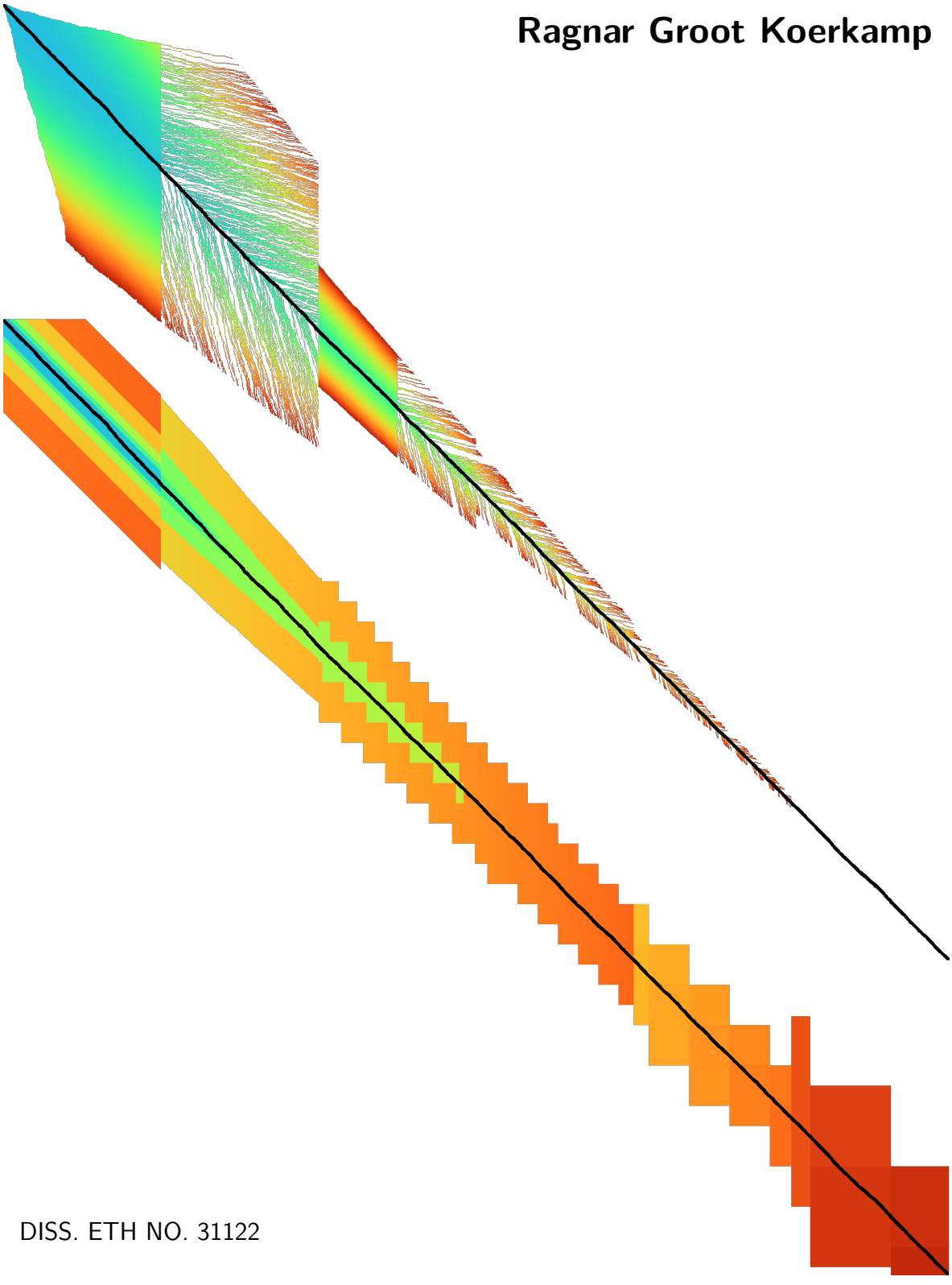
Optimal Throughput Bioinformatics

Ragnar Groot Koerkamp



DISS. ETH NO. 31122

DISS. ETH NO. 31122

Optimal Throughput Bioinformatics

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

Presented by

RAGNAR GROOT KOERKAMP

born on 22.02.1995

Accepted on the recommendation of

Prof. Dr. Gunnar Rätsch, examiner
Dr. Erik Garrison, co-examiner
Dr. Giulio Ermanno Pibiri, co-examiner

2025

https://doi.org/10.3929/ethz-c-000783091

Abstract

The amount of sequenced DNA is increasing quickly. To analyse this increasing amount of data, faster computers help, but also, faster algorithms and data structures are needed. These new methods may be fast in theory, but more importantly, they should be fast in practice.

In this thesis, our goal is to write algorithms that achieve *close-to-optimal throughput*. Ideally, we can prove that no other methods can be more than, say, twice as fast as our method. This has a few implications. Our code should use an algorithm that not only has a good *complexity*, but is also *efficient*. Further, and the focus of this thesis, this means that algorithms should be *implemented* optimally. This requires a deep understanding of how modern CPUs execute our code, so that we can ensure they do so efficiently.

We strive for optimality of a few different problems. First, we look at the problem of *pairwise alignment*: the task of finding the number of mutations between two DNA sequences. For example, one can count the number of mutations between two strains of the Sars-CoV-2 (COVID) virus. Research on this problem has a long and rich history, spanning over 50 years. We survey this history and introduce a new pairwise aligner, A*PA2, that combines a good, near-linear, complexity with a highly efficient implementation. We then extend these results to *text searching* or *mapping*, where shorter pieces of DNA must be found in longer strings.

Secondly, we consider *minimizers*. On a high level, this is a technique that takes a long DNA sequence and downsamples it to a small fraction of substrings. This is done in such a way that every sufficiently long piece of sequence contains at least one sample. We can then ask the question: what is the smallest number of samples we must select (or: the highest compression ratio we can achieve), while still satisfying this condition. We explore the theory of this problem, and give a new, near-tight, bound on the maximal compression ratio that can be achieved. We also develop minimizer schemes that reach a compression level (*density*) close to this bound.

Lastly, we look more into writing and optimizing high performance code in two categories: *compute-bound* and *memory-bound* code. We first implement a method to quickly compute minimizers, `simd-minimizers`, that is much faster than previous methods. Since this method is used to compress the input, it is the only part of a longer pipeline that works on the full data, rather than the compressed version, so that it can easily become a bottleneck.

In PtrHash, we implement a fast *minimal perfect hash function*, which is a data structure used in many applications, and specifically also in datastructures to index genomic sequences. We design a data structure that minimizes the number of memory accesses and that interleaves these as much as possible, to fully saturate the memory bandwidth of the hardware. Again, this achieves significant speedup over other methods.

In conclusion, we achieve a speedup on the order of 10× on three different problems, by using carefully designed algorithms and implementations. Thus, my thesis is that optimal software can only be achieved by designing algorithms in parallel with their implementation, and that these can not be considered independently.

II

Zusammenfassung

Die Menge an sequenzierter DNA wächst schnell. Um diese zunehmende Datenmenge zu analysieren, helfen zwar schnellere Computer, aber es werden auch schnellere Algorithmen und Datenstrukturen benötigt. Diese neuen Methoden mögen in der Theorie schnell sein, aber wichtiger ist, dass sie sich in der Praxis als schnell erweisen.

In dieser Arbeit ist unser Ziel, Algorithmen zu schreiben, die einen *nahezu optimalen Durchsatz* erreichen. Idealerweise können wir zeigen, dass keine andere Methode mehr als – sagen wir – doppelt so schnell wie unsere sein kann. Das hat einige Konsequenzen: Unser Code sollte einen Algorithmus verwenden, der nicht nur eine gute *Komplexität* hat, sondern auch *effizient* ist. Darüber hinaus – und darauf liegt der Fokus dieser Arbeit – bedeutet das, dass Algorithmen *optimal implementiert* sein müssen. Dazu ist ein tiefes Verständnis darüber erforderlich, wie moderne CPUs unseren Code ausführen, damit wir sicherstellen können, dass sie dies effizient tun.

Wir streben nach Optimalität bei verschiedenen Problemen. Zuerst betrachten wir das Problem des *Pairwise Alignment* – die Aufgabe, die Anzahl an Mutationen zwischen zwei DNA-Sequenzen zu finden. Zum Beispiel kann man die Anzahl der Mutationen zwischen zwei Varianten des Sars-CoV-2 (COVID) Virus zählen. Die Forschung zu diesem Problem hat eine lange und reichhaltige Geschichte, die über 50 Jahre zurückreicht. Wir geben einen Überblick über diese Geschichte und stellen einen neuen Pairwise-Aligner vor, *A*PA2*, der eine gute, nahezu lineare Komplexität mit einer hocheffizienten Implementierung kombiniert. Anschließend erweitern wir diese Ergebnisse auf das *Text-Suchen* bzw. *Mapping*, bei dem kürzere DNA-Stücke in längeren Sequenzen gefunden werden müssen.

