an insertion), to a longer prefix of B (in case of a deletion), or to longer prefixes of both (in case of matching or substituted letters). If we define an *alignment graph* to contain all pairs of prefixes as nodes, and all edits as edges, then any alignment will correspond to a path. Moreover, if the edges are weighted by the edit costs, any alignment with a minimal edit distance will correspond to a shortest path. Thus, we will find best alignments as shortest paths in an alignment graph.

Objective functions. Assuming that two sequences alignment have no errors is reasonable only for very short sequences. Othewise, for sequences of equal length we can calculate *Hamming distance* – the number of neccessary substitution operations to transform one to the other. Within this thesis we assume the edit distance and Levenshtein distance, which also allows for gaps. They can further be generalized to *affine costs* (starting a gap is more expensive than extending it), *convex* and *concave* costs (extending the gaps gets cheaper or more expensive with length) and others. The strive is for a ballance between optimization metric and the computational efficiency.

GENOMIC SEQUENCES

DNA structure. The discovery of the structure and function of DNA[2] seventy years ago started a revolution in our understanding of the living world. The non-branching (linear or circular) DNA structure resembles similar the linear structure of human text and speech. This property is massively exploited by both *sequencing* technologies used to digitalize genomic information, and by algorithms used to analyze the data.

Sequence alignment. Pairwise sequence alignment has been a core problem in computational biology for more than half a century, exhibiting a multitude of applications, including variation analysis, de novo assembly, read alignment, variant detection, phylogenetics, RNA quantification, multiple sequence alignment, differential expression [82].

Biological and technical variation. Even though our method should be applicable outside of DNA sequening and even computational biology, we focus on genomic data for its immense practical importance. More generally, the differences between the two sequences are explained by biological variation (resulting from

differences in the sequenced organism) and technical variation (resulting from sequencing errors). Next, we review the main sequencing technologies up to date and the data profile they produce.

Read mapping. Each sequence from a new biological sample can be compared to a reference genome (if such is available) in order to identify mutations, quantify expression per gene, and many other types of analysis. To figure out the location in the reference each sequence most probably originates from, it should first be semi-globally aligned to the reference. If the sequences of two proteins or viruses have been reconstructed, aligning them globally reveals the most probabale mutations between them.

First generation sequencing. Genomic (DNA, RNA) sequencing is a technology that takes genetic material and produces a data file with a set of sequences. Since the late 1970s, Sanger sequencing is the first widely used technology that dominated the field for over 40 years. The produced *reads* (each read is a sequence originating from a continuous DNA fragment) are of length $>$1000 nucleotides and have very high quality ($<$0.001% error rate) [42].

Second generation (high-throughput / next-generation / shotgun) sequencing. Initially, the sequenced data has been scarce and the sequence alignment was done by hand. The development of high-throughput sequencing in the late 1990s and 2000s lead to higher amounts of cheaper data due to sequencing millions and billions of molecules in parallel. The amount and complexity of the available genomic data bacame not only intractable by hand, but also challenging for computer analysis.

Third generation (long-read) sequencing. Since the end of the 2000s, novel technologies started to appear that are capable of producing long sequences. Long sequences enabled massive applications but the limiting factor was the high error rate (up to 20%) and the higher price.

Pangenomics and genome graphs. Traditionally, a single linear reference sequence has been used as a consensus genome. Nevertheless, linear references are limited in representing biological variation which the modern sequencing technologies are producing in great abundance. Thus, in the 2010s, the field started shifting from linear references towards sets of reference sequences, and their compact *genome graph* representations. A path in such a graph may represent a specie, an individual, a cell, a haplotype [60, 68]. Semi-global alignment is more accurate on reference graphs than on linear references due to capturing genomic variation [73].

Practical alignment algorithms. Practical alignment algorithms exploit similarities between the sequences to speed up the computation but give up on optimal-

ity guarantees. Thus, a satisfactory solution that combines optimality guanratees with heuristic performance is yet to be developed [95].

RELATED WORK

Over the last half-century, many algorithms have been developed for sequence alignment [37]. With the increasing quantity and length of genomic sequences, the need for fast and accurate algorithms is steadily increasing [91]. There are multiple directions in which the sequence alignment algorithms are being developed: scalable algorithms, generalizing the optimization metric, employing heuristics for approximate solutions, adapting existing algorithms to novel types of data, exploiting parallel computing, implementing fast and memory-efficient software. This thesis focuses on algorithmic approaches to sequence alignment so here we outline the leading existing optimal algorithms, the suboptimal seeding heuristics that we will will use in an optimal way, and the A* shortest path algorithm which will be the base for our provably optimal and heuristically fast approach.

Dynamic programming for global alignment. The first efficient solution for global alignment, known as the Needleman-Wunsch algorithm, got developed around 1968 [8, 9]. This algorithm is based on the dynamic programming (DP) technique [3] of splitting a task into overlapping subtasks (or *states*) each of which can be solved once and then being reused. Each node of the alignment graph (aligning a prefix of the first sequence to a prefix of the second sequence) is a state in the DP. Each subtask can be solved by reducing it to already solved alignments of shorter prefixes, and extending them by a last operation of matching or mismatching the last letters from the prefixes, inserting one, or inserting the other (equivalent to deleting from the first sequence). Each subtask is solved for $O(1)$ so the quadratic number of prefix pairs is solved for quadratic overall time, even though the number of possible alignments is exponential. Independently, it was applied to compute edit distance for biological sequences [9, 10, 12, 13]. This well-known algorithm is implemented in modern aligners like SeqAn [70] and Parasail [63]. Improving the performance of these quadratic algorithms has been a central goal in later works.

Dynamic programming for semi-global alignment. Semi-global alignment is a related problem to global alignment but in addition to the need to align letter-to-letter, it also requires determining a starting position in the reference. This task has also been efficiently solved using a similar DP approach [18, 20] taking $O(nm)$ to align a single query of length n to a reference of length m. It is highly redundant to explore the whole reference for each alignment, so various data structures were suggested that can be precomputed to index the reference, and then reuse this

index to quickly align multiple query sequences 1988 [28] and has become central to read alignment of high-throughput sequencing.

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 [59]. 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 [19] considering all possible starting nodes.

Suboptimal heuristic algorithms. All optimal solutions are near-quadratic and this is justified by the theoretical lower bound. To speed up beyond this near-quadratic barrier, current practical algorithms break the optimality guarantee with the hope that the produced alignments are accurate enough, or give up on alignment alltogether. Heuristics are employed for both sequence-to-sequence alignment and sequence-to-graph alignment, to keep the computation tractable [73, 31, 53], especially for large populations of human-sized genomes. In the last decades, approximate and alignment-free methods satisfied the demand for faster algorithms which process huge volumes of genomic data [76]. While heuristics may produce acceptable alignments in many cases, it fundamentally does not provide quality guarantees, resulting in suboptimal alignment accuracy.

Seed-and-extend. Since optimal alignment is often intractable, many aligners use heuristics, most commonly the *seed-and-extend* paradigm [31, 53, 43]. 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.

Shortest path formulation and A* for sequence alignment. A* is a shortest path algorithm that generalizes Dijkstra by directing the search towards the target [7]. A* is an *informed* search algorithm – it relies on a problem-specific huristic function that estimates the remaining distance from each explored node to the target. If this heuristic is *admissible*, i.e. it never overestimates the actual distance to the target, A* is guaranteed to find an optimal path. If the heuristic is accurate – similar to the actual remaining distance, then A* finds the path with minimal exploration. If the heuristic can be computed efficiently, the overall runtime stays low. Designing a heuristic function with all of these properties has been challenging in the context of sequence alignment. Attempts to solve optimal pairwise alignment by A* were not scalable due to overly simplistic heuristic functions [72]. Tries to

solve multiple sequence alignment (MSA) by A* were also impractical due to the persisting exponential scaling with the number of sequences [36, 39, 38].

Optimal semi-global alignment to graph references. 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 [45, 48]. 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 [56] to index walks in an augmented reference sequence, forming a local genome graph [77]. VG [73] uses a similar technique [69] to index variation graphs representing a population of references. Modern sequence-to-graph aligners use SIMD instructions (VG [73] and PaSGAL [74]) or reformulations of edit distance computation to allow for bit-parallel computations in GraphAligner [75]. Many of these, however, are designed only for specific types of genome graphs, such as *de Bruijn* graphs [61, 84] and variation graphs [73]. 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 [73, 81].

Trie index. The specifics with semi-global alignment requires a trie-like index which is well-used in the fields, and known since 1912 [1] and used in informatics since 1959 [5].

MOTIVATION

We motivate this thesis by the importance of optimality guarantees, the existence of unused information for scaling improvements, and the existance of informed search algorithms that can be heuristically fast and provably optimal. Additionally, novel alignment algorithms are useful in the modern pangenomics setting.

Practical read aligners are approximate

Curerent practical semi-global aligners are either vague about the metrics they optimize or they do not guarantee optimality [91]. An approximate algorithm can produce wrong results either because it did not succeed in optimizing its objective function or because it was aiming at a poor objective function. Requiring the solutions to be optimal according the an objective, would eliminate the former issue and focus the research on the latter.

Sequence alignment is an early step in most bioinformatics pipelines, so incorrect alignments can cause biases in downstream analyses. For instance, the combination of high variability of MHC sequences in humans and small differences between

alleles [51] leads to a risk of misclassifications due to suboptimal alignment. Guaranteeing optimal alignment against all variations represented in a graph would be a major step towards alleviating those biases.

While heuristic algorithms usually 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) [60], in complex but rare genotypes arising from somatic-subclones in tumor sequencing data [50], or in the presence of frequent sequencing errors [58]. Thus, the produced results are doubted.

Optimal aligners are slow

There is a fundamental trade-off between perforamance and optimality guarantees: an algorithm that is allowed suboptimality may exploit the lesser restrictions for better performance. Given the worst-case analysis requiring near-quadratic runtime even to compute edit distance exactly [59], it is understandable why most scholars are skeptical about faster optimal algorithms.

For semi-global sequence-to-sequence alignment, a common consensus seems to be that optimal algorithms are infeasible for read alignment, especially for long reads. All production read aligners following the approximate seed-extend paradigm [91][5].

Existing optimal sequence-to-graph aligners rely on dynamic programming (DP) and always reach quadratic worst-case asymptotic runtime [80]. Such aligners include Vargas [87], PaSGAL [74], GraphAligner [75], HGA [88], and VG [73], which focus on bit-level optimizations and parallelization to futher increase their throughput.

For global alignment, optimal algorithms are nevertheless used in practice, despite of their quadratic scaling. The ongoing competition between the optimal aligners employs both algorithmic advancements and implementation optimizations on caching, bit-parallelization, GPU. (Chapter 3).

Even for related sequences, the fastest optimal global [90, 66] and semi-global aligners [75] scale quadratically when the edit distance increases with the length, which is the case for sequencing errors and biological variation. In the age of big data and long reads (e.g. PacBio, ONP), this quadratic scaling with length is prohibitive, so the algorithms with practical usage (e.g. minimap2, bwt, kallisto) do not guarantee optimality but run in subquadratic time [76]. This is a computational bottleneck given the growing amounts of biological data and the increasing sequence

5 This study examines 107 aligners.

lengths [76]. For each type of sequence alignment (e.g. global and semi-global), a tradeoff exists between amount of computation and the alignment accuracy.

The gap between fast and optimal global alignment has been recognized but no optimal algorithms are known that run subquadratically for related sequences [95].

Room for performance improvement for optimal alignment

The following observations demonstrate room for improvement for the expected runtime for optimal semi-global and global alignment. Moreover they hint towards information that is currently not harnessed for the average case:

Observation 1. *The output of the alignment problem is linear in the input size.*

Observation 2. *Optimal global alignment with only substitutions take linear time.*

Observation 3. *Approximate aligners scale better and run faster than optimal.*

Observation 4. *Existing optimal algorithms for semi-global alignment are near-quadratic not only in the worst case but also for related sequences.*

Observation 5. *The quadratic DP to global alignment (Needleman-Wunsch) also solves the more complex semi-global and local alignment (Smith-Waterman).*

Paradox 1. *A long query has more information that may hint towards the best semi-global alignment position in a reference but all optimal algorithms are strictly slower for longer sequences.*

Observation 6. *The existing optimal aligners choose next step based only on previous (prefix) information and ignore the remaining (suffix).*

These observations and paradoxes shape the main question of this thesis:

Open problem 1. *Can we design more scalable and faster algorithms for optimal alignment of* related *sequences?*

$A^\star$ can harness the heuristic speed and provable optimality

Algorithm correctness is arguably a useful property which is often not simple to guarantee. It can undoubtedly improve accuracy, especially in the case of complex data. But finding an optimal alignment requires a conceptually different approach than finding an approximate alignment. Instead of finding *one* good alignment, finding an optimal alignment requires proving that *all* other exponentially-many alignments are not better.

	Ivanov et al. (2020)	Ivanov et al. (2022)	Groot Koerkamp & Ivanov (preprint)
Problem	Seq-to-graph alignment (=Read mapping)		Global alignment
Introduces	**Trie index** for mapping with A*	**Seed heuristic** (SH) for A*	**Inexact matching, chaining, match pruning** for SH
Scales	Sub-linear with reference	+Near-linear with read	+High error rate
Reference (in evals)	Linear (E. Coli), General graph (MHC) <5 Mbp		ONT read (<10 Mbp)
Queries (in evals)	Simulated Illumina (150 bp, ~3% errors)	+Simulated HiFi (<30 kbp, ~0.3% errors)	Simulated uniform errors (5-15%)
Tool	AStarix		A*PA
Compare	PaSGAL, BitParallel, vg	+Vargas	Edlib, BiWFA
Metric	Edit distance		Levenshtein distance
Guarantees	Minimizes edit distance		
Complete	Aligns all sequences		

FIGURE 0.2: Overview of the contributions by publication.

The optimal algorithms used in computational biology explore the search space of possible alignments in an uninformed fashion: by aligning a prefix of one sequence to a prefix of the other. This contrasts with the informed search algorithms such as the algorithm by Hunt & Szymanski [15] solving the longest common subsequence (LCS) problem (a special case of the edit distance alignment). Sequence alignment can naturally be formulated as a shortest path problem solvable by Dijkstra's algorithm [25]. A* is an *informed search* algorithm that generalizes Dijkstra's algorithm [7]. A* has not been successfully applied to sequence alignment but it may to "a major open problem to implement an algorithm with linear-like empirical scaling on inputs where the edit distance is linear in n" [95].

Alignment shifts towards pangenomics

The benefits of using graph references representing biological variation have been demonstrated to increase the alignment quality [73]. The transition towards graph references only aggravates the computational issues owing to the potentially complex graph topology [78]. The interest towards genome graphs keeps increasing with the first International Genome Graph Symposium being held this year in Ascona, Switzerland (2022).

CONTRIBUTIONS

In this thesis we introduce a novel principled approach for provably optimal and heuristically fast sequence alignment. The underlying idea of this thesis is to speed

up the search of an optimal alignment using information from the whole sequences, and not only from the aligned parts as existing algorithms do. We apply our approach to sequence-to-graph semi-global alignment (for read mapping) and global alignment. The question that we target is how general can the input sequences be while we could still efficiently compute provably-optimal alignments. To this end, we introduce novel algorithmic and data structure techniques, prove their correctness, and implement them as free and open source tools. Our evaluations demonstrate that remarkable performance scaling up to huge references, very long queries/sequences and high error rates. In absolute terms, we demontrate orders of magnitude of speedup compared to existing optimal algorithm on synthetic and real data.

Methods

All our methods we combine and build upon textbook informatics algorithms and data structures which have been known for many decades. These include the formulation of sequence alignment as a shortest path problem, the A* algorithm, the trie data structure, as well as trivial optimizations, such as hashing substring, greedily matching equal letters, topological sorting.

A* for sequence alignment. We consider the sequence alignment problem in its principled and powerful graph formulation: an alignment minimizing edit distance is equivalent to a shortest path in an *alignment graph*. It allows to choose any shortest path algorithm, and, assuming non-negative edit costs, we select the A* algorithm for its ability to use any available information to quickly direct the search and yet, to guarantee optimality. In order to efficiently apply A* to semi-global and global alignment, we complement the reference with a trie index, design a powerful *seed heuristic*, and implement a number of algorithmic optimizations. We formally prove the optimality of all algorithms, data structures, and optimizations.

Implicit constructiong of the alignment graph. The alignment graph is defined as a Cartesian product of the reference and the query. The structure of the alignment graph is thus regular, and we do not have to build it explicitly but to only constuct it locally at the node we are exploring. This optimization is crucial for the overall performance of A* which spends time only at explored nodes before terminating at the target.

Sequence-to-graph alignment. Unlike most dynamic programming solutions, we are not bound to acyclic graphs due to using the general A* shortest path algorithm. Thus, our reference is not limited to being a linear sequence but can as well be any genome graph. All semi-global alignment algorithms in this thesis are applicable to general graphs (possibly including cycles).

Scaling with reference size using a trie index. In Chapter 1 we suggest how to exploit that many query sequences are semi-globally aligned to the same reference. As a preprocessing step, we complement the reference with a trie index (similar to a suffix tree) so that any kmer from the reference appears as a path from the trie root to a trie leaf, and then links to the reference. This way, any substring in the trie is a spell of a path from the trie root. With an accurate heuristic function (i.e. estimating the remaining edit distance well), this trie complement allows to ignore most of the reference, setting the base for sublinear runtime scaling with the reference size.

Scaling with query length using seed heuristic. In Chapter 2 we introduce a powerful *seed heuristic* for A* for semi-global alignment. It estimates the remaining edit distance based on information from the whole reference and query, we prove its admissibility, and present an algorithm for its efficient computation even for the case of a trie index. We borrow the existing concept of seeds but apply it in a novel way: instead of searching for good alignments around seed matches, we negate the logic in order to prove that no alignment is better than the one we find. For each query, we do the following precomputation: split the query into non-overlapping seeds of equal length, find all exact matches of each seed in the reference, and mark all preceding reference nodes with *crumbs* that signify that follows a match of a specific seed. If A* explores a state, we can then compute the heuristic as the number of remaining seeds to be aligned that we do not see a crumb of – this is, each seed that has to be aligned but cannot be matched, will require at least 1 edit (assuming Levenshtein distance). Note that the lack of a match does not require any work but helps us penalize all mismatching seeds for all paths from all states. We call *potential* of the seed heuristic the maximal cost it can penalize (e.g. the number of seeds in case of Levenshtein distance), and we note that the potential grows linearly with the query length, providing the base for near-linear scaling with length. In order to precompute the crumbs for the seed heuristic in the case of a trie index over a potentially-cyclic graph reference, we introduce a linear algorithm based on topological sort.

Scaling with error rate using inexact matching, chaining and gap costs. In Chapter 3 on the task of global alignment we demonstrate how to extend the seed heuristic in order scale to high error rates. Our general *chaining seed heuristic* includes inexact matching, chaining, and gap costs. Allowing a single error to match a seed leads to the guarantee of requiring at least 2 edits to align that seed, which increases the seed heuristic potential twice. Chaining the seed matches exploits the order of the seeds in the query, which mitigates the effect of spurious matches. Accounting for the gap between consequitive matches in a chain allows to penalize more or longer insertions and deletions. The combination of these

extensions enables the efficient alignment of extremely long sequences with up to 30% errors.