Als Nächstes betrachten wir *minimizers*. Auf hoher Ebene ist das eine Technik, die eine lange DNA-Sequenz nimmt und sie auf einen kleinen Bruchteil an Teilstrings reduziert. Dies geschieht so, dass jedes ausreichend lange Teilstück der Sequenz mindestens eine dieser Proben enthält. Dann stellt sich die Frage: Wie wenige Proben müssen wir auswählen (bzw. welches höchste Kompressionsverhältnis können wir erreichen), während diese Bedingung weiterhin erfüllt ist? Wir untersuchen die Theorie dieses Problems und geben eine neue, nahezu exakte Schranke für das maximal erreichbare Kompressionsverhältnis an. Wir entwickeln außerdem Minimizer-Schemata, die eine Kompression (*density*) nahe an dieser Schranke erreichen.

Zum Schluss befassen wir uns tiefergehend mit dem Schreiben und Optimieren von Hochleistungscode in zwei Kategorien: *Compute-bound* und *Memory-bound* Code. Zuerst implementieren wir eine Methode zur schnellen Berechnung von Minimizers, `simd-minimizers`, die deutlich schneller ist als bisherige Methoden. Da diese Methode zur Komprimierung der Eingabe verwendet wird, ist sie der einzige Teil einer längeren Verarbeitungs-Pipeline, der auf den vollständigen Daten arbeitet, und kann somit leicht zum Flaschenhals werden.

In *PtrHash* implementieren wir eine schnelle *minimale perfekte Hashfunktion*, eine Datenstruktur, die in vielen Anwendungen verwendet wird – insbesondere auch beim Indexieren von genomischen Sequenzen. Wir entwerfen eine Datenstruktur, die die Anzahl der Speicherzugriffe minimiert und diese so weit wie möglich überlappt, um die Speicherbandbreite der Hardware vollständig auszunutzen.

Zusammenfassend erreichen wir durch sorgfältig entworfene Algorithmen und Implementierungen eine Beschleunigung um den 10× bei drei verschiedenen Problemen. Meine These ist daher: Optimale Software kann nur erreicht werden, wenn Algorithmusdesign und Implementierung gemeinsam gedacht werden – sie können nicht unabhängig voneinander betrachtet werden.

Acknowledgements

I will start by thanking my supervisor, Gunnar Rätsch, for taking me into the group, first for an internship, and then for this PhD. I will forever be grateful for the freedom you gave me to explore science in the unpredictable directions it led me. Thank you, André Kahles, for your co-supervision, guidance, and feedback.

Pesho Ivanov, thank you so much for both getting me in contact with the Biomedical Informatics group, and for the great time we had working on A*PA. I will forever remember the discussions we had in the Interlaken hostel and the accompanying skiing and flying on/above the slopes (including a first at landing without skis). Also, thank you so much to your parents, Svetlana and Ivelin, for hosting me in Shumen, where I got to enjoy their wonderful food, steam baths, and learned that tight underwear can be beneficial when riding horses.

Thank you, Daniel Liu, for giving me the courage to use Rust by showing the way with Block Aligner. And next, thank you for the many discussions we had online and also in person in Boston. You first got me thinking about many topics, including perfect hashing. I am also very happy we eventually got to author a paper together, and indeed, I hope we will work together more in the future.

Giulio Ermanno Pibiri, I am so happy we got to know each other. I will remember the first chats we had along the Amsterdam canals in autumn 2023, and the many conversations that followed. Thank you for hosting me twice in Venice and Padova: I had a great time. I am absolutely sure we will share many more beers and write many more papers together. Also, Elena, thank you for sharing a bit of him with me. We will eventually find (or bake) that perfect lava cake.

Thank you also to my remaining co-authors. Bryce Kille, it was so nice to discover (twice!) that we were exactly on the same track, both at RECOMB and later on. I absolutely loved our collaboration! Igor Martayan, thank you for joining the SimdMinimizers project, and motivating me to publish it. Thank you also for your repeated contributions to the slowly-growing underlying ecosystem. Rick Beeloo, I am very much enjoying our collaboration and the applied perspectice you bring to algorithm development. I hope that we will indeed be co-authors soon.

Thank you, Solon Pissis, for hosting me at CWI in Amsterdam and for many theoretical discussions, which eventually lead to the nice collaboration on the U-index. Likewise, thank you, Rob Patro, for always providing the practical context for my ideas.

To the students I supervised, Bjarni Dagur Thor Kárason, Marcel Pokorski, and František Kmječ, thank you for the passion you put into your projects.

Also many thanks to all the remaining people whom I shared discussions with, both online and in person, including Santiago Marco-Sola, Andrea Guarracino, Erik Garrison, Gauillaume Marçais, Ben Langmead, Heng Li, Barış Ekim, Seth Stadick, Karel Břinda, Sebastiano Vigna, Nicola Prezza, and many others.

Thank you, Hans-Peter Lehmann, for inspiring both the thesis template, and the matching style of the figures.

I would like to thank all the BMI lab members. In particular, thank you, Amir, for the many theoretical discussions we had, from random embeddings for tensor sketching to random neural networks. Thank you also for hosting my internship, and for sparking my interest in the theoretical understanding of neural networks. Thank you, Harun and Sayan, for the conversations we shared and your feedback on my presentations. Thank you also to the volleyball crew!

Mykola and Yura, thank you so much for spending your time with me. Mykola, I absolutely love your drive, skill, and passion – I am sure you will keep surprising me with the projects you pull off! Yura, thank you so much for keeping me company. I hope you will find your passion in studying physics, and I am excited that you are getting more into programming. To both of you, I hope you will forever look back fondly on this time, and that I was able to teach you a thing or two about life.

To my all my friends, both in Zurich and elsewhere, thank you for hanging out and exploring the world with me, and thank you for listening to my stories, rants, and opinions on science.

To my parents, Peter and Ellen, thank you for always supporting me in life, and encouraging me to walk my own path. It has always led to the right places. Quirijn and Louis, my brothers, thank you for being you, and going your own ways.

Schumi, my love, I am so grateful for our time and adventures together, and look forward to many more. And Bunny, you are the best writing companion and lap cat one can wish for.

This work was funded by ETH Research Grant ETH-17 21-1 to Gunnar Rätsch.

Table of Contents

x

1 Introduction

Over the last decades, the field of bioinformatics has grown a lot. DNA sequencing is quickly getting faster and cheaper, causing an exponential growth in the amount of sequenced data. At the same time, CPUs are also getting faster, but unfortunately, this growth does not keep up with the growth of genomic data. Thus, more and more algorithms and tools are developed to analyse this data more efficiently.

In this thesis, we continue the line of developing new tools. Naturally, we would like new methods to be faster and more efficient than previous methods, so they can keep up with the growing amount of data. We will achieve this not only by introducing new algorithms, but also by developing highly efficient implementations. In particular, modern CPUs are increasingly complex machines that apply a lot of techniques to execute any given code as fast as possible. In order to make our code as fast as possible, we must be aware of this, and write it in such a way that the CPU can indeed execute it efficiently.

Specifically, our goal will be to design solutions with *optimal throughput*. The simple interpretation is that one should be able to prove that a given piece of code solves some problem in the least possible amount of time. This typically implies a few things:

- The algorithm has optimal *complexity*, for example, linear ($O(n)$) instead of quadratic ($O(n^2)$).
- The algorithm has optimal *efficiency*, for example, a low "hidden constant" of $n/10$ instead of $100n$.
- The *implementation* of the algorithm optimally exploits the given hardware.

At times, the first two goals are at odds with each other: there may be a very inefficient linear solution (a classical example are linked lists), or a very efficient quadratic solution (such as dynamic programming, Part I). The last two goals, efficiency and implementation, are related, but slightly different. With *efficiency*, we mean the absolute *number* of (abstract) operations the code uses, while an optimal implementation efficiently *executes* these instructions, ideally by executing many of them in parallel.

Ideally, an implementation is accompanied by a proof that, indeed, the implementation is optimal and that better performance is not possible. In practice, this is often not quite possible. Nevertheless, some back-of-the-envelope calculations should ideally show that code is within a small factor of a lower bound imposed by hardware.

This thesis is divided into three parts, that each investigate a different problem.

1.1 Part 1: Pairwise Alignment

In the first part, we look at the classic problem of *pairwise alignment*. Given, for example, two DNA sequences, such as two Sars-CoV-2 (COVID) sequences, that consist of around 30 thousand bases ("DNA characters"), the task is to find the differences (mutations) between them.

The main challenge here is that as DNA sequencers get better, they output longer and longer sequences. While methods that scale quadratically with sequence length are fine for sequences up to length 10 thousand, they become slow for significantly longer sequences.

Chapter 2: A History of Pairwise Alignment. We start with a formal problem statement of pairwise alignment. Then, we review existing algorithms and techniques to implement them efficiently. The focus is on those methods that form the background for our own work.

Chapter 3: A*PA. In this chapter, we introduce *A* pairwise aligner*. The goal of A*PA is to achieve near-linear runtime on a large class of input sequences, thereby improving the quadratic complexity of most previous methods. The main technique we use is, as the name suggests, the A* shortest path algorithm. The benefit of this method is that it can use a *heuristic* that informs it about the alignment. This way, it can use *global* information to steer the search for an alignment, whereas all other methods only have the *local* picture. By using a number of optimizations, A*PA is linear-near on synthetic test data, and thus almost has the optimal $O(n)$ linear algorithmic complexity. This chapter is based on the following paper, which has shared first-authorship with Pesho Ivanov:

> **A*PA [GKI24]:** Ragnar Groot Koerkamp, Pesho Ivanov, *Exact Global Alignment using A* with Chaining Seed Heuristic and Match Pruning*, Bioinformatics 2024.

Chapter 4: A*PA2. Unfortunately, A*PA can be slow when run on real data. Specifically in regions with a lot of mutations, some local quadratic behaviour is inevitable. Because the A* algorithm is quite heavy, requiring many memory accesses, performance degrades very quickly in these cases.

In A*PA2, we improve on this. Instead of A*, which has great complexity but low efficiency, we fall back to the highly efficient methods based on dynamic programming. We are able to merge this with the good complexity of A*PA to achieve a significantly faster method.

A*PA2 balances doing *little* work (a good complexity) with doing work *fast* (a good efficiency). Compared to A*PA, this means that it is better to do 100× more work, but do this 1000× faster.

This chapter is based on the paper on A*PA2,

> **A*PA2 [GK24]:** Ragnar Groot Koerkamp, *A*PA2: Up to* 19× *Faster Exact Global Alignment*, WABI 2024.

Chapter 5: Semi-global alignment and mapping. In this last chapter on pairwise alignment, we generalize our method from *global* to *semi-global* alignment. Instead of aligning two full sequences, we now align one sequence to only a (small) part of another sequence. For example, we can search for some small known marker of length 100 in a sequenced *read* of a few thousand bases (known as *string searching*). Or we can search for a *read* of length around 10kbp (10 thousand base pairs) in a genome of 200Mbp (known as *mapping*).

The input data for this problem spans many orders of magnitude, and thus, different solutions are used. We review some variants of this problem, and adapt A*PA2 into A*Map for semi-global alignment and mapping.

1.2 Part 2: Low Density Minimizers

One way to handle the increasing amounts of sequenced biological data is by *compressing* or *sketching* the data. One sketching scheme is to compute the *minimizers* of the input: we can consider all the substrings of length k of the input (k-*mers*), and sample some subset of them. The relative size of this subset is called the *density*, and the smaller this size, the better the compression ratio. In this part, we investigate the maximal compression ratio these schemes can achieve in theory and practice, while still satisfying a number of guarantees.

There is a large number of papers on this topic, and there are many aspects to consider. Because of this, most papers touch upon multiple aspects of this problem. We attempt to somewhat untangle this situation, and cover the literature and our new contributions one topic at a time.

Chapter 6: Theory of Sampling Schemes. We start with a formal introduction of *minimizer schemes*, and also the slightly more general *sampling schemes*. We introduce how the *density* of these schemes is defined and how it can be computed, and review a number of theoretical results around this.

Chapter 7: Lower Bounds on Density. In this chapter, we review existing lower bounds on the density, that tell us something about the maximum possible compression ratio that can be achieved. As it turns out, existing lower bounds are not nearly tight. The main result is a new, near-tight lower bound. This is based on the following paper, which has shared first-authorship between Bryce Kille and myself.

> **Density lower-bound [KGKM⁺24]:** Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J. Treangen, *A Near-Tight Lower Bound on the Density of Forward Sampling Schemes*, Bioinformatics 2024.

Chapter 8: Sampling Schemes. We then turn our attention to practical minimizer and sampling schemes. We first review existing minimizer schemes, and then introduce the *open-closed minimizer* and the *mod-minimizer*. The main result is that the mod-minimizer has near-optimal density (close to the previously established lower bound) when parameters are large. This work is based on two papers:

> **Mod-minimizer [GKP24]:** Ragnar Groot Koerkamp and Giulio Ermanno Pibiri, *The mod-minimizer: A Simple and Efficient Sampling Algorithm for Long k-mers*, WABI 2024.
> **Open-closed minimizer [GKLP25]:** Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri, *The Open-Closed Mod-Minimizer Algorithm*, accepted to AMB 2025.

Chapter 9: Selection Schemes. We end the investigation of minimizers by asking the question: can we construct sampling schemes that are not just near-optimal, but *exactly* optimal? As a first step towards this goal, we consider the simple case where $k = 1$. We obtain the *anti-lexicographic sus-anchor*, which usually has density that is practically indistinguishable from optimal. But unfortunately, it does not exactly match the lower bound.

1.3 Part 3: High Throughput Bioinformatics

Lastly, we shift our attention to the efficient implementation of algorithms and data structures.

Chapter 10: Optimizing Throughput. First, we give an overview of techniques that can be used to speed up code. These are split into two categories: techniques to improve *compute-bound* code, where the executing the instructions is the bottleneck, and techniques to improve *memory-bound* code, where reading or writing from memory is the bottleneck.

Chapter 11: SimdMinimizers. As already seen, minimizers can be used as a way to obtain a smaller sketch of some input data. If the compression ratio is high, this means that the processing of this sketch can be much faster, so that the sketching in itself becomes the compute-bound bottleneck. SimdMinimizers is a highly optimized implementation of the most used minimizer method, that can be over 10× faster than previous implementations. It achieves this by using a nearly branch-free algorithm, and by using SIMD instructions to process 8 sequences in parallel.

> **SimdMinimizers [GKM25]:** Ragnar Groot Koerkamp and Igor Martayan, *SimdMinimizers: Computing Random Minimizers, Fast*, SEA 2025.

Chapter 12: PtrHash. We also investigate the memory-bound application of *minimal perfect hashing*. This data structure is an important part of the SSHash *k-mer dictionary* [Pib22], that is used in various applications in bioinformatics. In this application, a static dictionary (hashmap) is built on the set of minimizers. A minimal perfect hash function does this with only a few bits of space per key, rather than having to store the key itself. In PtrHash, we simplify previous methods to allow for a more optimized implementation and up to 4× faster queries, while only sacrificing a little bit of space.

> **PtrHash [GK25]:** Ragnar Groot Koerkamp, *PtrHash: Minimal Perfect Hashing at RAM Throughput*, SEA 2025.

Further results. I also briefly mention here one additional paper that closely relates to this thesis, but that is not otherwise a part of it: the U-index. This is again a data structure based on minimizers, that works by building an index on the sketched (compressed) representation of the text.

> **U-index [AFGK+25]:** Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, Solon P. Pissis, *U-index: A Universal Indexing Framework for Matching Long Patterns*, SEA 2025.

Part I

Pairwise Alignment

2 A History of Pairwise Alignment

Summary

In this chapter, we survey methods for *global pairwise alignment*, which is the problem of finding the minimal number of mutations between two genomic sequences.

There is a vast amount of literature on algorithms for pairwise alignment and its variants. We start with the classic DP algorithm by Needleman and Wunsch and go over a number of improvements, both algorithmic and in the implementation. We focus on those methods that A*PA (Chapter 3) and A*PA2 (Chapter 4) build on.

Attribution

This chapter is based on the introductions of the A*PA and A*PA2 papers [GKI24, GK24], and also on unpublished notes on pairwise alignment. The A*PA paper has joint first-authorship with Pesho Ivanov, and also the unpublished notes are coauthored with Pesho Ivanov.

2.1 A Brief History

The problem of *pairwise alignment* was originally introduced in genomics in 1970 by Needleman and Wunsch [NW70]. In their paper, they use their method to align proteins of around 100 amino acids. This way, they find the *mutations* between the two sequences, which can be insertions, deletions, or substitutions of single amino acids. Specifically, the *alignment* corresponds to the minimal number of such mutations that is needed to transform one sequence into the other.

Since this first paper, the sequences being aligned have grown significantly in length. For example, the mutations between different strains of the 30 000 bases long SARS-CoV-2 (COVID) virus can be found this way. Even more, full-genome alignment of 3 000 000 000 bases long human genomes is entering the picture, although here approximate rather than exact methods tend to be used.

Algorithmic improvements. Along with the increasing length and amount of genomic sequences, alignment algorithms have significantly improved since their introduction over 50 years ago. This started with a significant number of algorithmic improvements from 1970 to 1990. These first brought the runtime down from cubic ($O(n^2m)$ for sequences of length n and m) to near-linear when the sequences are similar, with the band-doubling method of Ukkonen [Ukk85a] with complexity $O(ns)$. This is still used by the popular tool Edlib [ŠŠ17]. At the same time, that complexity was further improved to $O(n + s^2)$ in expectation by Ukkonen and Myers [Ukk85a, Mye86] with the *diagonal transition* method, and also this is used in the popular BiWFA [MSMME21, MSEG+23]

aligner. BiWFA further incorporates the *divide-and-conquer* technique or Hirschberg [Hir75] to reduce its memory usage.

All this is to say: there was a large number of early algorithmic contributions, and at the core, the best methods conceptually haven't changed much since then [Med23a].

Implementation improvements. Nevertheless, the amount of genomic data has significantly increased since then, and algorithms had to speed up at the same time to keep up. As the algorithmic speedups seemed exhausted, starting in roughly 1990, the focus shifted more towards more efficient implementations of these methods. By 1999, this resulted in the $O(ns/w)$ bitpacking algorithm of Myers [Mye99], that provides up to $w = 64\times$ speedup by packing 64 adjacent states of the internal DP matrix into two 64 bit computer words. With more recent advances in computer hardware, this has also extended to SIMD instructions that can do up to 512-bit instructions, providing another up to 8× speedup [SK18].