Implementations. All presented algorithms are available as free and open source tools. AStarix[6] is our semi-global sequence-to-graph aligner, A*PA[7] is our global sequence-to-sequence aligner, MinSH[8] is a minimalistic implementation of the seed heuristic for global alignment. Here is a working 5-line implementation of the seed heuristic that captures its basic idea:

```
def build_seedh(A, B, k):
    seeds = [ A[i:i+k] for i in range(0, len(A)-k+1, k) ]
    kmers = { B[j:j+k] for j in range(len(B)-k+1) }
    is_seed_missing = [ s not in kmers for s in seeds ]
    suffix_sum = np.cumsum(is_seed_missing[::-1])[::-1]
    return lambda ij, k=k: suffix_sum[ ceildiv(ij[0], k) ]

h_seed = build_seed_heuristic(A, B, k=log(len(A)))
astar(A, B, h_seed)
```

Results

Since all our algorithms are provably correct, we only test the equivalence of the produced edit distance between in order to lower the risk of programming mistakes. Empirically, on a single-thread, we compare to existing optimal aligners the scaling, runtime and memory usage of our implementations: the semi-global aligner AStarix and the global aligner A*PA.

Worst case vs related sequence. Traditionally, the runtime and memory have been analysed for the worst case asymptotic behavior of the algorithm. Since the near quadratic worst case asymptotics is likely to be tight for both global [59] and semi-global alignment, we target related sequences with limited edit distance, and estimate their empiric runtime and memory scaling.

Setting. We demontrate that the additional information we use from the whole sequence can improve the scaling with query length, reference size and error rate, substantially decrease the necessary computations (by more than 99.99%), and result in algorithms that are orders of magnitude faster than existing optimal algorithms. Our empirical analysis of scaling is based on running the same algorithm on increasingly more complex input data (longer sequences, higher error rate), measuring its runtime and memory usage, and fitting a function to explain the correlation between the input parameter and the measurement. This approach is

6 https://github.com/eth-sri/astarix
7 https://github.com/RagnarGrootKoerkamp/astar-pairwise-aligner
8 https://github.com/pesho-ivanov/minSH

useful to estimate the scaling of the algorithm and its implementation. Note that these scaling analyses are not asymptotical, include noise, and we limit the fitting to only 1 parameter at a time.

Scaling with reference size. In Chapter 1 we present the tool AStarix which applies the A* algorithm to find optimal alignments, based on a simple heuristic and enhanced by multiple algorithmic optimizations. We demonstrate that using a trie index we can achieve sublinear runtime scaling with reference size, and that A* can scale 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.

Scaling with query length. In Chapter 2 we upgrade AStarix with a novel seed heuristic which guides the A* search by preferring crumbs on nodes that lead towards optimal alignments even for long reads. This approach enables the near-linear scaling of semi-global alignment with read length. On our linear and variation graph datasets, 99.99% of the states are skipped due to the accurate heuristic function. On both short reads and long low-error reads from linear and variant graph references, A* with seed heuristic consistently outperforms all state-of-the-art optimal aligners GraphAligner, Vargas, PaSGAL by more than 60 times.

Scaling with error rate. In Chapter 3 we resolve the third major bottleneck—handling high error rates. 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.

Seed heuristic modes of operation. The performance of A* with the seed heuristic follows one of two distinguishable patterns based on the capability of the seed heuristic to compensate for all the errors. If the heuristic potential is sufficiently higher than the resulting edit distance between the sequences, the heuristic punishes suboptimal paths very strongly, leading to a very directed A* exploration, which grows linear-like with the sequence length. Otherwise, if the error rate is too high, each unit of difference that the seed heuristic does not account for, is a unit of extended search to the side, leading to a quadratic-like exploration of the search space. Note that the total runtime may grow quadratically for other reasons too: due to preprocessing of dense seed matches (which is very

data dependent), or due to slow heuristic computation (which has not been a bottleneck for us). Interestingly, since the seed heuristic improves not from having a match but from punishing lacks of matches, no matches are required for the near-linear behavior of the seed heuristic.

1

SCALING WITH REFERENCE SIZE

In this chapter we consider the problem of semi-global alignment of a set of reads to a general graph reference. We supplement the reference graph with a trie index and run shortest path algorithms (Dijkstra and A* with a very simple heuristic) that start aligning from the trie root. We demonstrate that using a trie index enables sublinear scaling of the alignment time with the reference size. Even for moderate size references, we rearch orders of magnitude of speedup compared to other optimal algorithms.

1.1 OVERVIEW

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.

All optimal algorithms have to iterate over the whole reference for each query. We should be able to optimize this by precomputation.

Methods. 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.

Results. The main contributions presented in this chapter are:

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 (Sec. 1.3).
2. **Algorithmic optimizations.** To ensure that AStarix is practical, we introduce a number of algorithmic optimizations which increase performance and decrease memory footprint (Sec. 1.4). 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 (Sec. 1.5).

1.2 PRELIMINARIES

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. 1.1 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 [68, 73]. 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 $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ represents the four different nucleotides. Note that in contrast to sequence graphs [64], 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 [75] and Brownie [71] but missing from VG [73], PaSGAL [74] and V-ALIGN [81].

Alignment on Reference Graph. An *alignment* of *query* $q \in \Sigma^*$ to a reference graph $G_\mathrm{r} = (V_\mathrm{r}, E_\mathrm{r})$ 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 $0 \leq \Delta_{\text{match}} \leq \Delta_{\text{subst}}, \Delta_{\text{ins}}, \Delta_{\text{del}}$, which will be a prerequisite for the correctness of our approach.

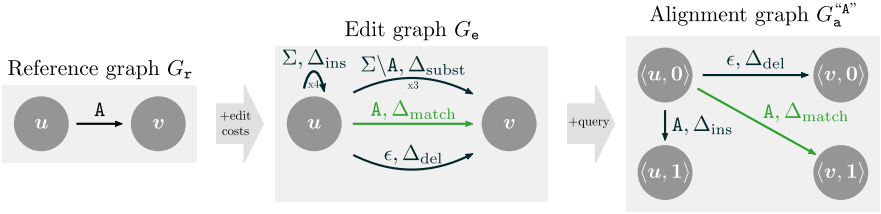


FIGURE 1.1: Starting from the reference graph (left), we can construct the edit graph (middle) and the alignment graph G for query $q =$ "A" (right). Edges are annotated with labels and/or costs, where sets of labels represent multiple edges, one for each letter in the set (indicated by "x3" and "x4").

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* $G := (V, E)$ has directed, labeled edges $E \subseteq V \times V \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ 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. 1.1). The edit and reference graphs consist of the same nodes, i.e., $V = V_r$. However, E contains more edges than E_r to account for edits. Concretely, for each edge $(u, v, \ell) \in E_r$, 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 $\ell' \in \Sigma \setminus \ell$, (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 $\pi \in G$ is defined analogously to reference graphs, except that deleted letters (represented by ϵ) are ignored. The cost cost π of a path $\pi \in G$ 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 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 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* $G = (V, E)$.

Here, each *state* $\langle v, i \rangle \in V$ consists of a node $v \in V$ and a query position $i \in \{0, \ldots, |q|\}$ (equivalent to [64]). Traversing a state $\langle v, i \rangle \in V$ 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 in Fig. 1.1 lacks substitution edges from G, as their labels (C, G, T) do not match the query $q = "A"$.

We construct the alignment graph G 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 of q to G and paths $\pi \in G$ from sources $S := V_\mathrm{r} \times \{0\}$ to targets $T := V_\mathrm{r} \times \{|q|\}$, with cost $\pi_\mathrm{r} = \operatorname{cost} \pi$. To find the best alignment in G, only paths in G (walks without repeating nodes) can be considered, since repeating a node in G cannot lead to a lower cost ($\Delta_\mathrm{del} \geq 0$) for the same state.

The edges $E \subseteq V \times V \times \Sigma_\epsilon \times \mathbb{R}_{\geq 0}$ are built based on the edges in 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:

$$(u, v, \ell, w) \in E \implies (\langle u, i\rangle, \langle v, i+1\rangle, \ell, w) \in E \quad \text{for } 0 \leq i < |q| \text{ with } q[i] = \ell \tag{1.1}$$

$$(u, v, \epsilon, w) \in E \implies (\langle u, i\rangle, \langle v, i\ \ \rangle, \epsilon, w) \in E \quad \text{for } 0 \leq i < |q| \tag{1.2}$$

Here, assuming 0-indexing, $q[i]$ is the next letter to be matched after matching i letters. Then, Eq. (1.1) represents matches, substitutions, and insertions (which advance the position in the query by 1), while Eq. (1.2) represents deletions (which do not advance the position in the query).

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

1.3 A* WITH A TRIE INDEX

In this section, we instantiate the A* algorithm with a simple admissible *prefix heuristic* and then we demonstrate how to supplement the reference with a trie index in order to avoid exploring the whole reference for each query alignment. We prove that that resulting alignment will be optimal.

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 ← DefineAlignmentGraph(G_r, q)                   ▷ Following Sec. 1.2
5:     S ← {⟨v, i⟩ ∈ V | i = 0}                           ▷ Sources: no letter matched
6:     T ← {⟨v, i⟩ ∈ V | i = |q|}                         ▷ Targets: all letters matched
7:     return A*(G, S, T, Heuristic)                      ▷ A* provided in Algorithm 5

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

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

1.3.1 *Simple prefix heuristic*

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. It is parameterized by hyper-parameters (d in Algorithm 1, more in Sec. 1.4) and edit costs (implicitly provided).

The function Heuristic (Algorithms 1 to 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 **RecursiveAlign**. It 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.

Algorithm 2 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 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
```

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 admissible 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 admissible.*

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

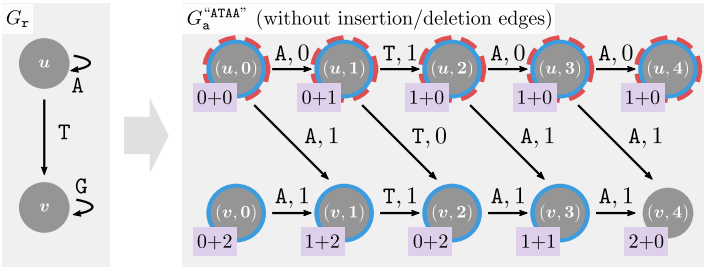


Figure 1.2: The benefit of using our heuristic over Dijkstra. Alignment graph G (right) is based on reference graph G_r (left), but omits insertion and deletion edges for simplicity. The pink boxes $g + h$ indicate the distance from the sources $S = \{\langle u, 0\rangle, \langle v, 0\rangle\}$ (in g) and the cost of aligning the next $d = 2$ letters (in h). Dijkstra (resp. A*) expands states circled in blue (resp. dashed red).

Benefit of A* heuristic over Dijkstra. Fig. 1.2 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. 1.2. 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.

1.3.2 *Trie index*

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 [74] and BITPARALLEL [75] 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 [72]) 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).

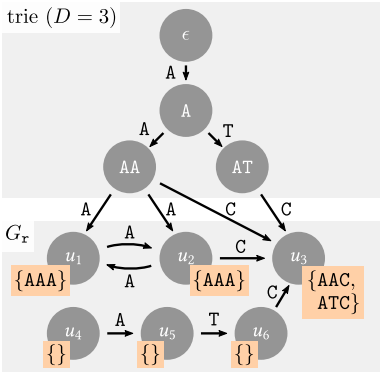


FIGURE 1.3: G_r^+ enables semi-global alignment by extending G_r with a trie.

Fig. 1.3 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. 1.3): 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:

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

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 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. 1.3, we (i) remove node AT including its edges $(\mathrm{A}, \mathrm{AT}, \mathrm{T})$ and $(\mathrm{AT}, u_3, \mathrm{C})$, but (ii) introduce an edge (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').

1.4 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). Our aligner AStarix handles both A* and Dijkstra algorithm (formally with $h \equiv 0$).

1.4.1 *Greedy match optimization*

We also employ an optimization originally developed for computing the edit distance between two strings [11, 33], but which has also been used in the context of string to graph alignment [72]. We omit the correctness proof of this optimization, which is already covered in [11], and only explain the intuition behind it.

Suppose there is only one outgoing edge $e = (u, v, \ell) \in E_{\mathrm{r}}$ from a node $u \in V_{\mathrm{r}}$. 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 (Sec. 1.3.2).

1.4.2 *Capping the cost and depth*

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 Sec. 1.5.1.

Theorem 2. *h_c is admissible.*

Proof. Follows from $h_c(u, s) \leq h(u, s)$ and $h(u, s)$ being admissible (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 $R_u \subseteq V$ at distance at most d from u (in Algorithm 1): $R_u := \{v \in V_{\mathrm{r}} \mid \exists \text{ path } \pi \in G_{\mathrm{r}} \text{ from } u \text{ to } v \text{ with } |\pi| \leq d\}$.

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

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

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

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

Theorem 3. *h_d is admissible.*

Proof. It suffices to show $h_d(u,s) \leq h(u,s)$ since $h(u,s)$ is admissible. 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 $h_d(u,s) \leq h(u,s)$. □

1.4.3 *Equivalence classes*

We have shown in Sec. 1.3.1 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 $r \in V_\mathrm{r}$ 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 $repr[u] = r$.

Identifying equivalence classes. To identify the nodes belonging to the same equivalence class, we assume the optimization from Sec. 1.4.2, 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.

1.4.4 *Data structures*

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 nodes are doubled, connected in the opposite direction, and labeled with complementary nucleotides (A $\leftrightarrow$ T, C $\leftrightarrow$ G). We do not limit the number of memoized heuristic function values (Sec. 1.3.1), but note we could do so by resetting the memoization table periodically.

1.5 EVALUATIONS

In this section we present a thorough experimental evaluation[1] of AStarix on simulated Illumina reads. Implementations and evaluations[2] demonstrate 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.

1.5.1 *Setting*

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

Input. Here we explain the used reference graphs, reads, and error costs.

We designed three experiments utilizing three different reference graphs (in Table 1.1). The first is a linear graph without variation based on the *E. coli* reference genome (strain: K-12 substr. MG1655, ASM584v2 [86]). The other two are variation graphs taken from the PaSGAL evaluations [74]: 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 [52] 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 [74] using the Mason2 [47] simulator.

We use costs corresponding to Illumina error profiles: $\Delta = (0, 1, 5, 5)$.

Metrics. As all aligners evaluated here 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 [54] 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.

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.

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

Comparison to optimal aligners. 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 [74].

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

Parameter tuning. While the optimality of AStarix is not affected by its parameters, its performance is. To compare with other aligners, we use values $d = 5$, $c = 5$, $D = \lfloor \log_\Sigma |G_r| \rfloor$. Follows an investigation of the influence of different parameters (c, d, D) on the runtime and memory usage.

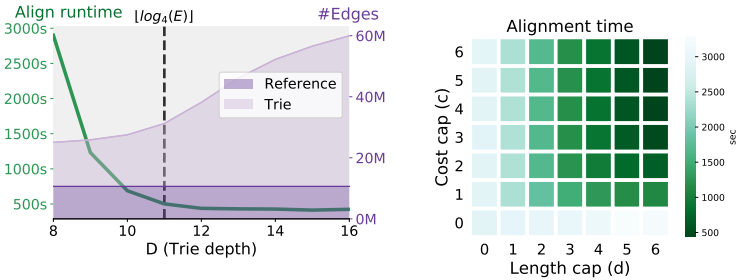


FIGURE 1.4: Left: Effect of the trie depth D on the performance of AStarix (MHC1 experiment). The dashed line shows our choice of D. Right: Runtime of AStarix depending on d and c (MHC1 experiment).

Sec. 1.5.1 demonstrates the benefit of using a trie with the size reduction optimization (end of Sec. 1.3.2): 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.

Sec. 1.5.1 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.

1.5.2 *Speedup on various references*

Table 1.1 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.

TABLE 1.1: 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 (graph)	1 Mbp	**437 sec** 1.12 GB	940 sec 1.09 GB	1 614 sec 0.30 GB	SegFault
MHC1 (graph)	5 Mbp	**1 282 sec** 4.35 GB	1 588 sec 1.21 GB	>7 200 sec 0.87 GB	SegFault

1.5.3 *Scaling with reference size*

Fig. 1.5 compares the performance of existing optimal aligners. BITPARALLEL and PASGAL always explore all states, thus their average-case reaches the worst-case complexity of $O(|G|) = O(m \cdot G_r)$. 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.

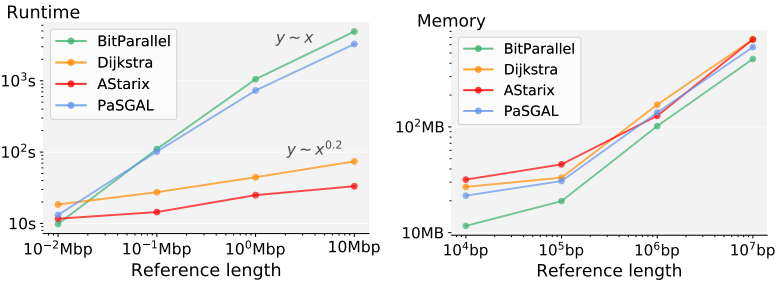


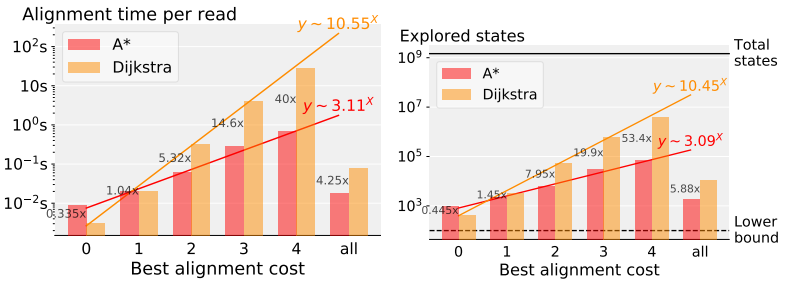
FIGURE 1.5: Comparison of overall runtime and memory usage of optimal aligners with increasing prefixes of E. coli as references

1.5.4 *Scaling with number of errors*

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, and the example in Fig. 1.2) between ASTARIX and Dijkstra on the MHC1 dataset.

Fig. 1.6 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. 1.6 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.



(A) (B)

FIGURE 1.6: Comparison of A* and Dijkstra in terms of mean alignment runtime per read and mean explored states depending on the alignment cost on MHC1.

1.6 ADDITIONAL DETAILS

1.6.1 *Generic Algorithms: A* and Dijkstra*

Algorithm 3 shows a generic implementation of the A* algorithm, roughly following [26]. 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 3 also shows a simple implementation of Dijkstra in terms of A*.

Algorithm 3 A* algorithm (generalizes Dijkstra)

```
1:  function A*(G: Graph, S: Sources, T: Targets, h: Heuristic function)
2:      f ← Map(default = ∞): Nodes → ℝ_{≥0}          ▷ Map nodes from G to priorities
3:      Q ← MinPriorityQueue(priority = f)            ▷ Priorities according to f
4:      for all s ∈ S do
5:          f[s] ← 0.0
6:          Q.push(s)                                 ▷ Initially, explore all s ∈ S
7:      while Q ≠ ∅ do
8:          curr ← Q.pop()                            ▷ Get state with minimal f to be expanded
9:          if curr ∈ T then
10:             return BacktrackPath(curr)            ▷ Reconstruct a path to curr (omitted)
11:         for all (curr, next, cost) ∈ G.outgoingEdges(curr) do
12:             f̂_next ← f[curr] + cost + h(next)     ▷ Candidate value for f[next]
13:             if f̂_next < f[next] then
14:                 f[next] ← f̂_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) ← 0.0                                    ▷ Constant-zero function h
19:     A*(G, S, T, h)
```

1.6.2 *Recursive Alignment Algorithm*

Algorithm 4 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.

Algorithm 4 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 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
```

1.6.3 *Parameter Estimation*

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

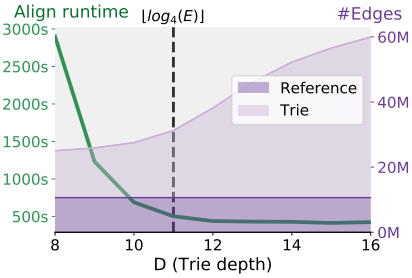


FIGURE 1.7: Effect of D on performance of AStarix (MHC1 experiment). The dashed line shows our choice of D.