Approximate alignment. At the same time, there also has been a rise in popularity of *approximate* alignment algorithms and tools. Unlike all methods considered so far, these are not guaranteed to find the *minimal* number of mutations between the two sequences. Rather, they sacrifice this *optimality* guarantee to achieve a speedup over exact methods. Two common techniques are x-drop [ZSWM00] and static banding, that both significantly reduce the region of the DP matrix (Figure 2.1) that needs to be computed to find an alignment.

Before proceeding with the survey, we briefly highlight the contributions A*PA and A*PA2 make.

2.1.1 A*PA

Despite all the algorithmic contributions so far, in a retrospective on pairwise alignment [Med23a], Medvedev observed that

> a major open problem is to implement an algorithm with linear-like empirical scaling on inputs where the edit distance is linear in n.

Indeed, the best algorithms so far scale as $O(s^2)$ when the edit distance is s, and this is still linear in n when the edit distance is, say, $s = 0.01 \cdot n$. A*PA attempts to break this boundary.

As we will see soon, pairwise alignment corresponds to finding the shortest path in a graph. The classic algorithm for this is Dijkstra's algorithm [Dij59]. A faster version of it that can use domain-specific information is the A* algorithm [HNR68]. It achieves this by using a *heuristic* that estimates the cost of the (remaining) alignment. In *A*PA* (Chapter 3), we take the *seed heuristic* of A*ix [IBM+20, IBV22] as a starting point and improve it to the *gap-chaining seed heuristic*, similar to [MM95], and extend it with *inexact matches*. For inputs with uniform error rate up to 10%, this can estimate the remaining distance very accurately. By additionally adding *pruning*, A*PA2 ends up being near-linear in practice when errors are uniformly distributed, improving on the quadratic behaviour of previous exact methods.

2.1.2 A*PA2

A drawback of A*PA is that it is based on plain A*, which, like Dijkstra's algorithm, is a relatively cache-inefficient graph algorithm.

On the other hand, some of the fastest aligners use DP (dynamic programming) with *bitpacking* to very efficiently compute the edit distance between sequences, even though they do not have the

near-linear scaling of A*PA. In *A*PA2* (Chapter 4), we build a highly optimized aligner. It merges the ideas of band-doubling and bit-packing (as already used by Edlib [ŠŠ17]) with SIMD and the heuristics developed for A*PA. This results in significant speedups over both A*PA and Edlib. In particular, A*PA2 is up to 1000× faster per visited state.

As Fickett stated 40 years ago [Fic84, p. 1] and still true today,

> at present one must choose between an algorithm which gives the best alignment but is expensive, and an algorithm which is fast but may not give the best alignment.

A*PA2 narrows this gap, and is nearly as fast as approximate methods.

2.1.3 Overview

The remainder of this chapter reviews the history of *exact global* pairwise alignment in more detail. In particular, we focus on those methods that A*PA and A*PA2 build on, including algorithmic improvements and implementation techniques. Rather than presenting all work strictly chronologically, we treat them topic by topic. At times, we include formal notation for the concepts we introduce, which will be useful in later chapters.

We start our survey with a formal problem statement for pairwise alignment (**??**). Then, we list a number of variants of global alignment (**??**, **??**). While these are not our focus, they can help to contextualize other existing methods. Then we move on to the classic DP algorithms in Section 2.5 and the algorithmic improvements in later sections. These are also covered in the surveys by Kruskal [Kru83] and Navarro [Nav01].

In **??** we present some theoretical results on the complexity of the pairwise alignment problem and the best worst-case methods (although not practical). We also introduce some methods for the strongly related longest common subsequence (LCS) problem (**??**). Then, in **??**, we briefly explain the methods used in some common tools that are the main baseline for the comparison of A*PA and A*PA2. We end with a table summarizing the papers discussed in this chapter, **??**.

Scope. There is also a vast literature on *text searching*, where all (approximate) occurrences of a short pattern in a long text must be found. This field has been very active since around 1990, and again includes a large number of papers. We consider these mostly out of scope and refer the reader to Navarro's survey [Nav01].

More recently, *read mapping* has become a crucial part of bioinformatics, and indeed, there is also a plethora of different tools for aligning and mapping reads. This is a generalization of text searching where patterns tend to be significantly longer (100 to 10000 of bases, rather than tens of characters). Due to the amounts of data involved, most solutions to this problem are approximate, with the notable exception of A*ix [IBM+20, IBV22], which is the precursor for the work on A*PA presented in subsequent chapters, and POASTA [vDME+24]. we refer the reader to the survey by Alser at al. [ARD+21] for a thorough overview of *many* tools and methods used for read alignment.

Lastly, we again note that most moderns read mappers and alignment tools are *approximate*, in that they are not guaranteed to return an alignment with provably minimal cost. A*PA and A*PA2 are both exact methods, and thus we will focus on these. We again refer the reader to [ARD+21].

2.2 Problem Statement

The main problem of this chapter is as follows.

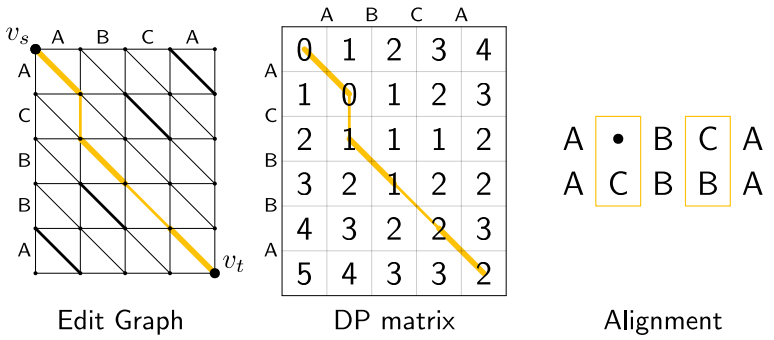


Figure 2.1 An example of an edit graph (left) corresponding to the alignment of strings `ABCA` and `ACBBA`, adapted from [Sel74]. Bold edges indicate matches of cost 0, while all other edges have cost 1. All edges are directed from the top-left to the bottom-right. The shortest path of cost 2 is highlighted. The middle shows the corresponding dynamic programming (DP) matrix containing the distance $g^*(u)$ to each state. The right shows the final alignment corresponding to the shortest path through the graph, where a `C` is inserted between the first `A` and `B`, and the initial `C` is substituted for a `B`.

▶ **Problem 2.1** (Global pairwise alignment). *Given two sequences A and B of lengths n and m, compute the edit distance* $\mathrm{ed}(A,B)$ *between them.*

Before looking into solutions to this problem, we first cover some theory to precisely define it.

Input sequences. As input, we take two sequences $A = a_0a_1\dots a_{n-1}$ and $B = b_0b_1\dots b_{m-1}$ of lengths n and m over an alphabet Σ that is typically of size $\sigma = 4$. We usually assume that $n \geq m$. We refer substrings $a_i\dots a_{i'-1}$ as $A_{i\dots i'}$ to a prefix $a_0\dots a_{i-1}$ as $A_{<i}$ and to a suffix $a_i\dots a_{n-1}$ as $A_{\geq i}$.

Edit distance. The *edit distance* $s := \mathrm{ed}(A,B)$ is the minimum number of insertions, deletions, and substitutions needed to convert A into B. In practice, we also consider the *divergence* $d := \mathrm{ed}(A,B)/n$, which is the average number of errors per character. This is different from the *error rate* e, which we consider to be the (relative) number of errors *applied* to a pair of sequence. The error rate is typically higher than the divergence, since random errors can cancel each other.

Dynamic programming. Pairwise alignment has classically been approached as a dynamic programming (DP) problem. For input strings of lengths n and m, this method creates a $(n+1)\times(m+1)$ table that is filled cell by cell using a recursive formula, as we. There are many algorithms based on DP, as we will see in Section 2.5.

Edit graph. The *alignment graph* or *edit graph* (Figure 2.1) is a way to formalize edit distance [Vin68, Ukk85a]. It contains *states* $\langle i,j\rangle$ $(0 \leq i \leq n,\ 0 \leq j \leq m)$ as vertices. It further contains edges, such that an edge of cost 0 corresponds to a pair of matching characters, and an edge of cost 1 corresponds to an insertion, deletion, or substitution. The vertical insertion and horizontal deletion edges have the form $\langle i,j\rangle \to \langle i,j+1\rangle$ and $\langle i,j\rangle \to \langle i+1,j\rangle$ of cost 1. Diagonal edges are $\langle i,j\rangle \to \langle i+1,j+1\rangle$ and have cost 0 when $A_i = B_j$ and substitution cost 1 otherwise. A shortest path from $v_s := \langle 0,0\rangle$ to $v_t := \langle n,m\rangle$ in the edit graph corresponds to an alignment of A and B. The *distance* $d(u,v)$ from u to v is the length of the shortest (minimal cost) path from u to v, and we use *edit distance*, *distance*, *length*, and *cost* interchangeably. Further we write $g^*(u) := d(v_s,u)$ for the distance from the start to u, $h^*(u) := d(u,v_t)$ for the distance from u to the end, and $f^*(u) := g^*(u) + h^*(u)$ for the minimal cost of a path through u.