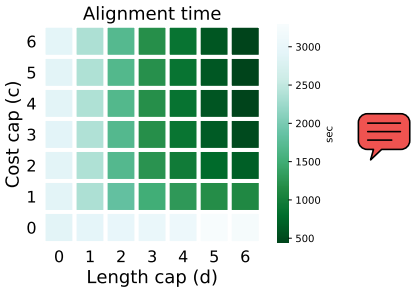


FIGURE 1.8: Runtime of AStarix depending on d and c (MHC1 experiment).

Fig. 1.7 demonstrates the benefit of using a trie with the size reduction optimization (end of Sec. 1.3.2): 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.

Fig. 1.8 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.

1.6.4 *Versions, commands, parameters*

In the following, we provide details on how we executed the approaches discussed in Sec. 1.5:

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`

2

SCALING WITH QUERY LENGTH

In this chapter we consider the same problem of semi-global alignment of a set of reads to a general graph reference. The trie index from Chapter 1 enables sublinear scaling with the reference size but each additional error triggers a deeper exploration of the trie, leading to exponential scaling. Now we equip the A* algorithm with a superior seed heuristic that is informed by substrings from the whole query length. This enables near-linear scaling with the error rate (as opposed the exponential scaling in Chapter 1), thus scaling to long reads with orders of magnitude of speedup compared to other optimal algorithms.

2.1 OVERVIEW

Contributions. Here 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.

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, 0.3% error rate), 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.

Overall, the contributions presented in this chapter include:

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.

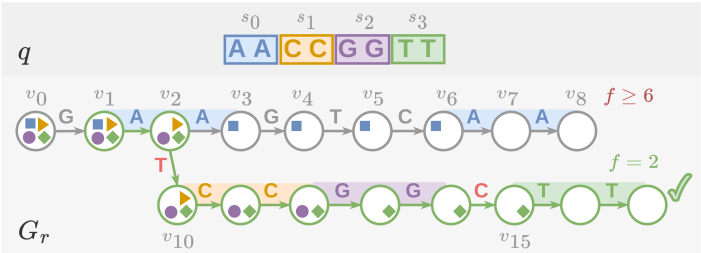


FIGURE 2.1: A toy illustration of the seed heuristic precomputation before aligning a read q to a reference graph $G_\mathbf{r}$. The read is split into four colored seeds, where their corresponding crumbs are shown inside reference graph nodes as symbols with matching color. The optimal alignment is highlighted as a green path ending with a tick (✓) and includes one substitution (T→A) and one deletion (C).

Intuition. Fig. 2.1 showcases the seed heuristic on an overview example. It shows a read q to be aligned to a reference graph $G_\mathbf{r}$. Our goal is to find an optimal alignment starting from an arbitrary node $v \in G_\mathbf{r}$. For simplicity of the exposition, we assume unit edit costs $\Delta = (0, 1, 1, 1)$, which we generalize in Sec. 2.3.1.

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. 2.1 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. 2.1 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 the cost of such an alignment: it never estimates a cost of more than 4, the number of seeds.

1 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.

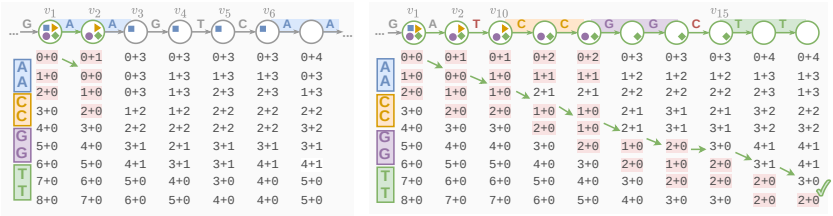


FIGURE 2.2: Exploration of G, searching for a shortest path from the first to the last row using the seed heuristic. The table entry in the i^{th} row (zero indexed) below node v shows $g\langle v,i\rangle + h\langle v,i\rangle$, where $g\langle v,i\rangle$ is the shortest distance from any starting state $\langle u,0\rangle$ to $\langle v,i\rangle$. States that may 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 (→).

Guiding the search example. Fig. 2.2 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. 2.1 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 Sec. 2.3.2 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.

2.2 PRELIMINARIES

We define the task of alignment as a shortest path problem (Sec. 2.2.1) to be solved using the A* algorithm (Sec. 2.2.2).

2.2.1 *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 [85] and is in line with [64].

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

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 $\sigma(\pi_r)$ to q. An alignment is *optimal* if it minimizes the sum of edit costs for a given real-valued cost model $\Delta = (\Delta_{match}, \Delta_{subst}, \Delta_{del}, \Delta_{ins})$. Throughout this work, we assume that edit costs are non-negative—a pre-requisite for the correctness of A*. Further, we assume that $\Delta_{match} \leq \Delta_{subst}, \Delta_{ins}, \Delta_{del}$—a prerequisite for the correctness of our heuristic.

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

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

FIGURE 2.3: Formal definition of alignment graph edges $E \subseteq V \times V \times \Sigma_\varepsilon \times \mathbb{R}_{\geq 0}$. Here, $u, v \in V_r$, $0 \leq i < |q|$, $\ell \in \Sigma$, and ε represents the empty string, indicating that letter ℓ was deleted.

Alignment graph. In order to formalize optimal alignment as a shortest path finding problem, we rely on an *alignment graph* $G = (V, E)$. Its nodes V are *states* of the form $\langle v, i\rangle$, where $v \in V_r$ is a node in the reference graph and $i \in \{0, \ldots, |q|\}$ corresponds to a position in the read q. Its edges E are selected such that any path π in G 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 path π in G from $\langle u, 0\rangle$ to $\langle v, |q|\rangle$, for any $u, v \in V_r$. Fig. 2.3 formally defines the edges E.

2.2.2 *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.

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.

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. 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.

2.3 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 (Sec. 2.1). Then, we formally define the heuristic and prove its admissibility (Sec. 2.3.1). Finally, we adapt our approach to rely on a trie, which leads to a critical speedup (Sec. 2.3.2).

2.3.1 *Seed heuristic with crumbs*

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.

Seeds. We split read $q \in \Sigma^*$ into a set $\mathcal{S}$ of non-overlapping seeds $s_0, \ldots, s_{|\mathcal{S}|-1} \in \Sigma^*$. For simplicity, we assume 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:

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

To compute $M(s)$ efficiently, we leverage the trie introduced in Sec. 2.3.2.

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

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

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 $misses\langle v, i\rangle := \left|\{s \in \mathcal{S}_{\geq i} \mid v \notin C(s)\}\right|$. Finally, we define the seed heuristic as

$$h\langle v, i\rangle = (|q| - i) \cdot \Delta_{\mathrm{match}} + misses\langle v, i\rangle \cdot \delta_{min}, \tag{2.1}$$
$$\text{for } \delta_{min} = \min(\Delta_{\mathrm{subst}} - \Delta_{\mathrm{match}}, \Delta_{\mathrm{del}}, \Delta_{\mathrm{ins}} - \Delta_{\mathrm{match}}), \tag{2.2}$$

Intuitively, Eq. (2.1) 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

$$n_{\text{del}} := \left\lceil \frac{|q| \cdot \Delta_{\text{match}} + |\mathcal{S}| \cdot \delta_{min}}{\Delta_{\text{del}}} \right\rceil . \tag{2.3}$$

In theorem 4, we show that $h\langle v, i\rangle$ is admissible, ensuring that our heuristic yields optimal alignments.

Theorem 4 (Admissibility). *The seed heuristic $h\langle v, i\rangle$ is admissible.*

Proof. We provide a proof for theorem 4 in Sec. 2.5.1. □

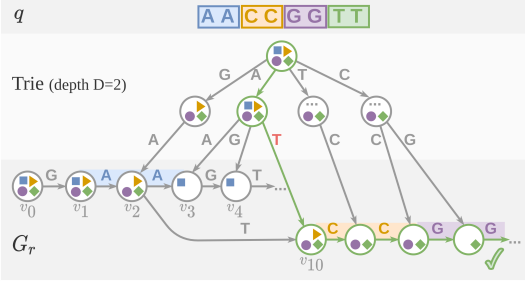


FIGURE 2.4: Reference graph from Fig. 2.1, extended by a trie of depth $D = 2$. For simplicity, the reverse-complement reference graph and parts marked by "…" are omitted.

2.3.2 *On a trie index*

Considering all nodes $v \in V_r$ 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 $|V_r|$. In line with the previous section, we avoid this overhead by complementing the reference graph with a trie index to produce a new graph $G_r^+ = (V_r^+, E_r^+)$, where V_r^+ is the union of the reference graph nodes V_r and the new trie nodes, 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. 2.4 extends the reference graph G_{r} from Fig. 2.1 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. 2.4). Thus, in order to find an optimal alignment, it suffices to consider paths starting from the trie *root*, by using state $\langle \mathit{root}, 0 \rangle$ 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 $\log_4 N$.

Importantly, our placement of crumbs (Sec. 2.3.1) generalizes directly to reference graphs extended with a trie (see also Fig. 2.4).

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 $i + n_{\mathrm{del}}$ edges (see Sec. 2.3.1). Here, we observe that it is sufficient to only climb into the trie from nodes u that can reach w using more than $i - n_{\mathrm{ins}} - D$ edges, for

$$n_{\mathrm{ins}} := \left\lceil \frac{|q| \cdot \Delta_{\mathrm{match}} + |\mathcal{S}| \cdot \delta_{min}}{\Delta_{\mathrm{ins}}} \right\rceil . \tag{2.4}$$

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 lemma 1 shows that this optimization preserves optimality.

Lemma 1 (Admissibility when skipping crumbs)**.** *The seed heuristic remains admissible when crumbs are skipped in the trie.*

Proof. We provide a proof for lemma 1 in Sec. 2.5.1. □

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.

2.3.3 *Implementation*

Both the seed heuristic and the prefix heuristic reuse the same free and open source C++ codebase of our AStarix aligner: https://github.com/eth-sri/astarix. 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 we call *greedy matching* (also known as *sliding*) 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.

2.4 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.

2.4.1 *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 Sec. 2.5.2. We note that we do not compare to VG [73] and HGA [88] since the optimal alignment functionality of VG is meant for debugging purposes and has been shown to be inferior to other aligners [88, Tab. 4], and HGA makes use of parallelization and a GPU but has been shown to be superseded in the single CPU regime [88, 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) [86], 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 [74]. 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 [49].

We generated long HiFi reads using the script RANDOMREADS.SH[2] 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[3].

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.

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

TABLE 2.1: 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.

	Illumina reads		**HiFi reads**		
Tool	*E. coli*	MHC	*E. coli*	MHC	
Seeds heuristic	**0.019**	**0.041**	**0.001**	**0.002**	s/kbp
	2.4	2.6	2.4	1.7	GB (max used)
	99.9996	99.9981	99.9989	99.9984	% skipped states
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	

2.4.2 *Speedup of the seed heuristic*

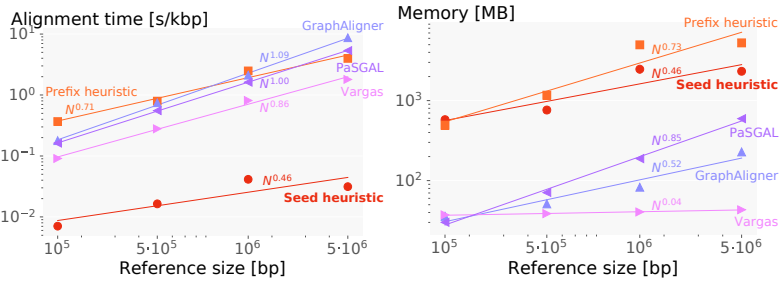
Table 2.1 shows that the seed heuristic achieves a speedup of at least 60 times compared to all considered aligners, across all regimes of operation: both Illumina and HiFi reads aligned on *E. coli* and MHC references.

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.

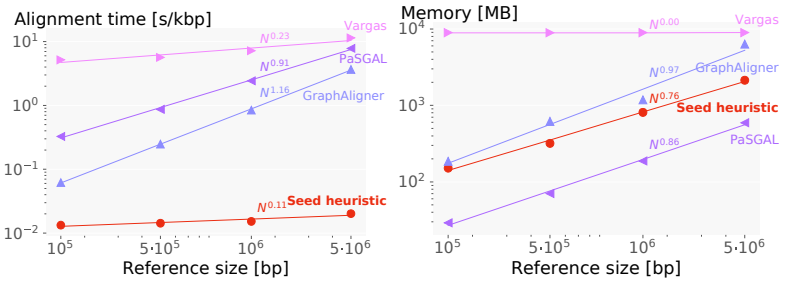
The prefix heuristic exceeded the available memory on HiFi reads, as it is not designed for long reads.

2.4.3 *Scaling with reference size*



(A) (B)

FIGURE 2.5: 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.



(A) (B)

FIGURE 2.6: 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.

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. 2.5 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 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. 2.6 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.

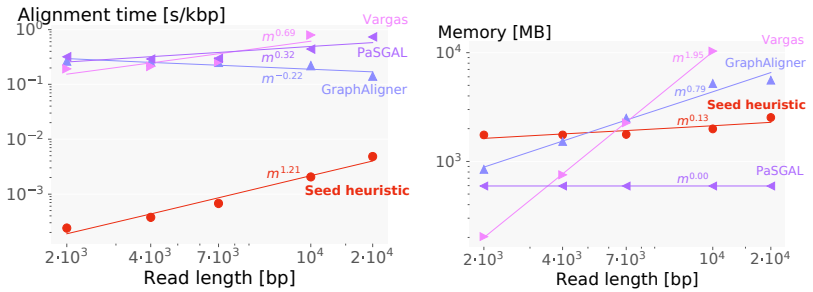


Figure 2.7: Performance degradation with HiFi read length. Log-log plots of total alignment time (left) and memory usage (right) show the scaling difference between aligners.

2.4.4 *Scaling with read length*

Fig. 2.7 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 Sec. 2.4.3), 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.

2.5 Additional Details

2.5.1 *Proofs*

In the following, we provide proofs for theorem 4 and lemma 1, restated here for convenience.

Theorem 4 (Admissibility). *The seed heuristic $h\langle v, i\rangle$ is admissible.*

Proof. Let A be an optimal alignment of $q[i{:}]$ starting from $v \in G_r$. 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 $n_{\text{del}} \cdot \Delta_{\text{del}}$, which is at least $|q| \cdot \Delta_{\text{match}} + |\mathcal{S}| \cdot \delta_{min}$ by plugging in n_{del} from Eq. (2.3). 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. (2.1), which concludes the proof in this case.

Otherwise, A contains less than n_{del} deletions. If we interpret A as a path in G, 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:

$$\begin{aligned}
\text{cost}(A) &= \sum_{j=-1}^{p} \text{cost}(A_k) && (2.5)\\
&\geq \sum_{j=-1}^{p} |\sigma(A_k)| \cdot \Delta_{\text{match}} + \sum_{j=0}^{p-1} \left\{ \begin{array}{ll} 0 & \text{if } v \in C\left(s_{\lceil i/k \rceil + j}\right) \\ \delta_{min} & \text{if } v \notin C\left(s_{\lceil i/k \rceil + j}\right) \end{array} \right\} && (2.6)\\
&= (|q| - i) \cdot \Delta_{\text{match}} + \left|\{v \notin C(s) \mid s \in \mathcal{S}_{\geq i}\}\right| \cdot \delta_{min} && (2.7)\\
&= h\langle v, i\rangle && (2.8)
\end{aligned}$$

Here, Eq. (2.5) follows from our decomposition of A. If we ignore the right-hand side in Eq. (2.6) (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 $i + n_{\text{del}}$ steps from v. However, because A contains less than n_{del} deletions, A_k must start within less than $i + n_{\text{del}}$ 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. (2.7) 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. (2.8) follows from our definition of $h\langle v, i \rangle$, concluding the proof. □

Lemma 1 (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. □

2.5.2 *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 Sec. 2.4:

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

Seed heuristic

Command
```
astarix align-optimal -D 14 -a astar-seeds
–seeds_len 1 -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

SCALING WITH ERROR RATE

The trie index from Chapter 1 enabled sublinear scaling with the reference size, and the seed heuristic from Chapter 2 additionally enabled long queries to be aligned. Nevertheless, when there are too many errors (more than 1% for long sequences), the seed heuristic may not be able to compensate for all of them, and the search has to start exporing in width (similar to Dijkstra).

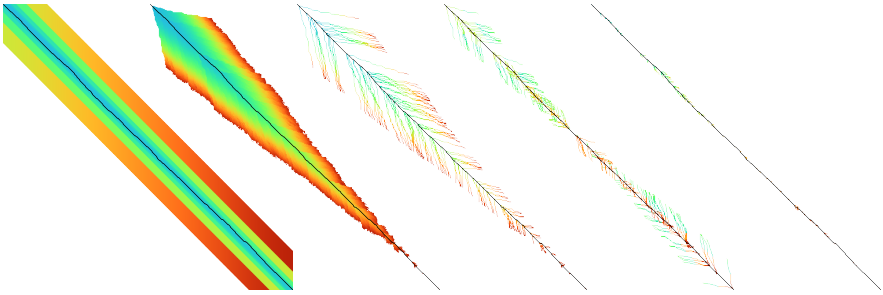
In this chapter we consider the more basic problem of global alignment between two sequences and demonstrate how to squeeze out more information from each seed by aligning it inexactly (with up to 1 edit), as well as from the connection between the seeds by chaining them in order in both sequences. As a result, we reach near-linear scaling to long sequence for error rates as high as 15% and orders of magnitude of speedup compared to other optimal aligners for 5% error rate.

This chapter is based on a work with joint first-authorship. The contributions of Pesho Ivnaov predominantly include the general seed heuristic with chaining, gaps and inexact matches, while the contributions of Ragnar Groot Koerkamp predominantly include match pruning, contours, DT, implementation, evaluations, and proofs.

3.1 INTRODUCTION

The problem of aligning one biological sequence to another is known as *global pairwise alignment* [37]. Among others, it is applied to genome assembly, read mapping, variant detection, and multiple sequence alignment [82]. Despite the centrality and age of pairwise alignment [9], "a major open problem is to implement an algorithm with linear-like empirical scaling on inputs where the edit distance is linear in n" [95].

Alignment accuracy affects subsequent analyses, so a common goal is to find a shortest sequence of edit operations (single letter insertions, deletions, and substitutions) that transforms one sequence into the other. The length of such a sequence is known as *Levenshtein distance* [6] and *edit distance*. It has recently been proven that edit distance can not be computed in strongly subquadratic time, unless SETH is false [59]. When the number of sequencing errors is proportional to the length, existing exact aligners scale quadratically both in the theoretical



(A) Band-doubling (Edlib) (B) Dijkstra (C) DT (WFA) (D) DT+D&C (BiWFA) (E) **This work (A*PA)**

Figure 3.1: **Computed states per algorithm.** Various optimal alignment algorithms and their implementation are demonstrated on synthetic data (length n=500 bp, divergence d=16%). The colour indicates the order of computation from blue to red. a Band-doubling (Edlib), b Dijkstra, c Diagonal transition/DT (WFA), d DT with divide-and-conquer/D&C (BiWFA), e A*PA with gap-chaining seed heuristic (GCSH), match pruning, and DT (seed length k=5 and exact matches).

worst case and in practice. Given the increasing amounts of biological data and increasing read lengths, this is a computational bottleneck [76].

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

3.1.1 *Related work*

Here we outline algorithms, tools, optimizations, and implementations for exact pairwise alignment of genetic sequences. Refer to Kucherov [76] for approximate, probabilistic, and affine-cost algorithms and aligners.

Dynamic programming (DP). This classic approach to aligning two sequences computes a table where each cell contains the edit distance between a prefix of the first sequence and a prefix of the second by reusing the solutions for shorter prefixes. This quadratic DP was introduced for speech signals Vintsyuk [8] and genetic sequences [9, 10, 12, 13]. The quadratic $O(nm)$ runtime for sequences of lengths n and m allowed for aligning of long sequences for the time but speeding

it up has been a central goal in later works. Implementations of this algorithm include SeqAn [70] and Parasail [63].

Band doubling and bit-parallelization. When the aligned sequences are similar, the whole DP table does not need to be computed. One such output-sensitive algorithm is the *band doubling* algorithm of Ukkonen [25] (Fig. 3.1a) which considers only states around the main diagonal of the table, in a *band* with exponentially increasing width, leading to $O(ns)$ runtime, where s is the edit distance between the sequences. This algorithm, combined with the *bit-parallel optimization* by Myers [35] is implemented in Edlib [66] with $O(ns/w)$ runtime, where w is the machine word size (nowadays 64).

Diagonal transition (DT). The *diagonal transition* algorithm [25, 27] exploits the observation that the edit distance does not decrease along diagonals of the DP matrix. This allows for an equivalent representation of the DP table based on *farthest-reaching states* for a given edit distance along each diagonal. Diagonal transition has an $O(ns)$ worst-case runtime but only takes expected $O(n+s^2)$ time (Fig. 3.1c) for random input sequences [27] (which is still quadratic for a fixed divergence $d = s/n$). It has been extended to linear and affine costs in the *wavefront alignment* (WFA) [90] in a way similar to Gotoh [21]. Its memory usage has been improved to linear in BiWFA [96] by combining it with the divide-and-conquer approach of Hirschberg [14], similar to Myers [27] for unit edit costs.

Contours. The longest common subsequence (LCS) problem is a special case of edit distance, in which gaps are allowed but substitutions are forbidden. *Contours* partition the state-space into regions with the same remaining answer of the LCS subtask (Fig. 3.3). The contours can be computed in log-linear time in the number of matching elements between the two sequences which is practical for large alphabets [15, 16].

Shortest paths and heuristic search using A*. A pairwise alignment that minimizes edit distance corresponds to a shortest path in the *alignment graph* [8, 25]. Assuming non-negative edit costs, a shortest path can be found using Dijkstra's algorithm [25] (Fig. 3.1b) or A* [7, 29]. A* is an informed search algorithm which uses a task-specific heuristic function to direct its search. Depending on the heuristic function, a shortest path may be found significantly faster than by an uninformed search such as Dijkstra's algorithm.

Gap cost. One common heuristic function to speed up alignments is the *gap cost* [25, 29, 34, 30, 44] This is a lower bound on the remaining edit distance and counts the minimal number of indels needed to align the suffixes of two sequences.

Seed-and-extend. *Seed-and-extend* is a commonly used paradigm for approximately solving semi-global alignment by first matching similar regions between sequences (*seeding*) to find *matches* (also called *anchors*), followed by *extending*

these matches [76]. Aligning long reads requires the additional step of chaining the seed matches (*seed-chain-extend*). Seeds have also been used to solve the LCSk generalization of LCS [57, 67]. Except for the seed heuristic [92], most seeding approaches seek for seeds with accurate long matches.

Seed heuristic. A* with *seed heuristic* is an exact algorithm that was recently introduced for exact semi-global sequence-to-graph alignment [92]. In a precomputation step, the query sequence is split into non-overlapping *seeds* each of which is matched exactly to the reference. When A* explores a new state, the admissible seed heuristic is computed as the number of remaining seeds that cannot be matched in the upcoming reference. A limitation of the existing seed heuristic is that it only closely approximates the edit distance for very similar sequences (divergence between 0.3% and 4%).

3.1.2 *Contributions*

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

Chaining seed heuristic. The *seed heuristic* [92] penalizes missing seed matches. Our chaining seed heuristic requires matches to be chained in order, which reduces the negative effect of spurious (off-track) matches [24, 62], and improves performance for highly divergent sequences. To handle long indels, we introduce the *gap-chaining seed heuristic* which uses a *gap cost* for indels between consecutive matches.

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

Match pruning. In order to further improve our heuristics, we apply the *multiple-path pruning* observation of Poole & Mackworth [65]: once a shortest path to a node has been found, no other paths through this node can improve the global shortest path. When A* expands the start or end of a match, we remove (*prune*) this match from further consideration, thus improving (increasing) the heuristic for the states preceeding the match. The incremental nature of our heuristic search is remotely related to Real-Time Adaptive A* [40]. Match pruning enables near-linear runtime scaling with length (Fig. 3.1e).

Diagonal transition A*. We speed up our algorithm by combining it with the diagonal transition optimization that skips states where a farther state along the same diagonal has already been reached with the same distance (Fig. 3.1c).

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

Scaling and performance. We compare the absolute runtime, memory, and runtime scaling of A*PA to other exact aligners on synthetic data, consisting of random genetic sequences (length $n \leq 10^7\,bp$, uniform divergence $d \leq 12\%$). For $d=4\%$ and $n=10^7\,bp$, A*PA is up to $300\times$ faster than the fastest aligners Edlib [66] and BiWFA [96]. Our real dataset experiments involve $>500\,kbp$ long Oxford Nanopore (ONT) reads with $d<19.8\%$. When only sequencing errors are present, A*PA is $2\times$ faster (in median) than the second fastest, and when genetic variation is also present, it is $1.4\times$ faster.

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

Alignment graph. Let *state* $\langle i,j\rangle$ denote the subtask of aligning the prefix $A_{<i}$ to the prefix $B_{<j}$. The *alignment graph* (also called *edit graph*) $G(V,E)$ is a weighted directed graph with nodes $V = \{\langle i,j\rangle \mid 0 \leq i \leq n, 0 \leq j \leq m\}$ corresponding to all states, and edges connecting subtasks: edge $\langle i,j\rangle \to \langle i{+}1, j{+}1\rangle$ has cost 0 if $a_i = b_j$ (match) and 1 otherwise (substitution), and edges $\langle i,j\rangle \to \langle i{+}1, j\rangle$ (deletion) and $\langle i,j\rangle \to \langle i, j{+}1\rangle$ (insertion) have cost 1. We denote the starting state $\langle 0,0\rangle$ by v_s, the target state $\langle n,m\rangle$ by v_t, and the distance between states u and v by $\mathrm{d}(u,v)$. For brevity we write $f\langle i,j\rangle$ instead of $f(\langle i,j\rangle)$.

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

Seeds and matches. We split the sequence A into a set of consecutive non-overlapping substrings (*seeds*) $\mathcal{S} = \{s_0, s_1, s_2, \ldots, s_{\lfloor n/k \rfloor - 1}\}$, such that each seed $s_l = A_{lk \ldots lk+k}$ has length k. After aligning the first i letters of A, our heuristics will only depend on the *remaining* seeds $\mathcal{S}_{\geq i} := \{s_l \in \mathcal{S} \mid lk \geq i\}$ contained in the suffix $A_{\geq i}$. We denote the set of seeds between $u = \langle i, j \rangle$ and $v = \langle i', j' \rangle$ by $\mathcal{S}_{u \ldots v} = \mathcal{S}_{i \ldots i'} = \{s_l \in \mathcal{S} \mid i \leq lk,\ lk + k \leq i'\}$ and an *alignment* of s to a subsequence of B by π_s. The alignments of seed s with sufficiently low cost (Sec. 3.3.2) form the set $\mathcal{M}_s$ of *matches*.

Dijkstra and A*. Dijkstra's algorithm [4] finds a shortest path from v_s to v_t by *expanding* (generating all successors) nodes in order of increasing distance $g^*(u)$ from the start. Each node to be expanded is chosen from a set of *open* nodes. The A* algorithm [7, 23], instead directs the search towards a target by expanding nodes 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* (is *more accurate* than) another heuristic h_2 when $h_1(u) \geq h_2(u)$ for all nodes u. A dominant heuristic will usually, but not always [46], expand less nodes. 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 Sec. 3.6.1.

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

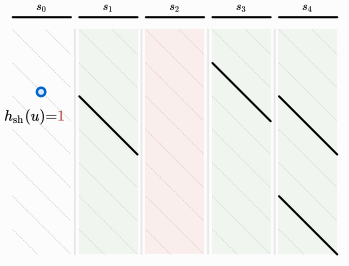
Gap cost. The number of indels to align substrings $A_{i \ldots i'}$ and $B_{j \ldots j'}$ is at least their difference in length: $c_{\text{gap}}(\langle i, j \rangle, \langle i', j' \rangle) := |(i' - i) - (j' - j)|$. For $u \preceq v \preceq w$, the gap cost satisfies the triangle inequalities $c_{\text{gap}}(u,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

Contours. To efficiently calculate maximal chains of matches, *contours* are used. Given a set of matches $\mathcal{M}$, $S(u)$ is the number of matches in the longest chain $u \preceq m_0 \preceq \ldots$, starting at u. The function $S\langle i, j \rangle$ is non-increasing in both i and j. *Contours* are the boundaries between regions of states with $S(u) = \ell$ and $S(u) < \ell$ (Fig. 3.3). Note that contour ℓ is completely determined by the set of matches $m \in \mathcal{M}$ for which $S(\text{start}(m)) = \ell$ [16]. Hunt & Szymanski [15] give an algorithm to efficiently compute S when $\mathcal{M}$ is the set of single-letter matches

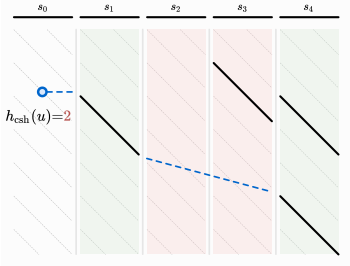
between A and B, and Deorowicz & Grabowski [55] give an algorithm when $\mathcal{M}$ is the set of k-mer exact matches.

3.3 METHODS

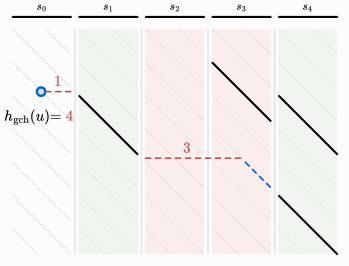
We intuitively motivate our A* heuristics and its improvements in Sec. 3.3.1. Then we formally define the general chaining seed heuristic (Sec. 3.3.2) that encompases *inexact matches*, *chaining*, and *gap costs* (Fig. 3.2). Next, we introduce the *match pruning* (Sec. 3.3.3) improvement and integrate our A* algorithm with the *diagonal-transition* optimization (Sec. 3.6.2). We present a practical algorithm (Sec. 3.3.4), implementation (Sec. 3.6.3) and proofs of correctness (Sec. 3.7).



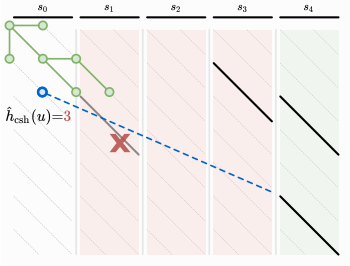
(A) Seed heuristic



(B) Chaining seed heuristic



(C) Gap-chaining seed heuristic



(D) CSH + match pruning

FIGURE 3.2: **Demonstration of seed heuristic, chaining seed heuristic, gap-chaining seed heuristic, and match pruning.** Sequence A is split into 5 seeds (horizontal black segments _) on top. Each seed is exactly matched in B (diagonal black segments \). The heuristic is evaluated at state u (blue circles ○), based on the 4 remaining seeds. The heuristic value is based on a maximal chain of matches (green columns ▮ for seeds with matches; red columns ▮ otherwise). Dashed lines denote chaining of matches. a The seed heuristic $h_{\text{s}}(u)=1$ is the number of remaining seeds that do not have matches (only s_2). b The chaining seed heuristic $h_{\text{cs}}(u)=2$ is the number of remaining seeds without a match (s_2 and s_3) on a path going only down and to the right containing a maximal number of matches. c The gap-chaining seed heuristic $h_{\text{gcs}}(u)=4$ is minimal cost of a chain, where the cost of joining two matches is the maximum of the number of not matched seeds and the gap cost between them. Red dashed lines denote gap costs. d Once the start or end of a match is expanded (green circles ○), the match is *pruned* (red cross x), and future computations of the heuristic ignore it. s_1 is removed from the maximum chain of matches starting at u so $\hat{h}_{\text{cs}}(u)$ increases by 1.

3.3.1 *Intuition*

To align sequences A and B with minimal cost, we use the A* shortest path algorithm from the start to the end of the alignment graph (Sec. 3.8.9). A* uses a heuristic function that estimates the shortest distance from the current node to the target. To ensure that A* finds a shortest path, the heuristic must be *admissible*, i.e. never overestimate the distance. A good heuristic efficiently computes an accurate estimate of the remaining distance.

Seed heuristic (SH). To define the *seed heuristic* h_s, we split A into short, non-overlapping substrings (*seeds*) of fixed length k. Since the whole sequence A has to be aligned, each of the seeds also has to be aligned somewhere in B. If a seed does not match anywhere in B without mistakes, then at least 1 edit has to be made to align it. To this end, we precompute for each seed from A, all positions in B where it matches exactly. Then, the SH is the number of remaining seeds (contained in $A_{\geq i}$) that do not match anywhere in B (Fig. 3.2a), and is a lower bound on the distance between the remaining suffixes $A_{\geq i}$ and $B_{\geq j}$. Next, we improve the seed heuristic with chaining, inexact matching, and gap costs.

Chaining (CSH). We improve on the SH by enforcing that the matches occur in the same order in B as their corresponding seeds occur in A, i.e., the matches form a *chain* going down and to the right. Now, the number of upcoming errors is at least the minimal number of remaining seeds that cannot be aligned on a single chain to the target. To compute this number efficiently, we instead count the maximal number of matches in a chain from the current state, and subtract that from the number of remaining seeds (Fig. 3.2b). This improves the seed heuristic when there are many spurious matches (i.e. outside the optimal alignment).

Gap costs (GCSH). In order to penalize long indels, the gap-chaining seed heuristic h_{gcs} (Fig. 3.2c) extends chaining seed heuristic by also incurring a *gap cost* if two consecutive matches in a chain do not lie on the same diagonal. In such cases, the transition to the next match requires at least as many gaps as the difference in length of the two substrings.

Inexact matches. To scale to higher divergence, we use *inexact matches*. For each seed in A, our algorithm finds all its inexact matches in B with cost at most 1. Then, not using any match of a seed will require at least $r{=}2$ edits for the seed to be eventually aligned. Thus, the *potential* of our heuristics to penalize errors roughly doubles.

Pruning. We introduce the *match pruning* improvement (Fig. 3.2d) for all our heuristics: once we find a shortest path to a state v, no shorter path to that state is possible. Since we are only looking for a single shortest path, we can ignore all alternative paths to v. If v is the start or end of a match and v has been expanded,

we are not interested in alternative paths using this match, and hence we simply ignore (*prune*) this match when evaluating a heuristic at states preceding v. Pruning a match increases the heuristic in states before the match, thus penalizing states preceding the "tip" of the A* search. Pruning significantly reduces the number of expanded states, and leads to near-linear scaling with the sequence length.

Diagonal transition (DT). The DT optimization (Sec. 3.6.2) only computes a subset of *farthest reaching* states. This speeds up the A* in cases where the search becomes quadratic (Fig. 3.1c).

Finally, we prove correctness of all heuristics and optimizations, and in Sec. 3.3.4 we show how to compute the heuristics efficiently.

3.3.2 *General chaining seed heuristic*

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

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

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

Chaining matches. Each heuristic depends on a *partial order* on states that limits how matches can be *chained*. We write $u \preceq_p v$ for the partial order implied by a function p: $p(u) \preceq p(v)$. A $\preceq_p$*-chain* is a sequence of matches $m_1 \preceq_p \cdots \preceq_p m_l$ that precede each other: $\mathrm{end}(m_i) \preceq_p \mathrm{start}(m_{i+1})$ for $1 \leq i < l$. To chain matches

according only to their i-coordinate, SH is defined using $\preceq_i$-chains, while CSH and GCSH are defined using $\preceq$ that compares both i and j.

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

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

For GCSH, we also include the gap cost $c_{\text{gap}}(\langle i,j\rangle, \langle i',j'\rangle) := |(i'-i)-(j'-j)|$ which is the minimal number of indels needed to correct for the difference in length between the substrings $A_{i\ldots i'}$ and $B_{j\ldots j'}$ between two consecutive matches (Sec. 3.2). Combining the seed cost and the chaining cost, we obtain the gap-seed cost $c_{\text{gs}} = \max(c_{\text{seed}}, c_{\text{gap}})$, which is capable of penalizing long indels and we use for GCSH. Note that $\gamma{=}c_{\text{seed}}{+}c_{\text{gap}}$ would not give an admissible heuristic since indels could be counted twice, in both c_{seed} and c_{gap}.

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

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

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

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

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

TABLE 3.1: **Definitions of our heuristic functions.** SH orders the matches by i and uses only the seed cost. CSH orders the matches by both i and j. GCSH additionally exploits the gap cost.

We instantiate our heuristics according to Table 3.1. Our admissibility proofs (Sec. 3.7.1) are based on $\mathrm{c_m}$ and γ being lower bounds on disjoint parts of the remaining path.

Since the more complex h_{gcs} dominates the other heuristics it usually expand fewer states.

Theorem 5. *The seed heuristic h_{s}, the chaining seed heuristic h_{cs}, and the gap-chaining seed heuristic h_{gcs} are* admissible. *Furthermore, $h_{\text{s}}^{\mathcal{M}}(u) \leq h_{\text{cs}}^{\mathcal{M}}(u) \leq h_{\text{gcs}}^{\mathcal{M}}(u)$ for all states u.*

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

3.3.3 *Match pruning*

In order to reduce the number of states expanded by the A* algorithm, we apply the *multiple-path pruning* observation: once a shortest path to a state has been found, no other path to this state could possibly improve the global shortest path [65]. As soon as A* expands the start or end of a match, we *prune* it, so that the heuristic in preceding states no longer benefits from the match, and they get deprioritized by A*. We define *pruned* variants of all our heuristics that ignore pruned matches:

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

The hat over the heuristic function ($\hat{h}$) denotes the implicit dependency on the progress of the A*, where at each step a different $h^{\mathcal{M} \setminus E}$ is used. Our modified A* algorithm (Sec. 3.6.1) works for pruning heuristics by ensuring that the f-value of a state is up-to-date before expanding it, and otherwise *reorders* it in the priority queue. Even though match pruning violates the admissibility of our heuristics for some nodes, we prove that A* is sill guaranteed to find a shortest path (Sec. 3.7.2). To this end, we show that our pruning heuristics are *weakly-admissible heuristics* (Def. 7) in the sense that they are admissible on at least one path from v_s to v_t.

Theorem 6. *A* with a weakly-admissible heuristic finds a shortest path.*

Theorem 7. *The pruning heuristics $\hat{h}_{\text{s}}$, $\hat{h}_{\text{cs}}$, $\hat{h}_{\text{gcs}}$ are weakly admissible.*

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

3.3.4 *Computing the heuristic*

We present an algorithm to efficiently compute our heuristics (pseudocode in Sec. 3.6.4). At a high level, we rephrase the minimization of costs (over paths) to a maximization of *scores* (over chains of matches). We initialize the heuristic by precomputing all seeds, matches, potentials and a *contours* data structure used to compute the maximum number of matches on a chain. During the A* search, the heuristic is evaluated in all explored states, and the contours are updated whenever a match gets pruned.

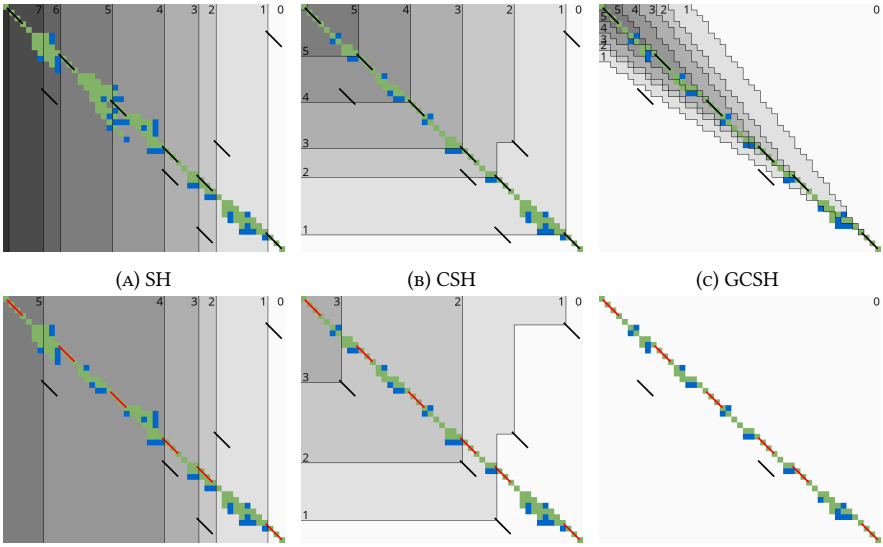
Scores. The *score of a match* m is $\mathrm{score}(m) := r - \mathrm{c_m}(m)$ and is always positive. The *score of a* $\preceq_p$*-chain* $m_1 \preceq_p \cdots \preceq_p m_l$ is the sum of the scores of the matches in the chain. We define the chain score of a match m as

$$S_p(m) := \max_{m \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t} \{\mathrm{score}(m) + \cdots + \mathrm{score}(m_l)\}. \tag{3.1}$$

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

$$S_p(m) = \mathrm{score}(m) + \max_{m \preceq_p m' \preceq v_t} S_p(m'). \tag{3.2}$$

We also define the chain score of a state u as the maximum chain score over succeeding matches m: $S_p(u) = \max_{u \preceq_p m \preceq_p v_t} S_p(m)$, so that Eq. (3.2) can be rewritten as $S_p(m) = \mathrm{score}(m) + S_p(\mathrm{end}(m))$.



(A) SH (B) CSH (C) GCSH

(D) SH + pruning (E) CSH + pruning (F) GCSH + pruning

FIGURE 3.3: **Contours and layers of different heuristics after aligning** (n=48, m=42, r=1, k=3, edit distance 10). Exact matches are black diagonal segments (➘). The background colour indicates $S_p(u)$, the maximum number of matches on a $\preceq_p$-chain from u to the end starting, with $S_p(u) = 0$ in white. The thin black boundaries of these regions are *Contours*. The states of layer $\mathcal{L}_\ell$ *precede* contour ℓ. Expanded states are green (■), open states blue (■), and pruned matches red (➘). Pruning matches changes the contours and layers. GCSH ignores matches $m \not\preceq_T v_t$.

The following theorem allows us to rephrase the heuristic in terms of potentials and scores for heuristics that use γ=c_{seed} and respect the order of the seeds, which is the case for h_{s} and h_{cs} (proof in Sec. 3.7.3):

Theorem 8. $h^{\mathcal{M}}_{p,c_{\text{seed}}}(u) = P(u) - S_p(u)$ *for any partial order* $\preceq_p$ *that is a refinement of* $\preceq_i$ *(i.e.* $u \preceq_p v$ *must imply* $u \preceq_i v$*).*

Layers and contours. We compute h_{s} and h_{cs} efficiently using *contours*. Let *layer* $\mathcal{L}_\ell$ be the set of states u with score $S_p(u) \geq \ell$, so that $\mathcal{L}_\ell \subseteq \mathcal{L}_{\ell-1}$. The ℓth *contour* is the boundary of $\mathcal{L}_\ell$ (Fig. 3.3). Layer $\mathcal{L}_\ell$ ($\ell > 0$) contains exactly those states that precede a match m with score $\ell \leq S_p(m) < \ell + r$ (lemma 6 in Sec. 3.7.3).

Computing $S_p(u)$. This last observation inspires our algorithm for computing chain scores. For each layer $\mathcal{L}_\ell$, we store the set $L[i]$ of matches having score ℓ:

$L[\ell] = \{m \in \mathcal{M} \mid S_p(m) = \ell\}$. The score $S_p(u)$ is then the highest ℓ such that layer $L[\ell]$ contains a match m reachable from u ($u \preceq_p m$). From lemma 6 we know that $S_p(u) \geq \ell$ if and only if one of the layers $L[\ell']$ for $\ell' \in [\ell, \ell + r)$ contains a match preceded by u. We use this to compute $S_p(u)$ using a binary search over the layers ℓ. We initialize $L[0]=\{m_\omega\}$ (m_ω is a fictive match at the target v_t), sort all matches in $\mathcal{M}$ by $\preceq_p$, and process them in decreasing order (from the target to the start). After computing $S_p(\text{end}(m))$, we add m to layer $S_p(m) = \text{score}(m) + S_p(\text{end}(m))$. Matches that do not precede the target ($\text{start}(m) \not\preceq_p m_\omega$) are ignored.

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

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

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

GCSH. theorem 8 does not apply to gap-chaining seed heuristic since it uses chaining cost $\gamma = \max(c_{\text{gap}}(u, v), c_{\text{seed}}(u, v))$ which is different from $c_{\text{seed}}(u, v)$. It turns out that in this new setting it is never optimal to chain two matches if the gap cost between them is higher than the seed cost. Intuitively, it is better to miss a match than to incur additional gapcost to include it. We capture this constraint by introducing a transformation T such that $u \preceq_T v$ holds if and only if $c_{\text{seed}}(u, v) \geq c_{\text{gap}}(u, v)$, as shown in Sec. 3.7.4. Using an additional *consistency* constraint on the set of matches we can compute $h^{\mathcal{M}}_{\text{gcs}}$ via S_T as before.

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

This condition means that for $r=2$, each exact match must be adjacent to four (or less around the edges of the graph) inexact matches starting or ending in the same state. Since we find all matches m with $\text{c}_\text{m}(m) < r$, our initial set of matches

is consistent. To preserve consistency, we do not prune matches if that would break the consistency of $\mathcal{M}$.

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

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

Theorem 9. *Given a* consistent *set of matches $\mathcal{M}$, the gap-chaining seed heuristic can be computed using scores in the transformed domain:*

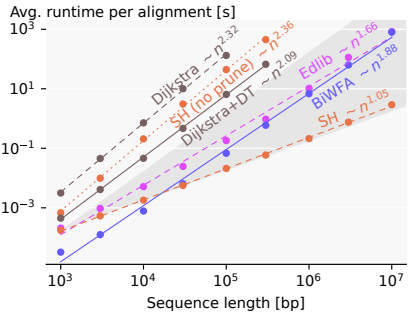
$$h^{\mathcal{M}}_{\text{gcs}}(u) = \begin{cases} P(u) - S_T(u) & \text{if } u \preceq_T v_t, \\ c_{\text{gap}}(u, v_t) & \text{if } u \not\preceq_T v_t. \end{cases}$$

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

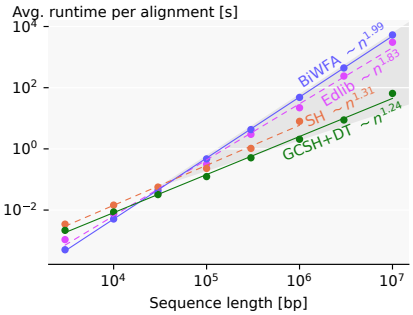
3.4 RESULTS

Our algorithm is implemented in the aligner A*PA[1] in Rust. We compare it with state of the art exact aligners on synthetic (Sec. 3.4.2) and human (Sec. 3.4.3) data[2] using PABENCH[3]. We justify our heuristics and optimizations by comparing their scaling and performance (Sec. 3.4.4).

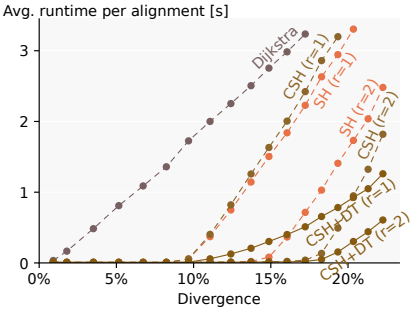
1 https://github.com/RagnarGrootKoerkamp/astar-pairwise-aligner/tree/1e3841
2 https://github.com/pairwise-alignment/pa-bench/releases/tag/datasets
3 https://github.com/pairwise-alignment/pa-bench/commit/55db710



(A) Runtime scaling with length (d=4%, r=1)



(B) Runtime scaling with length (d=12%, r=2)



(C) Runtime scaling with divergence

FIGURE 3.4: **Runtime comparison on synthetic data** ab Log-log plots comparing our simplest (SH) and most accurate heuristic (GCSH with DT) to EDLIB, BIWFA, and other algorithms (averaged over 10^6 to 10^7 total bp, seed length k=15). The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. SH without pruning is dotted, and variants with DT are solid. At 4% divergence, the complex techniques CSH, GCSH, and DT (not shown) are as fast as SH. Missing data points are due to exceeding the 32 *GiB* memory limit. c Runtime scaling with divergence (n=10^4, 10^6 total bp, k=10). GCSH (not shown) is as fast as CSH.

3.4.1 *Setup*

Synthetic data. Our synthetic datasets are parameterized by sequence length n, induced error rate e, and total number of basepairs N, resulting in N/n sequence pairs. The first sequence in each pair is uniform-random from Σ^n. The second is generated by sequentially applying $\lfloor e \cdot n \rfloor$ edit operations (insertions, deletions, and substitutions with equal 1/3 probability) to the first sequence. Introduced errors can cancel each other, making the *divergence* d between the sequences less

than e. Induced error rates of 1%, 5%, 10%, and 15% correspond to divergences of 0.9%, 4.3%, 8.2%, and 11.7%, which we refer to as 1%, 4%, 8%, and 12%.

Human data. We use two datasetsof ultra-long Oxford Nanopore Technologies (ONT) reads of the human genome: one without and one with genetic variation. All reads are 500–1100 kbp long, with mean divergence around 7%. The average length of the longest gap in the alignment is 0.1 kbp for ONT reads, and 2 kbp for ONT reads with genetic variation (detailed statistics in Table 3.2). The reference genome is CHM13 (v1.1) [93]. The reads used for each dataset are:

- *ONT*: 50 reads sampled from those used to assemble CHM13.
- *ONT with genetic variation*: 48 reads from another human [83], as used in the BiWFA paper [96].

Algorithms and aligners. We compare SH, CSH, and GCSH as implemented in A*PA to the state-of-the-art exact aligners BiWFA and Edlib. We also compare to Dijkstra and to variants without pruning, and without diagonal transition. We exclude SeqAn and Parasail since they are outperformed by BiWFA and Edlib [90].

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

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

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

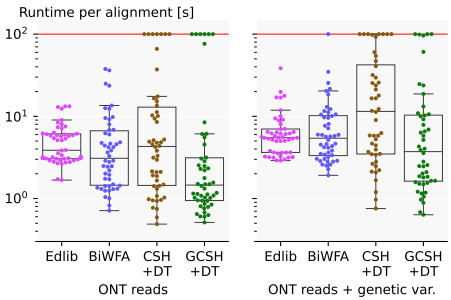


FIGURE 3.5: **Runtime on long human reads.** Each dot is an alignment without (left) and with (right) genetic variation. Runtime is capped at 100 s. The median speedup of A*PA (GCSH + DT, k=15, r=2) is 2× (left) and 1.4× (right) over EDLIB and BIWFA.

3.4.2 *Comparison on synthetic data*

Runtime scaling with length. We compare our A* heuristics with EDLIB and BIWFA in terms of runtime scaling with n and d (Fig. 3.4, extended comparison in Sec. 3.8.1). As theoretically predicted, EDLIB and BIWFA scale quadratically. For small edit distance, EDLIB is subquadratic due to the bit-parallel optimization. The empirical scaling of A*PA is subquadratic for $d \leq 12$ and $n \leq 10^7$, making it the fastest aligner for long sequences ($n > 30\,kbp$). For low divergence ($d \leq 4\%$) even the simplest SH scales near-linearly with length (best fit $n^{1.05}$ for $n \leq 10^7$). For high divergence (d=12%) we need inexact matches, and the runtime of SH sharply degrades for long sequences ($n > 10^6\,bp$) due to spurious matches. This is countered by chaining the matches in CSH and GCSH, which expand linearly many states (Sec. 3.8.2). GCSH with DT is not exactly linear due to state reordering and high memory usage (Sec. 3.8.6).

Performance. A*PA is >300× faster than EDLIB and BIWFA for d=4% and n=10^7 (Fig. 3.4a). For n=10^6 and $d \leq 12\%$, memory usage is less than 500 MB for all heuristics (Sec. 3.8.4).

3.4.3 *Comparison on human data*

We compare runtime (Fig. 3.5 and Sec. 3.8.3), and memory usage (Sec. 3.8.4) on human data. We configure A*PA to prune matches only when expanding their start (not their end), leaving some matches on the optimal path unpruned and speeding up the contour updates. A*PA (GCSH with DT) aligns ONT reads faster than EDLIB and BIWFA in all quartiles, being >2× faster in median. However,

the heuristic does not predict all errors when $d \geq 10\%$, causing 6 alignments to time out. With genetic variation, A*PA is 1.4× faster than Edlib and BiWFA in median. Low-divergence alignments are faster than Edlib, while high-divergence alignments are slower (3 sequences with $d \geq 10\%$ time out) because of expanding quadratically many states in complex regions (Sec. 3.8.8). Since slow alignments dominate the total runtime, Edlib is faster on average.

3.4.4 *Effect of pruning, inexact matches, chaining, and DT*

We visualize the effects of complex sequences on the A* search (Sec. 3.8.9).

Pruning enables near-linear runtime. Figure 3.4a shows that match pruning changes the quadratic runtime of SH to near-linear, by penalizing the expansion of states before the search tip.

Inexact matches cope with higher divergence. Inexact matches increase the heuristic potential, allowing higher divergence (Fig. 3.4c). For low divergence, the runtime is nearly constant (Sec. 3.8.5), while for higher divergence it switches to linear. Inexact matches nearly double the threshold for constant runtime from $d \leq 10\% = 1/k$ to $d \leq 18\% \approx 2/k$.

Chaining copes with spurious matches. When seeds have many spurious matches (typical for inexact matches, $r{=}2$ in Fig. 3.4c), SH loses its strength. CSH reduces the degradation of SH by chaining matches.

Gap-chaining copes with indels. Gap costs penalize both short and long indels which are common in genetic variation. As a result, GCSH is significantly faster than CSH with genetic variation (Fig. 3.5).

Diagonal transition speeds up quadratic search. When A* explores quadratically many states, it behaves similar to Dijkstra. DT speeds up Dijkstra 20× (Fig. 3.4a) and CSH 3× (Fig. 3.4c). For Dijkstra this is close to $1/d$, the expected reduction in number of expanded states. DT also speeds up the alignment of human data (Sec. 3.8.3).

3.5 DISCUSSION

Seeds are necessary, matches are optional. Unlike the seed-and-extend paradigm which reconstructs good alignments by connecting seed matches, we use the lack of matches to penalize alignments. Given the admissibility of our heuristics, the more seeds without matches, the higher the penalty for alignments and the easier it is to dismiss suboptimal ones. In the extreme, not having any matches can be sufficient for finding an optimal alignment in linear time (Sec. 3.8.7).

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

Limitations.

1. *Limited divergence.* Our heuristics can estimate the remaining edit distance up to about $d \leq 10\%$ (on human data). Higher divergence triggers a slow quadratic search.
2. *Complex regions.* Regions with high divergence, long indels, or many matches trigger quadratic runtime (Sections 3.8.8 and 3.8.9).
3. *Computational overhead of A*.* Graph algorithms are $\gg 100\times$ less efficient than DP (compare Dijkstra and Edlib in Fig. 3.4a). Despite the near-linear scaling of A*PA, it is only faster than Edlib and BiWFA for $n \geq 30\,kbp$.

Asymptotic analysis. We do not

Future work.

1. *Performance.* The runtime could be improved by variable seed lengths, overlapping seeds, more aggressive pruning, and better parameter tuning. Implementation optimizations may include computational domains [29], block computations [89], SIMD (similar to BiWFA), or bit-parallelization [35] (similar to Edlib).
2. *Generalizations.* Our chaining seed heuristic could be generalized to non-unit and affine costs, and to semi-global alignment. Cost models that better correspond to the data can speed up the alignment.
3. *Relaxations.* At the expense of optimality guarantees, inadmissible heuristics could speed up A* considerably. Another possible relaxation would be to validate the optimality of a given alignment instead of computing it from scratch.
4. *Analysis.* The near-linear scaling of A* is not asymptotic and requires a more thorough theoretical analysis [94].

3.6 PSEUDOCODE

3.6.1 *A* algorithm for match pruning*

We present our variant of A* [7] that supports match pruning (Algorithm 5). All computed values of g are stored in a hash map, and all *open* states are stored in a bucket queue of tuples $(v, g(v), f(v))$ ordered by increasing f. Line 14 prunes (removes) a match and thereby increases some heuristic values before that match.

As a result, some f-values in the priority queue may become outdated. Line 11 solves this problem by checking if the f-value of the state about to be expanded was changed, and if so, line 12 pushes the updated state to the queue, and proceeds by choosing a next best state. This way, we guarantee that the expanded state has minimal updated f. To reconstruct the best alignment we traceback from the target state using the hash map g (not shown).