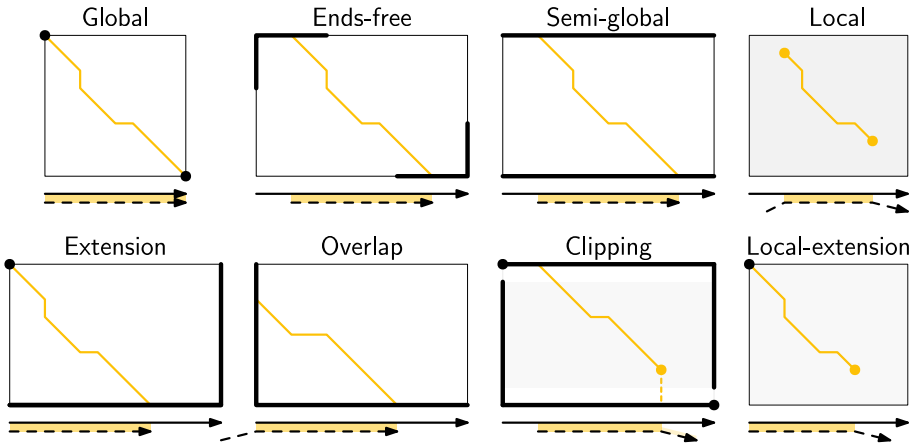


Figure 2.2 Different types of pairwise alignment. Bold vertices and edges indicate where each type of alignment may start and end. Local-extension alignment can end anywhere, whereas local alignment can also start anywhere. Like clipping, these modes require a *score* for matching characters, while in other cases, a *cost model* (with cost-0 matches) suffices. This figure is based on an earlier version that was made in collaboration with Pesho Ivanov.

In figures, we draw sequence A at the top and sequence B on the left. Index i will always be used for A and indicates a column, while index j is used for B and indicates a row.

Shortest path algorithms. Using this graph, the problem of pairwise alignment reduces to finding a shortest path in a graph. There are many shortest path algorithms for graphs, and indeed, many of them are used for pairwise alignment. Since the graph is *acyclic*, the simplest method is to greedily process the states in any topologically sorted order such as row-wise, column-wise, or anti-diagonal by anti-diagonal. We then start by setting $d(\langle 0,0\rangle) = 0$, and find the distance to any other state as the minimum distance to an incoming neighbour plus the cost of the final edge. As we will see soon, this is often implemented using *dynamic programming* (DP).

Dijkstra's shortest path algorithm, which visits states in order of increasing distance, can also be applied here [Dij59]. This does require that all edges have non-negative weights. An extension of Dijkstra's algorithm is A* [HNR68], which visits states in order of increasing "anticipated total distance".

2.3 Alignment types

There are a few variants of pairwise alignments and edit distance. While the focus of this chapter is (unit cost) edit distance, it is helpful to first have an overview of the different variants since most papers each assume a slightly different context.

In *global* pairwise alignment, the two sequences must be fully matched against each other. In practice though, there are a number of different settings, see Figure 2.2.

- **Global:** Align both sequences fully, end-to-end.
- **Ends-free:** *Ends-free* alignment allows one of the sequences on each end to have a (small) number of unmatched characters. This way, the alignment is still mostly from the top-left to the bottom-right, and global-alignment algorithms still work.

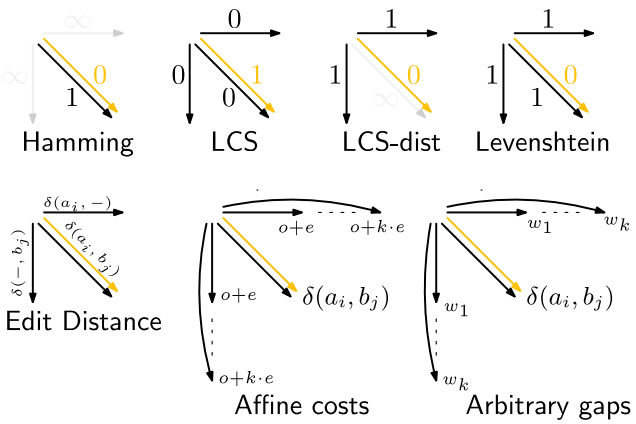


Figure 2.3 Overview of different cost models. The highlighted edge indicates the cost of matching characters, while all remaining edges indicate the cost of mismatch or indel edges. The longest common subsequence (LCS) is a *score* rather than a *cost*, and counts the maximal number of matching characters. This figure is based on an earlier version that was made in collaboration with Pesho Ivanov.

- **Semi-global:** Align a full sequence to a substring of a reference, e.g., when *mapping* a read onto a larger genome.
- **Extension:** Align one sequence to a prefix of the other.
- **Overlap:** Align two partially overlapping reads against each other.
- **Local-extension:** Align a prefix of the two sequences. Unlike all previous methods, this alignment can end *anywhere* in the edit graph. This requires a cost model that trades off matching aligned characters with not matching characters at all, and thus gives a positive *score* to the matching bases.
- **Local:** Local alignment takes this a step further and aligns a substring of A to a substring of B. This is somewhat like ends-free, but now we may skip the start/end of *both* sequences at the same time. The increased freedom in the location of the alignment prevents the faster global-alignment algorithms from working well.
- **Clipping:** When a sequence (read) is aligned onto a longer sequence (reference), it can happen that the read has, say, around 100 bp of noise at the ends that can not be aligned. When doing a local alignment, these characters will simply not be matched. It can be desirable to add an additional fixed-cost *clipping penalty* that is applied whenever the start or end of the read indeed has unaligned bases, so that effectively, an affine cost is paid for this.

In this chapter, we only consider global alignment and corresponding algorithms. These methods also work for ends-free alignment when the number of characters that may be skipped is relatively small. Later, in Chapter 5, we look deeper into semi-global alignment and its variants.

2.4 Cost Models

There are different models to specify the cost of each edit operation (Figure 2.3) [Spo91]. In particular, in a biological setting the probability of various types of mutations may not be equal, and thus, the associated costs should be different. We list some of them here, from simple to more complicated.

- **Hamming distance:** The *hamming distance* [Ham50] between two sequences is the number of substitutions required to transform one into the other, where insertions or deletions are not allowed. This is simple to compute in linear time.
- **LCS:** The *longest common subsequence* maximizes the number of matches, or equivalently, minimizes the number of *indels* (insertions or deletions) while not allowing substitutions.
- **Unit cost edit distance / Levenshtein distance:** The classic edit distance counts the minimum number of indels and/or substitutions needed, where each has a cost of 1.
- **Edit distance:** In general, the edit distance allows for arbitrary indel and substitution costs. Matches/mismatches between characters a_i and b_j have cost $\delta(a_i, b_j)$. Inserting/deleting a character has cost $\delta(\varepsilon, b_j) > 0$ and $\delta(a_i, \varepsilon) > 0$ respectively. Usually the cost of a match is 0 or negative ($\delta(a, a) \leq 0$) and the cost of a mismatch is positive ($\delta(a, b) > 0$ for $a \neq b$).
  In this chapter, when we use edit distance, we usually mean the unit-cost version.
- **Affine cost:** Insertions and deletions in DNA sequences are somewhat rare, but that once there is an indel, it is relatively common for it to be longer than a single character. This is modelled using *affine* costs [SWF81, Got82], where there is a cost o to *open* a gap, and a cost e to *extend* a gap, so that the cost of a gap of length k is $w_k = o + k \cdot e$.
  It is also possible to have different parameters $(o_{\text{ins}}, e_{\text{ins}})$ and $(o_{\text{del}}, e_{\text{del}})$ for insertions and deletions.
- **Dual affine:** Affine costs are not sufficient to capture all biological processes: the gap cost can give a too large penalty to very long indels of length 100 to 1000. To fix this, a *second* gap cost can be introduced with separate parameters (o_2, e_2), with for example an offset of $o = 1000$ and an extend cost of $e = 0.5$. The cost of a gap of length k is now given by $w_k = \min(o_1 + k \cdot e_1, o_2 + k \cdot e_2)$. More general, a piecewise linear cost can be considered as well [Got90].
- **Concave:** Even more general, we can give gaps of length k a cost w_k, where w_k is a concave function of k, where longer gaps become relatively less expensive. Double-affine costs are an example of a concave gap cost.
- **Arbitrary:** Even more general, we can merge the concave gap costs with arbitrary substitution costs $\delta(a, b)$ for (mis)matches.

In practice, most methods for global alignment use a match cost $\delta(a, a) = 0$, fixed mismatch cost $\delta(a, b) = X > 0$ for $a \neq b$, and fixed indel cost $\delta(a, \varepsilon) = \delta(\varepsilon, b) = I$.

In the chapter on semi-global alignment and mapping (blog), we discuss additional cost models used when the alignment mode is not global.