Algorithm 5 A* algorithm with match pruning.
Lines added for pruning (11, 12, and 14) are marked in **bold**.

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

3.6.2 *Diagonal transition for A**

For a given distance g, the diagonal transition method only considers the *farthest-reaching* (f.r.) state $u=\langle i,j\rangle$ on each diagonal $k=i-j$ at distance g. We use $F_{gk} :=$

$i+j$ to indicate the antidiagonal[4] of the farthest reaching state. Let X_{gk} be the farthest state on diagonal k adjacent to a state at distance $g-1$, which is then *extended* into F_{gk} by following as many matches as possible. The edit distance is the lowest g such that $F_{g,n-m} \geq n+m$, and we have the recursion

$$X_{gk} := \max(F_{g-1,k-1}+1, F_{g-1,k}+2, F_{g-1,k+1}+1), \tag{3.3}$$

$$F_{gk} = X_{gk} + \text{LCP}\left(A_{\geq (X_{gk}+k)/2}, B_{\geq (X_{gk}-k)/2}\right). \tag{3.4}$$

The base case is $X_{0,0}=0$ with default value $F_{gk}=-\infty$ for $k>|g|$, and LCP is the length of the longest common prefix of two strings. Each edge in a traceback path is either a match created by an extension (3.4), or a mismatch starting in a f.r. state (3.3). We call such a path an *f.r. path*.

Implementation. In Algorithm 6 we further modify the A* algorithm to only consider f.r. paths. We replace the map g that tracks the best distance by a map F_{gk} that tracks f.r. states (lines 4, 18, and 19). Instead of $g(u)$ decreasing over time, we now ensure that $F_{g,k}$ increases over time. Each time a state u is opened or expanded, the check whether $g(u)$ decreases is replaced by a check whether F_{gk} increases (line 9). This causes the search to skip states that are known to not be farthest reaching. The proof of correctness (theorem 6) still applies.

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

4 Previous works indicate the column i of u, but using the antidiagonal $i+j$ keeps the symmetry between insertions and deletions.

Algorithm 6 A*-DT algorithm with match pruning.
Lines changed for diagonal transition (4, 9, 18, and 19) are in **bold**.

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

3.6.3 *Implementation notes*

Here we list implementation details on performance and correctness.

Bucket queue. We use a hashmap to store all computed values of g in the A* algorithm. Since the edit costs are bounded integers, we implement the priority queue using a *bucket queue* [32]. Unlike heaps, this data structure has amortized constant time push and pop operations since the value difference between consecutive pop operations is bounded.

Greedy matching of letters. From a state $\langle i, j\rangle$ where $a_i = b_j$, it is sufficient to only consider the matching edge to $\langle i{+}1, j{+}1\rangle$ [33, 85], and ignore the insertion and deletion edges to $\langle i, j{+}1\rangle$ and $\langle i{+}1, j\rangle$. During alignment, we greedily match as many letters as possible within the current seed before inserting only the last open state in the priority queue, but we do not cross seed boundaries in

order to not interfere with match pruning. This optimization is superseded by the DT-optimization. We include greedily matched states in the reported number of expanded states.

Priority queue offset. Pruning the last remaining match in a layer may cause an increase of the heuristic in all states preceding the start u of the match. This invalidates f values in the priority queue and causes reordering. We skip most of the update operations by storing a global offset to the f-values in the priority queue, which we update when all states in the priority queue precede u.

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

Binary search speedup. Instead of using binary search to determine the layer/score $S_p(u)$ (Algorithm 7), we first try a linear search around either the score of the parent of u or a known previous score at u. This linear search usually finds the correct layer in a few iterations, or otherwise we fall back to binary search.

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

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

3.6.4 *Computation of the heuristic*

Algorithm 7 shows how the heuristic is initialized, how $S_p(u)$ and $h(u)$ are computed, and how matches are pruned.

3.7 PROOFS

3.7.1 *Admissibility*

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

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

Lemma 2 (Weak triangle inequality). *For states u, v, and w with v and w at the starts of some seeds, all $\gamma \in \{c_{\text{seed}}, c_{\text{gap}}, c_{\text{gs}}\}$ satisfy*

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

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

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

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

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

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

□

Lemma 3 (Weak consistency). *Let $h \in \{h_{\text{s}}^{\mathcal{M}}, h_{\text{cs}}^{\mathcal{M}}, h_{\text{gcs}}^{\mathcal{M}}\}$ be a heuristic with partial order $\preceq_p$, and let $u \preceq_p v$ be states with v at the start of a seed. When there is a shortest path π^* from u to v such that $\mathcal{M}$ contains all matches of cost less than r on π^*, it holds that $h(u) \leq \mathrm{d}(u, v) + h(v)$.*

Proof. The path π^* covers each seed in $\mathcal{S}_{u\ldots v}$ that must to be fully aligned between u and v. Since the seeds do not overlap, their shortest alignments π_s^* in π^* do not have overlapping edges. Let $u \preceq m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v$ be the chain of matches $m_i \in \mathcal{M}$ corresponding to those π_s^* of cost less than r (Fig. 3.6). Since the matches and the paths between them are disjoint, $\mathrm{c}_{\text{path}}(\pi^*)$ is at least the cost of the matches $\mathrm{c}_{\mathrm{m}}(m_{i+1}) = \mathrm{d}(\mathrm{start}(m_{i+1}), \mathrm{end}(m_{i+1}))$ plus the cost to chain

these matches $\gamma(\text{end}(m_i), \text{start}(m_{i+1})) \leq \text{d}(\text{end}(m_i), \text{start}(m_{i+1}))$. Putting this together:

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

Now let $v \preceq_p m_{l+1} \preceq_p \cdots \preceq_p m_{l'} \preceq_p v_t$ be a chain of matches minimizing $h(v)$ (Def. 1) with $w := \text{start}(m_{l+1})$. This chain also minimizes $h(w)$ and thus $h(v) = \gamma(v, w) + h(w)$. We can now bound the cost of the joined chain from u to v and from w to the end and get our result via $\gamma(m_l, w) \leq \gamma(m_l, v) + \gamma(v, w)$ (lemma 2)

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

□

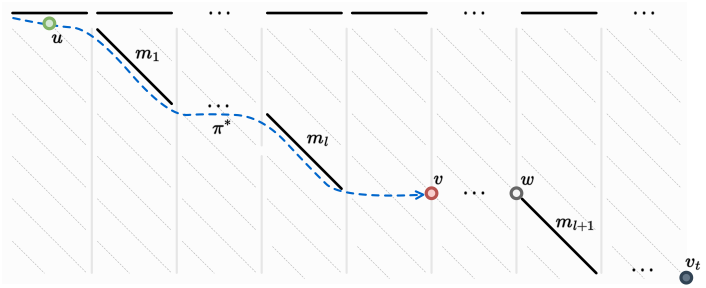


FIGURE 3.6: **Variables of the proof of lemma 3.**

Theorem 5. *The seed heuristic h_s, the chaining seed heuristic h_cs, and the gap-chaining seed heuristic h_gcs are* admissible. *Furthermore, $h_\text{s}^{\mathcal{M}}(u) \leq h_\text{cs}^{\mathcal{M}}(u) \leq h_\text{gcs}^{\mathcal{M}}(u)$ for all states u.*

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

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

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

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

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

3.7.2 *Match pruning*

During the A* search, we continuously improve our heuristic using match pruning. The pruning increases the value of our heuristics and breaks their admissibility. Nevertheless, we prove in two steps that A* with match pruning still finds a shortest path. First, we introduce the concept of a *weakly-admissible heuristic* and show that A* using a weakly-admissible heuristic finds a shortest path (theorem 6). Then, we show that our pruning heuristics are indeed weakly admissible (theorem 7).

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

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

A fixed node cannot be opened again (Algorithm 5, line 18), and hence remains fixed.

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

To prove that A* finds a shortest path when used with a weakly-admissible heuristic, we follow the structure of Hart, Nilsson & Raphael [7]. First we restate their Lemma 1 in our notation with a slightly stronger conclusion that follows directly from their proof.

Lemma 4 (Lemma 1 of Hart, Nilsson & Raphael [7])**.** *For any unfixed node n and for any shortest path π^* from v_s to n, there exists an open node n' on π^* with $g(n') = g^*(n')$ such that π^* does not contain fixed nodes after n'.*

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

Corollary 1 (Generalization of Corollary to Lemma 1 of Hart, Nilsson & Raphael [7])**.** *Suppose that $\hat{h}$ is weakly admissible, and that A* has not terminated. Then, there exists an open node n' on a shortest path from v_s to v_t with $f(n') \leq g^*(v_t)$.*

Proof. Let π^* be the shortest path from v_s to v_t given by the weak admissibility of $\hat{h}$ (Def. 7). Since A* has not terminated, v_t is not fixed. Substitute $n = v_t$ in lemma 4 to derive that there exists an open node n' on π^* with $g(n') = g^*(n')$. By definition of f we have $f(n') = g(n') + \hat{h}(n')$. Since π^* does not contain any fixed nodes after n', the weak admissibility of $\hat{h}$ implies $\hat{h}(n') \leq h^*(n')$. Thus, $f(n') = g(n') + \hat{h}(n') \leq g^*(n') + h^*(n') = g^*(v_t)$. □

Theorem 6. *A* with a weakly-admissible heuristic finds a shortest path.*

Proof. The proof of Theorem 1 in Hart, Nilsson & Raphael [7] applies, with the remark that instead of an arbitrary shortest path, we use the specific path π^* given by the weak admissibility and the specific node n' given by corollary 1. □

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

Lemma 5. *For $\hat{h} \in \{\hat{h}_\mathrm{s}, \hat{h}_\mathrm{cs}, \hat{h}_\mathrm{gcs}\}$, every state at the start of a seed becomes fixed immediately when A* expands it.*

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

Let n^* be the last fixed state on π^* before v, and let $u \in \pi^*$ be the successor of n^*. State u is open because its predecessor n^* is fixed and on a shortest path to u. Let the chain of all matches of cost less than r on π^* between u and v be $u \preceq m_1 \preceq \cdots \preceq m_l \preceq v$. Since n^* is the last fixed state on π^*, none of these matches has been pruned, and they are all in $\mathcal{M}\backslash E$ as well. This means we can apply lemma 3 to get $h(u) \leq \mathrm{d}(u, v) + h(v)$, so

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

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

Theorem 7. *The pruning heuristics $\hat{h}_\mathrm{s}$, $\hat{h}_\mathrm{cs}$, $\hat{h}_\mathrm{gcs}$ are weakly admissible.*

Proof. Let π^* be a shortest path from v_s to v_t. At any point during the A* search, let n^* be the farthest expanded state on π^* that is at the start of a seed. By lemma 5, n^* is fixed. By the choice of n^*, no states on π^* after n^* that are at the start of a seed are expanded, so no matches on π^* following n^* are pruned. Now the proof of theorem 5 applies to the part of π^* after n^* without changes, implying that $\hat{h}(u) \leq h^*(u)$ for all u on π^* following n^*, for any $\hat{h} \in \{\hat{h}_s, \hat{h}_{cs}, \hat{h}_{gcs}\}$. □

3.7.3 *Computation of the (chaining) seed heuristic*

Theorem 8. *$h^{\mathcal{M}}_{p,c_{seed}}(u) = P(u) - S_p(u)$ for any partial order $\preceq_p$ that is a refinement of $\preceq_i$ (i.e. $u \preceq_p v$ must imply $u \preceq_i v$).*

Proof. For a chain of matches $\{m_i\} \subseteq \mathcal{M}$, let s_i and t_i be the start and end states of m_i. We translate the terms of our heuristic from costs to potentials and match scores (Sec. 3.3.4):

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

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

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

□

Lemma 6. *Layer $\mathcal{L}_\ell$ ($\ell > 0$) is fully determined by the set of those matches m for which $\ell \leq S_p(m) < \ell + r$:*

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

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

3.7.4 *Computation of the gap-chaining seed heuristic*

In this section we prove (theorem 9) that we can change the dependency of GCSH on c_{gs} to c_{seed} by introducing a new partial order $\preceq_T$ on the matches. This way, theorem 8 applies and we can efficiently compute GCSH. Recall that the gap-seed cost is $c_{\text{gs}} = \max(c_{\text{gap}}, c_{\text{seed}})$, and that the gap transformation is:

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

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

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

Lemma 7. *Let u and v be two states with v at the start of some seed. Then $u \preceq_T v$ if and only if $c_{\text{gap}}(u,v) \le c_{\text{seed}}(u,v)$. Furthermore, $u \preceq_T v$ implies $u \preceq v$.*

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

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

This simplifies to

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

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

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

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

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

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

$$c_{\text{gs}}(u,v) = \begin{cases} c_{\text{seed}}(u,v) & \text{if } u \preceq_T v, \\ c_{\text{gap}}(u,v) & \text{if } u \not\preceq_T v. \end{cases} \tag{3.5}$$

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

Lemma 8. *When the set of matches $\mathcal{M}$ is consistent, $h^{\mathcal{M}}_{\text{gcs}}(u)$ can be computed using $\preceq_T$-chains only:*

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

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

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

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

By definition, $c_{\text{gs}} \geq c_{\text{gap}}$, and $\text{c}_{\text{m}}(m_i) \geq c_{\text{gap}}(\text{start}(m_i), \text{end}(m_i))$, so the weak triangle inequality (lemma 2) for c_{gap} gives

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

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

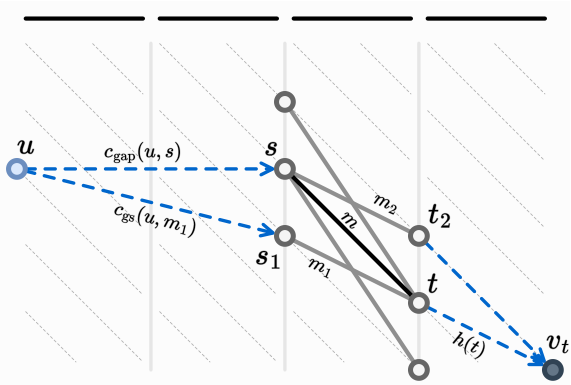


FIGURE 3.7: **Variables in Case 2 of the proof of lemma 8.** Match m has $\text{score}(m) = 2$, so it has adjacent matches m_1 and m_2 (gray) with $\text{score}(m_i) \geq 1$.

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

$$h(u) = \min_{\substack{m \in \mathcal{M} \\ u \preceq m \preceq v_t}} \left(c_{\text{gs}}(u, m) + \text{c}_{\text{m}}(m) + h(\text{end}(m)) \right) \tag{3.6}$$

$$h'(u) = \min_{\substack{m \in \mathcal{M} \\ u \preceq_T m \preceq_T v_t}} \left(c_{\text{gs}}(u, m) + \text{c}_{\text{m}}(m) + h'(\text{end}(m)) \right), \tag{3.7}$$

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

$$h(u) = \min_{\substack{m\in\mathcal{M}\\ u\preceq_T m\preceq_T v_t}} \big(c_{\text{gs}}(u,m) + \text{c}_\text{m}(m) + h(\text{end}(m))\big), \tag{3.8}$$

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

By lemma 7, every $\preceq_T$-chain is a $\preceq$-chain, so $h(u) \le h'(u)$. To prove $h(u){=}h'(u)$, it remains to show the reverse inequality, $h'(u) \le h(u)$. To this end, choose a match m that

(PRIORITY 0) minimizes $h(u)$ in Eq. (3.6), and among those, has
(PRIORITY 1) maximal $c_{\text{seed}}(u,m)$, and among those, has
(PRIORITY 2) minimal $c_{\text{gap}}(u,m)$, and among those, has
(PRIORITY 3) minimal $c_{\text{gap}}(m,v_t)$.

We show that $u \preceq_T m$ (in 2.A) and $m \preceq_T v_t$ (in 2.B), which proves Eq. (3.8).

Part 2.A: $u \preceq_T m$. Let s and t be the begin and end of m (Fig. 3.7), and let m' be a match minimizing $h(t)$ in Eq. (3.6) so

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

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

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

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

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

so we have

$$c_{\text{gs}}(u,m) + \text{c}_\text{m}(m) < c_{\text{gs}}(u,t). \tag{3.9}$$

From the triangle inequality for c_{gap}, from $c_{\text{gap}}(u,s) \le c_{\text{gs}}(u,s)$, and from $c_{\text{gap}}(s,t) \le \text{c}_\text{m}(m)$, and from Eq. (3.9) we obtain

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

This implies $c_{\text{gs}}(u,t) = c_{\text{seed}}(u,t)$ and hence reusing Eq. (3.9)

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

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

When $\mathrm{score}(m) = 1$, this implies $c_{\mathrm{gap}}(u,s) \leq c_{\mathrm{seed}}(u,s)$ and $u \preceq_T s = \mathrm{start}(m)$ by lemma 7.

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

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

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

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

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

$$h(u) = c_{\mathrm{gs}}(u,s) + \mathrm{c_m}(m) + c_{\mathrm{gs}}(m,v_t) < c_{\mathrm{gs}}(u,v_t).$$

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

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

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

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

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

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

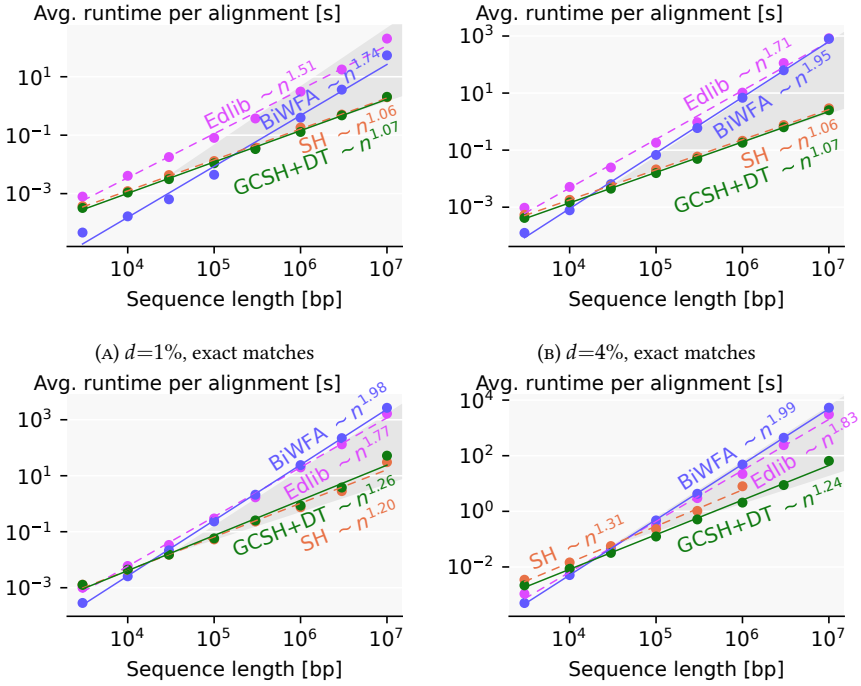
Theorem 9. *Given a* consistent *set of matches $\mathcal{M}$, the gap-chaining seed heuristic can be computed using scores in the transformed domain:*

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

Proof. Write $h := h^{\mathcal{M}}_{\text{gcs}}$ and $h' := h^{\mathcal{M}}_{\preceq_T, c_{\text{gs}}}$. When $u \not\preceq_T v_t$, $h(u) = c_{\text{gap}}(u, v_t)$ by lemma 8. Otherwise, when $u \preceq_T v_t$, we have $h(u) = h'(u)$. Let $u \preceq_T m_1 \preceq_T \ldots \preceq_T v_t$ be a $\preceq_T$-chain for h' as in Def. 1. All terms in h' satisfy $\mathsf{end}(m_i) \preceq_T \mathsf{start}(m_{i+1})$, so $c_{\text{gap}} \leq c_{\text{seed}}$ and by lemma 7 $c_{\text{gs}}(m_i, m_{i+1}) = c_{\text{seed}}(m_i, m_{i+1})$. Thus, $h' = h_{\preceq_T, c_{\text{seed}}}$, and $h^{\mathcal{M}}_{\text{gcs}}(u) = P(u) - S_T(u)$ by theorem 8. □

3.8 FURTHER RESULTS

3.8.1 *Tool comparison on synthetic data*



(A) d=1%, exact matches

(B) d=4%, exact matches

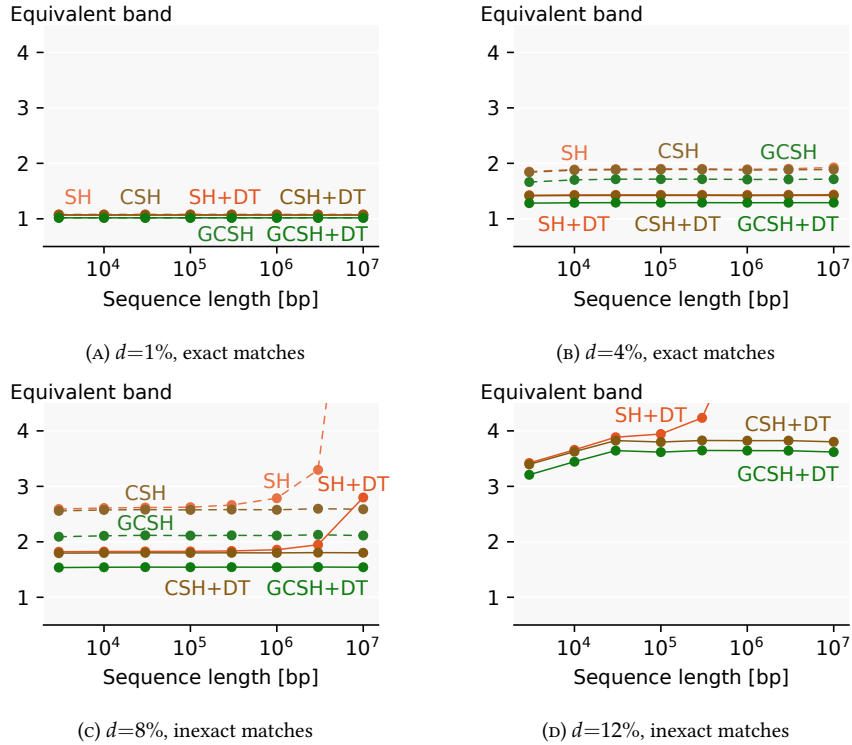
(C) d=8%, inexact matches

(D) d=12%, inexact matches

FIGURE 3.8: **Runtime scaling with sequence length on synthetic data.** Log-log plots comparing our simplest heuristic (SH) and our most accurate heuristic (GCSH, with DT) with EDLIB and BIWFA. The slopes of the bottom (top) of the dark-grey cones correspond to linear (quadratic) growth. The seed length is k=15 and for $d \geq 8\%$ the heuristics use inexact matches (r=2). Averages are over total $N = 10^7$ bp. The missing data points for SH at d=12% are due to exceeding the 32 GiB memory limit.

In Fig. 3.8 we compare our A* heuristics with EDLIB and BIWFA in terms of runtime scaling with sequence length n and divergence d.

3.8.2 *Expanded states and equivalent band*



(A) d=1%, exact matches

(B) d=4%, exact matches

(C) d=8%, inexact matches

(D) d=12%, inexact matches

FIGURE 3.9: **Equivalent band scaling with sequence length on synthetic data.** (k=15, r=2 for $d \geq 8\%$). The equivalent band is the number of expanded states per bp for aligning synthetic sequences. Averages are over total $N=10^7$ bp. At d=12%, the bands of non-DT methods are $\geq 3\times$ wider.

The main benefit of an A* heuristic is a lower number of expanded states, which translates to faster runtime. Instead of evaluating the runtime scaling with length (Fig. 3.8), we can judge how well a heuristic approximates the edit distance by directly measuring the *equivalent band* (Fig. 3.9) of each alignment: the number of expanded states divided by sequence length n, or equivalently, the number of expanded states per base pair. The theoretical lower bound is an equivalent band of 1, resulting from expanding only the states on the main diagonal.

The equivalent band tends to be constant in n, indicating that the number of expanded states is linear on the given domain. The equivalent band of SH starts

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

3.8.3 *Human data results*

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

TABLE 3.2: **Human datasets statistics.** ONT reads only include short gaps, while genetic variation also includes long gaps. **Cnt**: number of sequence pairs. **Max gap**: longest gap in the reconstructed alignment.

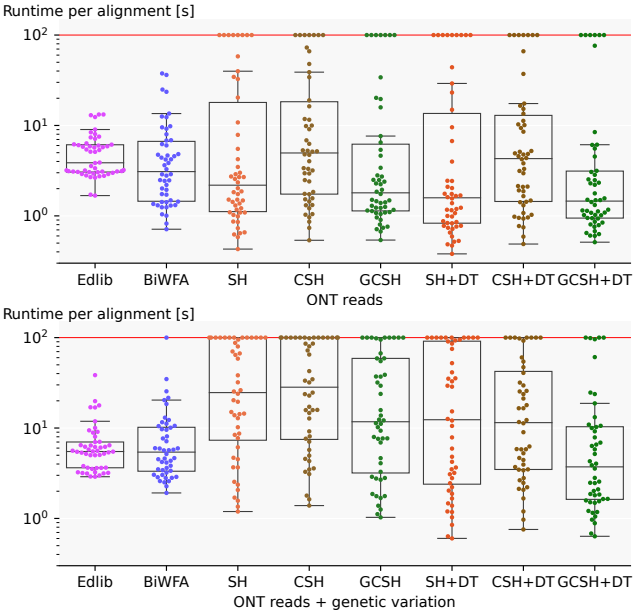


FIGURE 3.10: **Runtime on long reads of human data** (logarithmic, $\geq 500\,kbp$). Each dot corresponds to an alignment of a sequence pair. The ONT reads are without (top) and with (bottom) genetic variation. Runtime is capped at 100 s.

Statistics on our human data are presented in Table 3.2. We compare all our heuristics in Fig. 3.10.

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

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

3.8.4 *Memory usage*

Table 3.3 compares memory usage of aligners on synthetic sequences of length $n{=}10^6$. Alignments with inexact matches ($d{\geq}8\%$) use significantly more memory

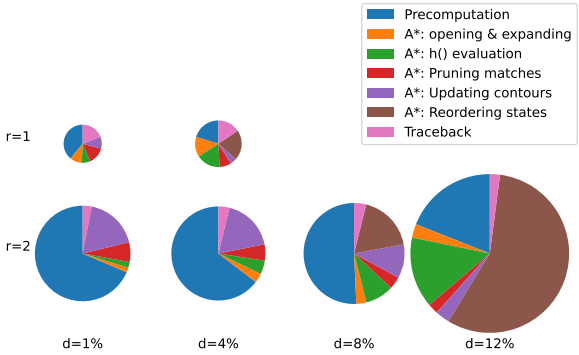
than those with exact matches because more k-mer hashes need to be stored to find inexact matches. Table 3.4 compares memory usage on the human data sets.

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

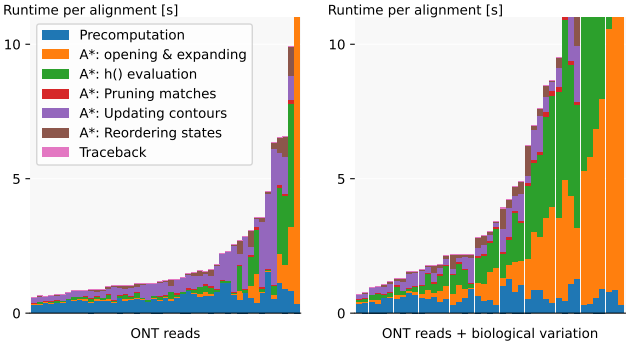
Table 3.3: **Memory usage per algorithm** (synthetic data, n=10^6). Exact matches are used when $d \leq 4\%$, and inexact matches when $d \geq 8\%$.

	ONT reads		+ genetic var.	
Aligner	Median	Max	Median	Max
Edlib	2	5	2	6
BiWFA	11	19	15	24
A*PA (GCSH + DT)	160	3478	270	6926

Table 3.4: **Memory usage [MB] of aligners on human data.** Medians are over all alignments; maximums are over alignments not timing out.



(A) Synthetic data



(B) Human data

FIGURE 3.11: **Runtime distributions per stage of A*PA (GCSH with DT)** (stages do not overlap). Stage $A^\star$ includes expanding and opening states. *Pruning matches* includes consistency checks. *Updating contours* includes updating of contours after pruning. a On synthetic data (n=10^6 bp, N=10^7 bp total). The circle area is proportional to the total runtime. Figures for r=1 and $d \geq 8\%$ are skipped due to timeouts (100 s). b On human data (r=2). Alignments are sorted by total runtime (timeouts not shown).

3.8.5 *Scaling with divergence*

Figure 3.12 shows the runtime scaling with divergence for various heuristics. The heuristics using inexact matches (r=2) are slower for low-divergence sequences due to spending relatively a lot of time on the initialization of the heuristic, which requires finding all inexact matches.

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

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

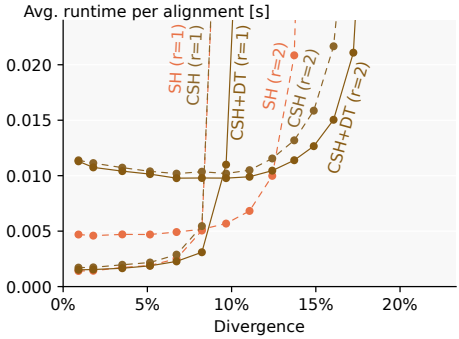


FIGURE 3.12: **Runtime scaling with divergence (zoomed)** (linear, synthetic, n=10^4, 10^6 bp total, k=10). The figure shows the same results as Fig. 3.4c, but zoomed in to small runtimes where the scaling with n degrades from linear to quadratic. r=1 and r=2 indicate exact and inexact matches, respectively.

3.8.6 *Runtime of A*PA internals*

Figures 3.11a and 3.11b compare the time used by stages of A*PA.

3.8.7 *Linear mode without matches*

States are penalized by the number of remaining seeds that cannot be matched. So, curiously, matches are not always needed to direct the A* search to an optimal path. In fact, when each seed contains exactly one mutation seed heuristic scales linearly even though there are no matches, as shown by the artificial example in Fig. 3.13.

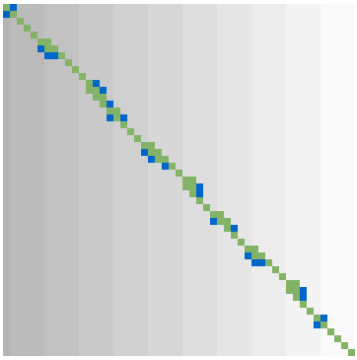
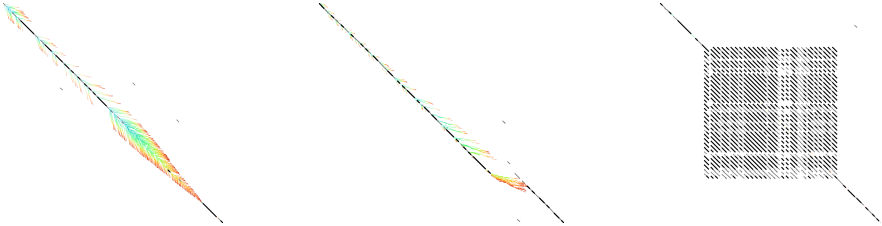


FIGURE 3.13: **Artificial example of A* with seed heuristic with no matches** (n=m=50, r=1, k=5, 80% similarity, 1 mutation per seed alternating substitution, insertion, and deletion). The background colour indicates $h_s(u)$ with higher values darker. Expanded states are green (■), open states blue (■).

3.8.8 *Complex cases*



(A) High divergence (B) Long indel (C) Short repeats

FIGURE 3.14: **Quadratic exploration behavior for complex alignments** (GCSH with DT, r=2, k=10, synthetic sequences, n=1000). a A highly divergent region, b a deletion, and c a short repeated pattern inducing a quadratic number of matches. The colour corresponds to the order of expansion, from blue to red.

3.8.9 *Comparison of heuristics and techniques*

Figure 3.15 shows the effect of our heuristics and optimizations for aligning complex short sequences. The effect of pruning is most noticeable for CSH and GCSH without DT. GCSH is our most accurate heuristic, so, as expected, it leads to the lowest number of expanded states.

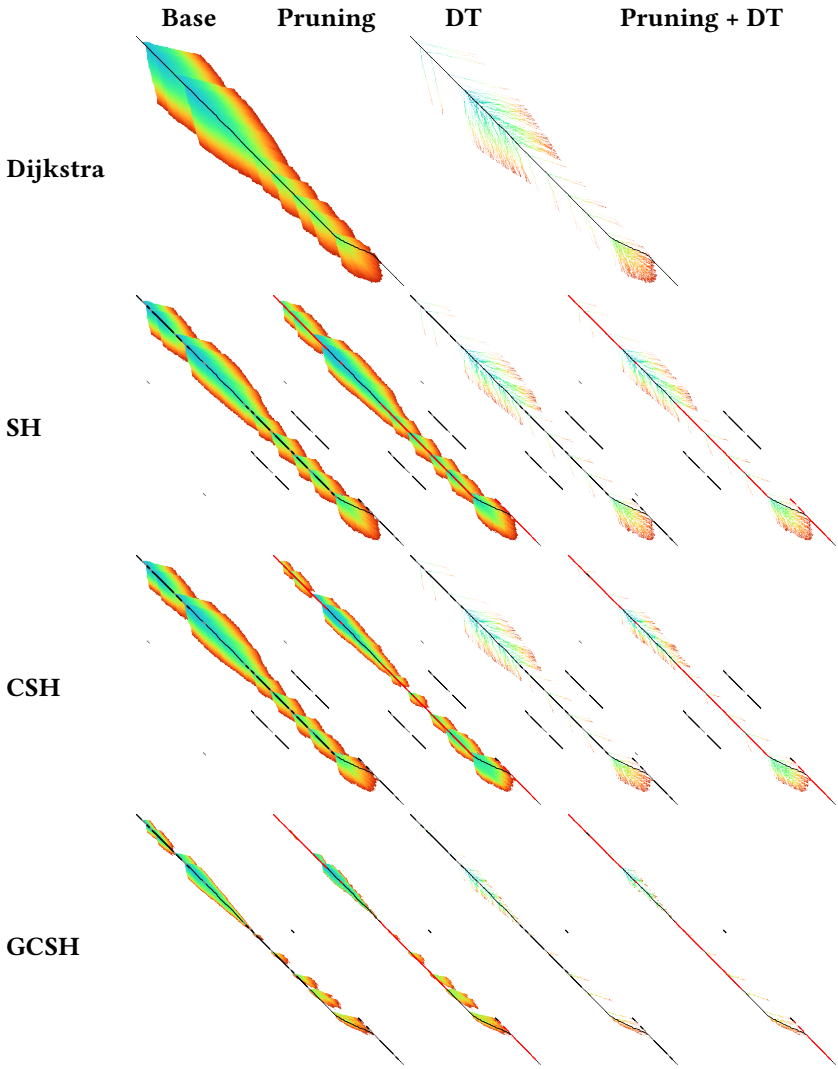


FIGURE 3.15: **Expanded states for various heuristics and techniques, on a sequence containing a noisy region, a repeat, and an indel** (n=1000, d=17.5%). The colour shows the order of expanding, from blue to red. The sequences include a highly divergent region, a repeat, and a gap. Matches are shown as **black diagonals**, with inexact matches in **grey** and pruned matches in red. The final path is **black**. Dijkstra does not have pruning variants, and Dijkstra with DT is equivalent to WFA. More accurate heuristics reduce the number of expanded states by more effectively punishing repeats (CSH) and gaps (GCSH). Pruning reduces the number of expanded states before the pruned matches, and diagonal transition reduces the density of expanded states in quadratic regions.

Algorithm 7 Computation of the heuristic.

Input: Sequences A and B
Output: Heuristic h
function INITIALIZEHEURISTIC(A, B, k)
    $S \leftarrow$ Non-overlapping k-mers of A ▷ Seeds
    $H \leftarrow$ Map of all k-mers (and $k\pm1$-mers for $r = 2$) of B
    $\mathcal{M}_s \leftarrow \bigcup_{s' \colon \mathrm{ed}(s,s')<r} H[s']$ ▷ Seed matches
    $\mathcal{M} \leftarrow \bigcup_{s \in \mathcal{S}} \mathcal{M}_s$ ▷ Matches
    $P(i) \leftarrow r \cdot |S_{\geq i}|$ for all $i \in [0, |A|]$ ▷ Potentials
    $L[0] = \{m_\omega\}$ ▷ Layers
    for $m \in \mathcal{M}$ in decreasing order of $\preceq_p$ **do**
        $\ell \leftarrow \mathrm{score}(m) + S_p(\mathrm{end}(m))$
        if $m \preceq_p m_\omega$ **then** ▷ If m precedes the target
            $L[\ell].push(m)$

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

function $h_{\mathrm{s}}(u)$ ▷ SH
    return $P(u) - S_i(u)$
function $h_{\mathrm{cs}}(u)$ ▷ CSH
    return $P(u) - S_{\preceq}(u)$
function $h_{\mathrm{gcs}}(u)$ ▷ GCSH
    return $\max(P(u) - S_T(u), c_{\mathrm{gap}}(u, v_t))$

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

CONCLUSION

We presented a novel A* approach to alignment which is provably optimal and heuristically fast on related sequences. We incrementally extended the applicability of our approach to several data dimensions: reference size, query/sequence length and error rate. Using a trie, semi-global alignment empirically scaled sublinearly with reference size. We introduced a *seed heuristic* which scaled semi-global alignment subquadratically with query length. Extended the seed heuristic with chaining, inexact matches, gap costs, and match pruning enabled near-linear scaling with sequence length for up to 30% error rate. We proved that all our heuristics and optimizations are guaranteed to find an alignment with minimal edit distance. Except for the German translation of the abstract, this last mastodon has been written fully on human intelligence.

Scaling with reference size. In Chapter 1 we presented AStarix, a sequence-to-graph optimal alignment tool based on the A* algorithm with a simple admissible heuristic and enhanced by multiple algorithmic optimizations. We complemented the refernece with a trie index to allow a shortest path from the trie root to be found for sublinear time with the reference size. Our approach allows for general graph references that may include cycles. 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 in orders of magnitude.

Scaling with sequence length. In Chapter 2 we upgraded our sequence-to-graph aligner AStarix with a novel admissible *seed heuristic* which guides the search based on seed matches between the query and the reference. In order to compute the seed heuristic efficiently for each explored node, for each query, we split it to seeds, precompute the seed matches in the reference, and place *crumbs* on the nodes before the match in order to mark an upcoming match. In addition to the sublinear scaling with reference size, the seed heuristic allowed subquadratic scaling with the query length. AStarix outpeformed existing optimal aligners on long reads.

Scaling with error rate. In Chapter 3 we introduced A*PA which solves global alignment. To this end, we extended the existing seed heuristic for A* with inexact matches, match chaining, gap costs and other algorithmic improvements. On random sequences with 8% uniform divergence, the runtime of A*PA scales near-linearly to sequences of tens of millions of letters and outperforms other exact

aligners. On long ONT reads of human data with $d \leq 10\%$, A*PA is over $2\times$ faster than other aligners, and $1.4\times$ faster when genetic variation is also present.

DISCUSSION

Here we discuss the proposed A* algorithm for sequence alignment: its limitations, potential for future developments, and general thoughts about it.

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 approach requires quadratic time. In particular, regions with high error rate, long indels, and too many matches are challenging and trigger quadratic exploration.
2. *High constant in runtime complexity.* Despite the near-linear scaling of the number of expanded states, A*PA only outperforms Edlib and BiWFA for sufficiently long sequences due to the relatively high computational constant that the A* search induces.
3. *Complex parameter tuning.* The performance of our approach 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 may require a more comprehensive framework when introducing additional optimizations.
4. *Consistency.* The seed heuristic, which is in the core of our A* algorithm, is admissible (optimistic) but not consistent (monotone). As a consequence, any of the quadratic number of state can be expanded multiple times, which could theoretically lead to over-quadratic scaling. In practice, repeated expansions happen but the empirical scaling is preserved near-linear as long as the number of seed matches is near-linear and the seed heuristic is capable of compensating for the errors.

FUTURE WORK

We initiated a new direction of sequence alignment based on the informed shortest path algorithms A* that we have shown to be both provably optimal, practically scalable and more performant than existing aligners in certain cases. A general objective is to develop the A* for sequence alignment from prototypical to prac-

tical, which includes the development of algorithms, formal proofs and software development. Our specific aim is to use the already constructed base for the development of ASTARIX as an industrial-scale aligner handling human-scale graph references, long and noisy reads, while being competitive in runtime and memory even to suboptimal aligners. In order to fulfill this aim, work in several directions will be needed: 1) rigorous development of the seed heuristic theory, 2) theoretical analysis of the expected runtime, 3) implementation optimized for runtime, memory, parallelization and use-cases. We foresee a multitude of extensions and optimizations that may lead to efficient sequence aligning for production usage. Moreover, we suppose that due to incorporating fuller information for its informed search, A* could become the default framework for sequence alignment, useful for computational biology, bioinformatics and general informatics.

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 [25, 24], more aggressive pruning, and better parameter tuning. More efficient implementations may be possible by using computational domains [29], bit-parallelization [35], and SIMD [90], lowering the A* constant. The efficiency of the presented algorithm has high variability on real data (Sec. 3.4.3) due to high error rates, long indels, and multiple repeats. Further optimizations are needed to align complex data. More accurate heuristics lead to better A* performance [22], so machine learning may be useful for tuning seed heuristic parameters.
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 [94]. 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.
5. *Applicability.* More dimensions of data complexity could be explored: scaling to larger alphabet size, general language, compressed low-enthropy text,

higher reference complexity (repeats, multiple best alignments, bubbles, cycles).

6. *Optimality verification* Verify the optimality of a given alignment.

Planned work packages

Practical sequence-to-graph aligner. Develop the AStarix sequence-to-graph aligner from prototype to production As our previous work demonstrates, scaling optimal A* alignment to long sequences and high error rates relies on various non-trivial algorithms, data structures and optimizations. In order for AStarix to become of practical importance, it must be comparable to not only optimal aligners but to approximate ones as well – this includes tolerating more errors, lowering the runtime and memory consumption, and supporting features. Tolerating errors. We observe two qualitatively-different modes in which the A* explores the state space: linear and quadratic. The mode of exploration is mainly determined by the ability of the seed heuristic to "compensate" for the type and amount of errors. When the heuristic cannot cope with the errors, A* has to continue exploring without the aid of the heuristic (similar to Dijkstra): for long indels, the exploration becomes locally quadratic (around the indel), and for too high error rate it becomes globally-quadratic (because of the accumulating insufficiency of the seeds). By tolerating errors, we mean the ability of the exploration to stay linear or near-linear. Currently, AStarix supports 2-4% errors on short reads and only 0.3% on HiFi reads, whereas our recent developments on global alignment demonstrated the potential to tolerate error rate to 25% using match pruning, inexact matching, seed chaining. Developing these features in the semi-global setting is not trivial because of the added complexity of crumbs and the trie index but we do not envision any fundamental reasons against adopting them. A natural approach to tolerating long indels, well known in the bioinformatics community, is to generalize the edit distance metric to affine costs in order to match the error model more closely. This extension is algorithmically simple and consists of or there are longer indels. affine costs for, inexact matches Memory optimization. The main memory bottleneck is currently the trie index (which would take >90% of the memory given a standard compressed genome graph representation) so our primary priority is algorithmically reduce the number of nodes in the trie and then implement it efficiently. Currently, it is used with two separate functions: 1) to find all seed matches, and 2) to allow the search A* to explore the whole reference only implicitly by abstracting together the reference location reachable with the same prefix. The first function is efficiently solvable by using the positional Burrows-Wheeler transform (gBWT) to find exact seed matches and, in the case of inexact matches, to pre-generate

all kmers at distance 1 from the seeds and invoke gBWT separately for each. The second function requires a trie but if possibly not a over the whole graph, only pointing to those reference locations which hold at least one crumb: otherwise the A* search will surely be quadratic in which case the fallback options are to either refuse to align the read (which is still optimal but incomplete), or find an alignment by explicitly considering all starting locations (which could be done faster using another tool). We foresee that building a separate "slim" trie per query will bring orders of magnitude of shrinkage of the trie, making it proportional only to the query length, and not the reference size. Additionally, a compressed implementation of the trie and the reference graph has the potential to lower the current memory usage by an additional order of magnitude. Runtime optimization. For small error rate (0.3%) and long reads (30 kbp), the bottleneck for the runtime is the amount of crumbs. To effectively skip the work for placing most of the crumbs, the placement of different seeds could be done together in one backward traversal – this is expected to eliminate most of the work for placing crumbs on the optimal path. Another approach is to also "jump over unitigs" when placing crumbs in the backwards traversal, and "carry" crumbs in the forwards A* search in order to compensate for the skipped ones. A major optimization in the case of high error rate, is to adapt the match pruning that we have developed for the global alignment case. Features. By discussing AStarix on conferences, we have figured out that extension alignment (very similar in the approach to global alignment) is of great interest for its reusability in the seed-and-extend paradigm that other aligners follow. Evaluation. The dataset for evaluation ideally includes bigger graphs, noisy long reads (e.g. ONP, PacBio), reads with bigger indels. The comparison should be not only to optimal aligners but also to approximate oces. Potential risks. The preliminary experiments demonstrate sublinear scaling with the reference size but since this is an empirical result, it may also deteriorate for bigger graphs. Some of the optimizations may not provide the full expected effect or the bottleneck they would resolve may not yet be current. Usage of AStarix by bioinformaticians is crucial for the adoption of AStarix by the broader community.

Theoretical guarantees. Derive theoretical guarantees for the runtime scaling of A* alignment Two equal sequences can be aligned for linear time, whereas two general sequences cannot be aligned strongly-faster than quadratically [59]. There is a knowledge gap for the computational limitations of related sequences [94] which may benefit for various statistical approaches different from the worst case asymptotic analysis and the empirical investigation [95]. Our strong empirical results on scaling with increasing sequence length and error rates, together with our intuition from the internals of our A* algorithms, hint towards much stricter theoretical bounds on the expected computational costs. Such theoretical results