2.4.1 Minimizing Cost versus Maximizing Score

So far, most of the cost models we considered are just that: *cost* models. They focus on minimizing the cost of the edits between two sequences, and usually assume that all costs are ≥ 0, so that in particular matching two characters has a cost of 0.

In some settings, *scores* are considered instead, which are simply the negative of the cost. In this setting, matching characters usually give a positive score, so that this is explicitly rewarded. This is for example the case when finding the longest common subsequence, where each pair of matching characters gives a score of 1, and everything else has a score of 0.

Both approaches have their benefits. When using non-negative costs, all edges in the alignment graph have non-negative weights. This significantly simplifies the shortest path problem, since this is, for example, a requirement for Dijkstra's algorithm. Scores, on the other hand, work better for overlap, extension, and local alignment: in these cases, the empty alignment is usually a solution,

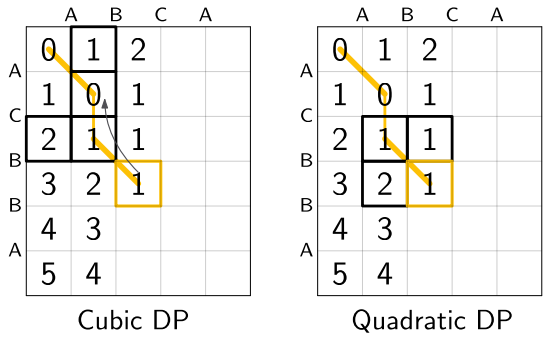


Figure 2.4 The cubic algorithm as introduced by Needleman and Wunsch [NW70]. Consider the highlighted cell. In the cubic algorithm, we first compute the cost between the two preceding characters, which are both B and thus create a match. Then, we consider all earlier cells in the the preceding row and column, and consider all gaps of arbitrary length k and cost $w_k = k$. The quadratic algorithm does not support arbitrary gap costs, but relies on $w_k = k$. This allows it to only consider three neighbouring states.

and thus, we must give some bonus to the matching of characters to compensate for the inevitable mismatches that will also occur. Unfortunately, this more general setting usually means that algorithms have to explore a larger part of the alignment graph. The ratio between the match bonus (score > 0) and mismatch penalty (score < 0) influences the trade-off between how many additional characters must be matched for each additional mismatch.

Cost-vs-score duality. For the problem of longest common subsequence there is a duality between scores and costs. When p is the length of the LCS, and s is the cost of aligning the two sequences via the LCS cost model where indels cost 1$ and mismatches are not allowed, we have

$$2 \cdot p + s = n + m. \tag{2.1}$$

Thus, maximizing the number of matched characters is equivalent to minimizing the number of insertions and deletions.

A similar duality holds for global alignment: there is a direct correspondence between maximizing score and minimizing cost [SWF81, EP22]: given a scoring model with fixed affine costs $\delta(a, a) = M$, $\delta(a, b) = X$, and $w_k = O + E \cdot k$, there is a cost-model (with $\delta(a, a) = 0$) that yields the same optimal alignment.

2.5 The Classic DP Algorithms

We are now ready to look into the first algorithms. We start with DP algorithms, that process the graph one column at a time. Note that we present all algorithms as similar as possible: they go from the top-left to the bottom-right, and always minimize the cost. We write $D(i, j) = g^*(\langle i, j \rangle)$ for the cost to state $\langle i, j \rangle$. Smith et al. [SWF81] provide a nice overview of the similarities and differences between the early approaches.

Note that for the sake of exposition, we start with the paper of Needleman and Wunsch [NW70], even though Vintsyuk [Vin68] already discovered a very similar method a few years before, although in a different domain.

Needleman-Wunsch' cubic algorithm. The problem of pairwise alignment of biological sequences was first formalized by Needleman and Wunsch [NW70]. They provide a *cubic* recurrence that

assumes a (mis)match between a_{i-1} and b_{j-1} of cost $\delta(a_{i-1}, b_{j-1})$ and an arbitrary gap cost w_k. The recursion uses that before matching a_{i-1} and b_{j-1}, either a_{i-2} and b_{j-2} are matched to each other, or one of them is matched to some other character:

$$\begin{aligned}
&D(0,0) = D(i,0) = D(0,j) := 0 \\
&D(i,j) := \delta(a_{i-1}, b_{j-1}) && \text{cost of match} \\
&\qquad + \min\Big(\min_{0 \le i' < i} D(i', j-1) + w_{i-i'-1}, && \text{cost of matching } a_{i'-1} \text{ against } b_{j-2} \text{ next} \\
&\qquad\qquad \min_{0 \le j' < j} D(i-1, j') + w_{j-j'-1} \Big). && \text{cost of matching } a_{i-2} \text{ against } b_{j'-1} \text{ next}
\end{aligned}$$

The value of $D(n, m)$ is the final cost of the alignment.

The total runtime is $O(nm \cdot (n+m)) = O(n^2 m)$ since each of the $n \cdot m$ cells requires $O(n+m)$ work.

A quadratic DP. The cubic DP was improved into a quadratic DP by Sellers [Sel74] and Wagner and Fisher [WF74]. The improvement comes from dropping the arbitrary gap cost w_k, so that instead of trying all $O(n+m)$ indels in each position, only one insertion and one deletion is tried:

$$\begin{aligned}
&D(0,0) := 0 \\
&D(i,0) := D(i-1,0) + \delta(a_i, \varepsilon) \\
&D(0,j) := D(0,j-0) + \delta(\varepsilon, b_j) \\
&D(i,j) := \min\big(D(i-1,j-1) + \delta(a_i, b_j), && \text{(mis)match} \\
&\qquad\qquad D(i-1,j) + \delta(a_i, \varepsilon), && \text{deletion} \\
&\qquad\qquad D(i,j-1) + \delta(\varepsilon, b_j) \big). && \text{insertion.}
\end{aligned}$$

This algorithm takes $O(nm)$ time since it now does constant work per DP cell.

This quadratic DP is now called the Needleman-Wunsch (NW) algorithm (Figure 2.5a, Figure 2.6a). Gotoh [Got82] refers to it as Needleman-Wunsch-Sellers' algorithm, to highlight the speedup that Sellers contributed [Sel74]. Apparently Gotoh was not aware of the identical formulation of Wagner and Fischer [WF74].

Vintsyuk published a quadratic algorithm already before Needleman and Wunsch [Vin68], but in the context of speech recognition. Instead of a cost of matching characters, there is some cost $\delta(i,j)$ associated to matching two states, and it does not allow deletions:

$$D(i,j) := \min\big(D(i-1,j-1), D(i-1,j)\big) + \delta(i,j).$$

Sankoff also gives a quadratic recursion [San72], similar to the one by Sellers [Sel74], but specifically for LCS. This leads to the recursion

$$S(i,j) := \max\big(S(i-1,j-1) + \delta(a_i, b_j),\ S(i-1,j), S(i,j-1)\big),$$

where we use S to indicate that the *score* is maximized.

Local alignment. Smith and Waterman [SW81] introduce a DP for *local* alignment. The structure of their algorithm is similar to the cubic DP of Needleman and Wunsch and allows for arbitrary gap costs w_k. While introduced as a maximization of score, here we present it as minimizing cost (with $\delta(a,a) < 0$) for consistency. The new addition is a $\min(0, \dots)$ term, that can *reset* the alignment

whenever the cost goes above 0. The best local alignment ends in the smallest value of $D(i,j)$ in the table.

$$
\begin{aligned}
D(0,0) &= D(i,0) = D(0,j) := 0 \\
D(i,j) &= \min\big(0, && \text{start a new local alignment} \\
&\qquad D(i-1,j-1) + \delta(a_{i-1}, b_{j-1}), && \text{(mis)match} \\
&\qquad \min_{0 \le i' < i} D(i',j) - w_{i-i'}, && \text{deletion} \\
&\qquad \min_{0 \le j' < j} D(i,j') - w_{j-j'}\big). && \text{insertion}
\end{aligned}
$$

This algorithm uses arbitrary gap costs w_k, as first mentioned by Needleman and Wunsch [NW70] and formally introduced by Waterman [WSB76]. Because of this, it runs in $O(n^2 m)$.

The quadratic algorithm for local alignment is now usually referred to as the Smith-Waterman-Gotoh (SWG) algorithm, since the ideas introduced by Gotoh [Got82] can be used to reduce the runtime from cubic by assuming affine costs, just like to how Sellers [Sel74] sped up the Needleman-Wunsch algorithm [NW70] for global alignment costs by assuming linear gap costs. Note though that Gotoh only mentions this speedup in passing, and that Smith and Waterman [SW81] could have directly based their idea on the quadratic algorithm of Sellers [Sel74] instead.

Affine costs. To my knowledge, the first mention of affine costs of the form $o + k \cdot e$ is by Smith, Waterman, and Fitch [SWF81]. Gotoh [Got82] generalized the quadratic recursion to these affine costs, to circumvent the cubic runtime needed for the arbitrary gap costs w_k of Waterman [WSB76]. This is done by introducing two additional matrices $P(i,j)$ and $Q(i,j)$ that contain the minimal cost to get to (i,j) where the last step is required to be an insertion or deletion respectively:

$$
\begin{aligned}
D(i,0) &= P(i,0) = I(i,0) := 0 \\
D(0,j) &= P(0,j) = I(0,j) := 0 \\
P(i,j) &:= \min\big(D(i-1,j) + o + e, && \text{new gap} \\
&\qquad P(i-1,j) + e\big) && \text{extend gap} \\
Q(i,j) &:= \min\big(D(i,j-1) + o + e, && \text{new gap} \\
&\qquad Q(i,j-1) + e\big) && \text{extend gap} \\
D(i,j) &:= \min\big(D(i-1,j-1) + \delta(a_{i-1}, b_{j-1}), && \text{(mis)match} \\
&\qquad P(i,j), && \text{close gap} \\
&\qquad Q(i,j)\big) && \text{close gap}
\end{aligned}
$$

This algorithm run in $O(nm)$ time.

Gotoh also mentions that this method can be modified to solve the local alignment of Smith and Waterman [SW81] in quadratic time. Later, Gotoh further extended the method to support double affine costs and more general piecewise linear gap costs [Got90].

Traceback. To compute the final alignment, we can follow the *trace* of the DP matrix: starting at the end $\langle n, m \rangle$, we can repeatedly determine which of the preceding DP-states was optimal as predecessor and store these states. This takes linear time, but requires quadratic memory since all states could be on the optimal path, and thus we need to keep the entire matrix in memory. Gotoh notes [Got82] that if only the final score is required, only the last two columns of the DP matrix D (and P and Q) are needed at any time, so that linear memory suffices.

2.6 Linear Memory using Divide and Conquer

Hirschberg [Hir75] introduces a divide-and-conquer algorithm to compute the LCS of two sequences in linear space. Instead of computing the full alignment from $\langle 0,0\rangle$ to $\langle n,m\rangle$, we first fix a column halfway, $i^\star = \lfloor n/2 \rfloor$. This splits the problem into two halves: we compute the *forward* DP matrix $D(i,j)$ for all $i \le i^\star$, and introduce a *backward* DP $D'(i,j)$ that is computed for all $i \ge i^\star$. Here, $D'(i,j)$ is the minimal cost for aligning suffixes of length $n-i$ and $m-j$ of A and B. It is shown that there must exist a $j^\star$ such that $D(i^\star,j^\star) + D'(i^\star,j^\star) = D(n,m)$, and we can find this $j^\star$ as the j that minimizes $D(i^\star,j) + D'(i^\star,j)$.

At this point, we know that the point $(i^\star,j^\star)$ is part of an optimal alignment. The two resulting subproblems of aligning $A[0,i^\star]$ to $B[0,j^\star]$ and $A[i^\star,n]$ to $B[j^\star,m]$ can now be solved recursively using the same technique, where again we find the midpoint of the alignment. This recursive process is shown in Figure 2.5e and Figure 2.6e. The recursion stops as soon as the alignment problem becomes trivial, or, in practice, small enough to solve with the usual quadratic-memory approach.

Space complexity. The benefit of this method is that it only uses linear memory: each forward or reverse DP is only needed to compute the scores in the final column, and thus can be done in linear memory. After the midpoint $\langle i^\star,j^\star\rangle$ is found, the results of the left and right subproblem can be discarded before recursing. Additionally, the space for the solution itself is linear.

Time complexity. We analyse the time complexity following [MM88]. The first step takes $2{\cdot}O((n/2)m) = O(nm)$ time. We are then left with two subproblems of size $i^\star{\cdot}j^\star$ and $(n{-}i^\star)(m{-}j^\star)$. Since $i^\star = n/2$, their total size is $n/2 \cdot j^\star + n/2 \cdot (m-j^\star) = nm/2$. Thus, the total time in the first layer of the recursion is $nm/2$. Extending this, we see that the total number of states halves with each level of the recursion. Thus, the total time is bounded by

$$mn + \frac{1}{2}\cdot mn + \frac{1}{4}\cdot mn + \frac{1}{8}\cdot mn + \cdots \le 2\cdot mn = O(mn).$$

Indeed, in practice this algorithm indeed takes around twice as long to find an alignment as the non-recursive algorithm takes to find just the score.

Applications. Hirschberg introduced this algorithm for computing the longest common subsequence [Hir75]. It was then applied multiple times to reduce the space complexity of other variants as well: Myers first applied it to the $O(ns)$ LCS algorithm [Mye86], and also improved the $O(nm)$ algorithm by Gotoh [Got82] to linear memory [MM88]. Similarly, BiWFA [MSEG+23] improves the space complexity of WFA from $O(n+s^2)$ to $O(s)$ working memory, where s is the cost of the alignment.

2.7 Dijkstra's Algorithm and A*

Dijkstra's algorithm. Both Ukkonen [Ukk85a] and Myers [Mye86] remarked that pairwise alignment can be solved using Dijkstra's algorithm [Dij59] (Figure 2.5b, Figure 2.6b), which visits states by increasing distance. Ukkonen gave a bound of $O(nm\log(nm))$, whereas Myers' analysis uses the fact that there are only $O(ns)$ at distance $\le s$ (see Section 2.8) and that a discrete priority queue that avoids the log is sufficient, and thus concludes that the algorithms runs in $O(ns)$.

However, Myers [Mye86, p. 2] observes that

> the resulting algorithm involves a relatively complex discrete priority queue and this queue may contain as many as $O(ns)$ entries even in the case where just the length of the [...] shortest edit script is being computed.

And indeed, I am not aware of any tool that practically implemented Dijkstra's algorithm to compute the edit distance.

A* and the gap cost heuristic. Hadlock realized [Had88] that Dijkstra's algorithm can be improved upon by using A* [HNR68, HNR72, Pea84], a more *informed* algorithm that uses a *heuristic* function h that gives a lower bound on the remaining edit distance between two suffixes. He proposes two heuristics, one based on character frequencies, and also the widely used *gap cost heuristic* [Ukk85a, Had88, Spo89, Spo91, MM95]. This uses the difference in length between two sequences as a lower bound on their edit distance (Figure 2.5f, Figure 2.6f):

$$c_{\text{gap}}(\langle i, j\rangle, \langle i', j'\rangle) = |(i - i') - (j - j')|.$$

We specifically highlight the papers by Wu et al. [WMMM90] and Papamichail and Papamichail [PP09], where the authors' method exactly matches the A* algorithm with the gap-heuristic, in combination with diagonal transition (Section Section 2.9, Figure 2.5g, Figure 2.6g).

Seed heuristic. Much more recently, A*ix [IBM+20, IBV22] introduced the much stronger *seed heuristic* for the problem of sequence-to-graph alignment. This heuristic splits the sequence A into disjoint k-mers, and uses that at least one edit is needed for each remaining k-mer that is not present in sequence B.

In A*PA [GKI24] (Chapter 3) we will improve this into the *gap-chaining seed heuristic* and add *pruning*, which results in near-linear alignment when the divergence is sufficient low.

Notation. To prepare for the theory on A*PA, we now introduce some formal terminology and notation for Dijkstra's algorithm and A*. Dijkstra's algorithm finds a shortest path from $v_s = \langle 0, 0\rangle$ to $v_t = \langle n, m\rangle$ by *expanding* (generating all successors) vertices in order of increasing distance $g^*(u)$ from the start. This next vertex to be expanded is chosen from a set of *open* vertices. The A* algorithm, instead, directs the search towards a target by expanding vertices in order of increasing $f(u) := g(u) + h(u)$, where $h(u)$ is a heuristic function that estimates the distance $h^*(u)$ to the end and $g(u) \geq g^*(u)$ is the shortest length of a path from v_s to u found so far. We say that u is *fixed* when the distance to u has been found, i.e., $g(u) = g^*(u)$. 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. A heuristic h_1 *dominates* (is *more accurate* than) another heuristic h_2 when $h_1(u) \geq h_2(u)$ for all vertices u. A dominant heuristic will usually (but not always [Hol10]) expand less vertices. Note that Dijkstra's algorithm is equivalent to A* using a heuristic that is always 0, and that both algorithms require non-negative edge costs.

We end our discussion of graph algorithms with the following observation, as Spouge states [Spo91, p. 3],

> algorithms exploiting the lattice structure of an alignment graph are usually faster,

and further [Spo89, p. 4]:

> This suggests a radical approach to A* search complexities: dispense with the lists [of open states] if there is a natural order for vertex expansion.

In A*PA2 [GK24] (Chapter 4), we follow this advice and replace the plain A* search in A*PA with a much more efficient approach based on *computational volumes* that merges DP and A*.