may also be of general interest outside of the computational biology community. Our approach to proving theoretical bounds is likely to be tightly related to the term potential (of a seed heuristic function) which is a measure of the ability of a specific seed heuristic instantiation (i.e. for fixed seed length, etc) to punish for future errors before exploring them. The analysis should also account for the cost for computing the heuristic (including amortized precomputation and querying) since infinite resources on it can result in an oracle but also destroy the total performance. The risks of such analyses include the potential inability to account for data and error models (random reference sequence and uniform error model) that are realistic enough. Global alignment. Our intuition on the scaling of the explored states includes several parameters: the seed heuristic potential and the number of errors. Informally speaking, for each error that is not compensated by the potential, the A* search will explore another layer of states that are 1 more distant from the diagonal. This is also theoretically and practically true for the special case of A*, Dijkstra, which has potential 0 and thus reaches O(ns) where s is the overall number of errors. Our empirical results demonstrate that a near-linear runtime scaling is practically possible up to very long sequences (100 Mbp) for error rates of 10%. In addition to the computational costs for aligning a sequence to another sequence, alignment has the burden of localization of the best alignment. Our intuition behind his localization cost is related to to usage of A* on the trie: for each error that the A* potential does not compensate for, the exploration would get one lever deeper in the trie, thus exploring expected 4x more states. This is an interesting observation since it hints to an exponential scaling with error rate, which is not asymptotically possible because of the quadratic worst case, but may be the case near the best-case. In such a case, a type of a sigmoid would possibly describe the number of explored states as a function of the number of errors.

Alignment on a probabilistic graph references. Probabilistic interpretations arise in different places among the alignment problems. Phred values. Sequencing machines report a standardized score per nucleotide that estimates the probability of that nucleotide being wrong. This information is currently largely ignored and very valuable for highly erroneous sequencing technologies. Extending the edit distance metric to account for phred values should be directly applicable and may even benefit the performance of the A* algorithm due to the better correspondence between the scores and the error model. Weighted genome reference. A topic for current discussions in the pangenomics field is whether to add more genomes to the reference graph or to keep the reference unbiased (by unbalanced number of entries for different variations, or by adding genomes from other populations). One reason for such discussions are the memory and runtime limitations, but the accuracy of alignment is not less important. A weighting of the graph may

be useful as a soft tradeoff. HMM. HMM queries have been useful in the context of microbial sequencing. Possibly, A* can be extended to handle HMM queries instead of linear sequences. The current solutions are prohibitively slow and AStarix is expected to greatly accelerate the optimal alignment after being generalized to probabilistic queries. The mapq value [41] may be calculated more precisely if accounting for the probability estimates.

Apply A* to local alignment. Explore the applicability of A* to local alignment Our previous work has been focusing on semi-global alignment (alignment) and global alignment (and extension). The issue with extending the A* approach to local alignment is two-fold: The search for the best local alignment is being reset when the current score becomes negative – in other words, negative scores are crucial for local alignment but not directly applicable to Dijkstra and A*. A potential solution may follow the approach of Jonson's algorithm [17] which reweights the graph to only non-negative edges as long as there are no negative cycles. For semi-global and global alignments, the whole query is being aligned, which is a useful property for the admissibility of the seed heuristic, since each seed is guaranteed to be aligned eventually. In the case of local alignment, some seeds may not be aligned. Possible workarounds have to be explored. A solution to local alignment may as well be applicable to the split alignment mode when "teleportations" are allowed. Another application of local alignment is to align reads without trimming their ends from adapters first.

Machine learning for tuning the seed heuristic. Explore the applications of machine learning to generating better admissible heuristics Tuning parameters is generally hard and it has been problematic for the seed heuristic as well. One potential place for runtime improvement is to carefully choose the seed heuristic parameters: seed length and allowed number of errors per match. Usually machine learning methods do not provide correctness guarantees about the produced results. Nevertheless, since the parametrization of the seed heuristics influences only its performance and not its admissibility and optimality of the results. Additionally, the seed heuristic parameters could be chosen or modified dynamically, during the A* search. Reinforcement learning may provide a reasonable method to tuning the parameters in real-time.

K-best paths. Explore the applications of finding K-best alignments which is trivially achievable in A* framework. A natural extension of the shortest path algorithms is to find not one shortest path, but K-shortest paths. One application of the K-best alignments is the computation of the mapq mapping score [41] which is based on the alignment costs of the K-best alignments and provides an estimation on the certainty of the best alignment. Potentially, there may be other applications.

Jointly align a set of reads. Explore joint alignment of a set of sequence to a graph. The reads to be mapped to a reference genome are highly correlated since they usually come from the same biological sample. On the other hand, they can be biased according to the reference genome, due to biological variation. Ideally, a clever alignment algorithm would reconstruct this biological variation and align the reads to the corrected reference. This problem is ill-posed and needs careful consideration to develop a theory/metric. Another complication is computational: aligning even a single query to a graph that is allowed to change may require exponential time with the query length.

Generalize the seed heuristic to multiple sequences. Explore A* for multiple sequence alignment (MSA) Existing approaches to MSA using A* consider pairwise edit distances, which do not considerably accelerate the alignment. A simultaneous seed match (anchor) within several sequences may provide much bigger penalties for suboptimal paths.

Relax the A* optimality guarantees for performance gain. Various relaxations of the heuristic admissibility, variants of A*, windowing the search (dropping expanded states that are too far behind the exploration head), have the potential to greatly increase the performance at the price of alignment optimality. This may be useful for the performance and thus the practical adoption of the algorithm but we prefer first gaining the performance from the obvious promising optimality-preserving optimizations.

FINAL THOUGHTS

Knowledge gap

It comes as a surprise that a novel approach to alignment appears 60 years after the problem was first efficiently solved. Without the intuition that we do not exploit all available information, it would have been an academical mistake to work on such a problem. Neglecting the suffix information for faster alignment may have resulted from the immense focus on new genomic data and sequencing technology and parallel computing, not believing that optimal solutions could be more efficient as a result of the near-quadratic worst case.

Challenging the seed-(chain)-extend *paradigm*

As we saw in Chapters 1 and 2, previous optimal semi-global aligners compute the whole dynamic programming table, thus reaching the prohibitively slow quadratic runtime. On the other side, all current production-level aligners rely on the *seed-*

extend paradigm or its *seed-chain-extend* variants for long reads. This paradigm searches for long alignments around shorts matches, which is a very intuitive idea if we look for a good alignment.

If we instead seek not good but provably *best* alignments, we are required to refute all the exponentially-many competing alignments as not being better. According to our seed heuristic, any seed match not seen is an inevitable edit in the future. Thus, we do not need to choose the seeds to support the best alignment but to refute the suboptimal alignment. Instead of long and good matches, we prefer many short seeds with hardly any matches. This novel usage of seeds carrie different problems and different possibilities.

Observation 7 (Seeds without matches)**.** *To efficiently find an optimal alignment using A* with the seed heuristic, seeds are not required to match (even on the resulting alignment).*

This *negated seeding* strategy has mostly opposing goals to the current seeding approaches. Studying it deeper may be beneficial for the futher developments of optimal sequence algorithms.

Education

The potential adoption of the A* approach to sequencing (incl. the usage of AStarix aligner and the A*PA global aligner) may hugely alleviate the decades-long conflict between alignment optimality and computational costs. This basic task is an extremely common element in the upstream pipelines and other high-level tools like assemblers. Possibly, many alignment tools can be outcompeted (similar to the Parasail aligner) and become outdated. The general impact of guaranteed optimality is increased trust in the tools by bioinformaticians, ability to explain any alignment, reduced number of misalignments, applicability to both linear and graph references. A risky high-stake problem we plan to explore is the multiple sequence alignment (MSA) since it has many resemblances to pairwise alignment but suffers from the combinatorial explosion with the number of sequences. Even with relaxed optimality-guarantees, a novel approach to this general problem may have a direct impact on comparative genomics, including protein folding. Our experience with presenting the A* approach convinced us that it is not a common knowledge in the computation biology community, even though most conference attendants with technical background are familiar with the name and the general usage. Our conviction is that A* is a powerful instrument that has fallen out of focus in our community but that may prove useful for other algorithmic biology problems as well. Potentially, it could become a standard topic in the education in computational biology.

1. Thue, A. *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen* **1** (Jacob Dybwad, 1912).
2. Watson, J. D. & Crick, F. H. *The structure of DNA* in *Cold Spring Harbor symposia on quantitative biology* **18** (1953), 123.
3. Bellman, R. The theory of dynamic programming. *Bulletin of the American Mathematical Society* **60**, 503 (1954).
4. Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik* **1**, 269 (1959).
5. De La Briandais, R. *File searching using variable length keys* in *Papers presented at the the March 3-5, 1959, western joint computer conference* (1959), 295.
6. Levenshtein, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 707 (1966).
7. 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).
8. Vintsyuk, T. K. Speech discrimination by dynamic programming. *Cybernetics* **4**, 52 (1968).
9. 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).
10. Sankoff, D. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences* **69**, 4 (1972).
11. Sellers, P. H. An algorithm for the distance between two finite sequences. *Journal of Combinatorial Theory* (1974).
12. Sellers, P. H. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics* **26**, 787 (1974).
13. Wagner, R. A. & Fischer, M. J. The string-to-string correction problem. *Journal of the ACM (JACM)* **21**, 168 (1974).
14. Hirschberg, D. S. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* **18**, 341 (1975).
15. Hunt, J. W. & Szymanski, T. G. A fast algorithm for computing longest common subsequences. *Communications of the ACM* **20**, 350 (1977).
16. Hirschberg, D. S. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)* **24**, 664 (1977).

17. Johnson, D. B. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)* **24**, 1 (1977).

18. Sellers, P. H. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms* **1**, 359 (1980).

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

20. Smith, T. F. & Waterman, M. S. Identification of common molecular subsequences. *Journal of molecular biology* **147**, 195 (1981).

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

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

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

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

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

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

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

28. Pearson, W. R. & Lipman, D. J. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences* **85**, 2444 (1988).

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

30. Wu, S., Manber, U., Myers, G. & Miller, W. An O(NP) sequence comparison algorithm. *Information Processing Letters* **35**, 317 (1990).

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

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

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

34. Myers, G. & Miller, W. *Chaining multiple-alignment fragments in sub-quadratic time* in *SODA* **95** (1995), 38.

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

36. Lermen, M. & Reinert, K. The practical use of the A* algorithm for exact multiple sequence alignment. *Journal of Computational Biology* **7**, 655 (2000).

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

38. McNaughton, M., Lu, P., Schaeffer, J. & Szafron, D. *Memory-Efficient A* Heuristics for Multiple Sequence Alignment.* in *AAAI/IAAI* (2002), 737.

39. Zhou, R. & Hansen, E. A. *Multiple Sequence Alignment Using Anytime A*.* in *AAAI/IAAI* (2002), 975.

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

41. Li, H., Ruan, J. & Durbin, R. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research* **18**, 1851 (2008).

42. Shendure, J. & Ji, H. Next-generation DNA sequencing. *Nature biotechnology* **26**, 1135 (2008).

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

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

45. 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).

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

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

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

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

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

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

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

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

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

56. 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).

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

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

59. 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.

60. 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).

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

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

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

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

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

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

67. 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).

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

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

70. 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).

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

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

73. 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).

74. 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).

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

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

77. 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).

78. Equi, M., Grossi, R. & Mäkinen, V. On the Complexity of Exact Pattern Matching in Graphs: Binary Strings and Bounded Degree. *arXiv:1901.05264 [cs]*. arXiv: 1901.05264 (2019).

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

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

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

82. Prjibelski, A. D., Korobeynikov, A. I. & Lapidus, A. L. in *Encyclopedia of Bioinformatics and Computational Biology* (eds Ranganathan, S., Gribskov, M., Nakai, K. & Schönbach, C.) 292 (Academic Press, Oxford, 2019).

83. 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).

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

85. 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).

86. 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).

87. 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).

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

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

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

91. Alser, M., Rotman, J., Deshpande, D., Taraszka, K., Shi, H., Baykal, P. I., Yang, H. T., Xue, V., Knyazev, S., Singer, B. D., *et al.* Technology dictates algorithms: recent developments in read alignment. *Genome biology* **22**, 1 (2021).

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

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

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

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

96. Marco-Sola, S., Eizenga, J. M., Guarracino, A., Paten, B., Garrison, E. & Moreto, M. Optimal gap-affine alignment in $O(s)$ space. *Bioinformatics* **39** (ed Martelli, P. L.) (2023).

CURRICULUM VITAE

PERSONAL DATA

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

EDUCATION

2018 – 2022	**Doctor of Sciences** Computer Science Dept. ETH Zurich, Switzerland
2015 – 2017	**Master of Science** Computer Science Dept. Shumen University, Bulgaria
2013 – 2014	**Bachelor of Science** (transcript from MSU) Computer Science Dept. Shumen University, Bulgaria
2009 – 2013	Specialist program (interrupted by visa issues) Computational Mathematics and Cybernetics Dept. Moscow State University, Russia
2008 – 2009	Bachelor program (transferred to MSU) Mathematics and Informatics Dept. Sofia University, Bulgaria
2000 – 2008	High school of Sciences, Shumen, Bulgaria

EMPLOYMENT

2014 – 2016	Software Engineer, Zurich, Google Switzerland