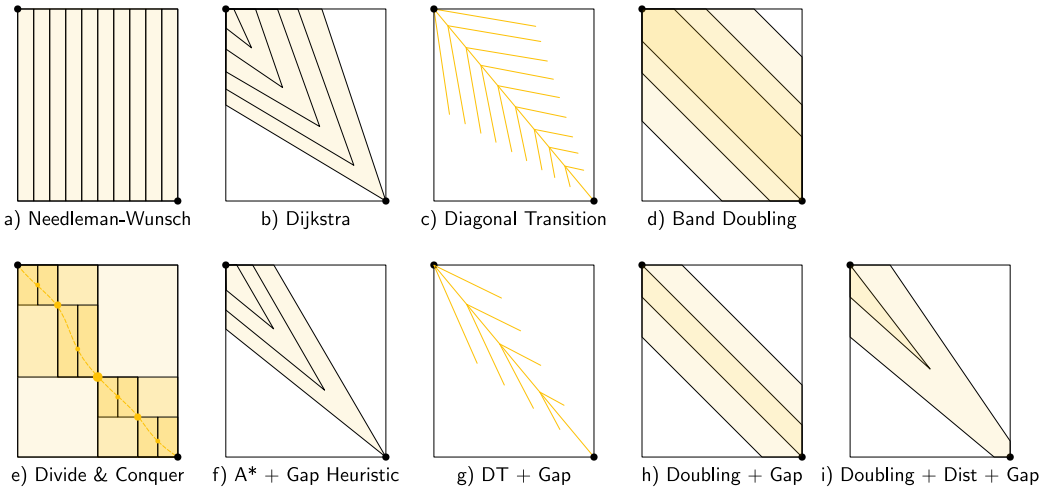


Figure 2.5 Schematic overview of global pairwise alignment methods. The shaded areas indicate states computed by each method, and darker shades indicate states that are computed multiple times. In practice, diagonal transition only computes a very sparse set of states, as indicated by lines rather than an area.

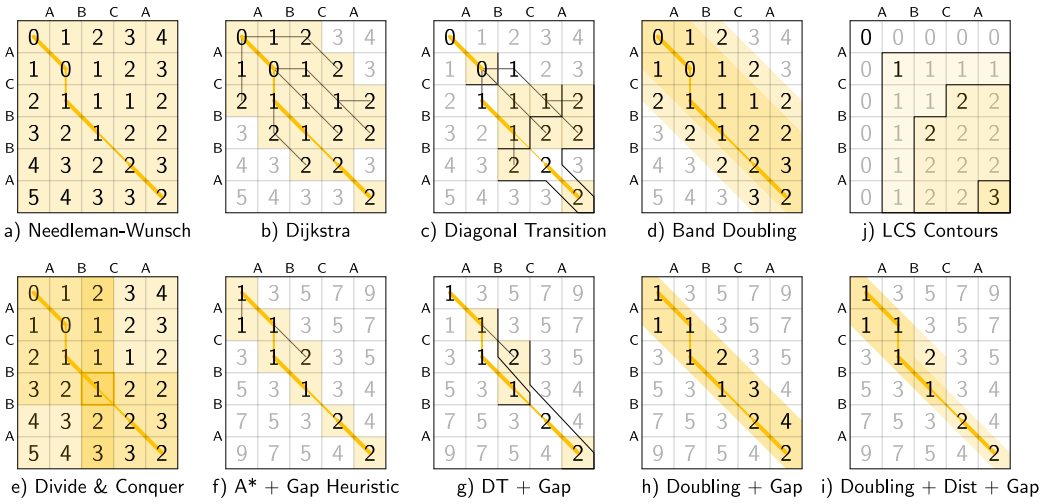


Figure 2.6 A detailed example of each method shown in Figure 2.5. Shaded areas indicate computed values, and darker shades states are computed more than once. The yellow path indicates the optimal alignment. For diagonal transition (DT), the wavefronts are indicates by black lines, and this grey lines indicate a best path to each state. The top right (j) shows contours for the longest common subsequence.

2.8 Computational Volumes and Band Doubling

So far, Dijkstra's algorithm is the only method we've seen that is faster than $\Theta(nm)$. But as remarked by Spouge, unfortunately it tends to be slow. To our knowledge, Wilbur and Lipman [WL83, WL84] are the first to give a sub-quadratic algorithm, by only considering states near diagonals with many *k-mer matches*, not unlike the approach taken by modern seed-and-extend algorithms. However, this method is not *exact*, i.e., it could return a suboptimal alignment. Nevertheless, they raise the question whether the alignments found by their method are closer to biological truth than the true minimal cost alignments found by exact algorithms.

Reordering the matrix computation. The main reason the methods so far are quadratic is that they compute the entire $n \times m$ matrix. But, especially when the two sequences are similar, the optimal alignment is likely to be close to the main diagonal. Thus, Fickett [Fic84] proposes to compute the entries of the DP matrix in a new order: Instead of column by column, we can first compute all entries at distance up to some *threshold* t, and if this does not yet result in a path to the end ($\langle n, m\rangle$), we can expand this computed area to a larger area with distance up to $t' > t$, and so on, until we try a $t \geq s$. In fact, when t is increased by 1 at a time this is equivalent to Dijkstra's algorithm (Figure 2.5b, Figure 2.6b).

Vertices at distance $\leq t$ can never be more than t diagonals away from the main diagonal, and hence, computing them can be done in $O(nt)$ time. This can be much faster than $O(nm)$ when s and t are both small, and works especially well when t is not too much larger than s. For example, t can be set as a known upper bound for the data being aligned, or as the length of some known suboptimal alignment.

Gap heuristic. In parallel, Ukkonen introduced a very similar idea [Ukk85a], *statically* bounding the computation to only those states that can be contained in a path of length at most t from the start to the end of the graph. In particular, it uses the gap heuristic, so that the minimal cost of an alignment containing $\langle i, j\rangle$ is

$$f(\langle i,j\rangle) \coloneqq c_{\text{gap}}(\langle 0,0\rangle, \langle i,j\rangle) + c_{\text{gap}}(\langle i,j\rangle, \langle n,m\rangle) = |i-j| + |(n-i)-(m-j)|.$$

Ukkonen's algorithm then only considers those states for which $f(\langle i, j\rangle) \leq t$ (Figure 2.5h, Figure 2.6h). Thus, instead that the *actual* distance to a state is at most t ($g^*(\langle i, j\rangle) \leq t$), it requires that the best possible cost of a path containing $\langle i, j\rangle$ is sufficiently low.

Band doubling. Ukkonen also introduces *band doubling* [Ukk85a]. The method of Fickett computes all states with distance up to some threshold t. The idea of band doubling is that if it turns out that $t = t_0 < s$, then it can be doubled to $t_1 = 2t_0$, $t_2 = 4t_0$, and so on, until a $t_k = 2^k \geq s$ is found. As we already saw, testing t takes $O(nt)$ time. Now suppose we test $t_0 = 1$, $t_1 = 2$, …, $t_{k-1} = 2^{k-1} < s$, up to $t_k = 2^k \geq s$. Then the total cost of this is

$$t_0 n + t_1 n + \cdots + t_k n = 1 \cdot n + 2 \cdot n + \cdots + 2^k \cdot n < 2^{k+1} \cdot n = 4 \cdot 2^{k-1} \cdot n < 4sn.$$

Thus, band doubling finds an optimal alignment in $O(ns)$ time. Note that computed values are not reused between iterations, so that each state is computed twice on average.

Two tools implementing this band doubling are Edlib and KSW2.

Computational volumes. Spouge unifies the methods of Fickett and Ukkonen in *computational volumes* [Spo89, Spo91], which are subgraphs of the full edit graph that are guaranteed to contain *all* shortest paths. Thus, to find an alignment, it is sufficient to only consider the states in such a computational volume. Given a bound $t \geq s$, some examples of computational volumes are:

1. $\{u\}$, the entire $(n+1) \times (m+1)$ graph [NW70] (Figure 2.5a, Figure 2.6a).
2. $\{u : g^*(u) \leq t\}$, the states at distance $\leq t$, introduced by Fickett [Fic84] and similar to Dijkstra's algorithm [Dij59] (Figure 2.5b, Figure 2.6b).
3. $\{u : c_{\text{gap}}(v_s, u) + c_{\text{gap}}(u, v_t) \leq t\}$ the *static* set of states possibly on a path of cost $\leq t$ [Ukk85a] (Figure 2.5h, Figure 2.6h).
4. $\{u : g^*(u) + c_{\text{gap}}(u, v_t) \leq t\}$, as used by Edlib [ŠŠ17, Spo91, PP09] (Figure 2.5i, Figure 2.6i).
5. $\{u : g^*(u) + h(u) \leq t\}$, for any admissible heuristic h, which we will use in A*PA2 and is similar to A*.

2.9 Diagonal Transition

Around 1985, the *diagonal transition* (Figure 2.5c, Figure 2.6c) algorithm was independently discovered by Ukkonen [Ukk83, Ukk85a] (for edit distance) and Myers [Mye86] (for LCS). It hinges on the observation that along diagonals of the edit graph (or DP matrix), the value of $g^*(\langle i, j\rangle) = D(i, j)$ never decreases [Ukk85a, Lemma 3], as can be seen in Figure 2.1.

We already observed before that when the edit distance is s, only the s diagonals above and below the main diagonal are needed, and on these diagonals, we only are interested in the values up to s. Thus, on each diagonal, there are at most s transition from a distance $g \leq s$ to distance $g+1$. We call the farthest state along a diagonal with a given distance a *farthest reaching state*. Specifically, given a diagonal $-s \leq k \leq s$, we consider the farthest $u = \langle i, j\rangle$ on this diagonal (i.e., with $i - j = k$) at distance g ($g^*(u) \leq g$). Then we write $F_{gk} := i + j$ to indicate the antidiagonal of this farthest reaching state. (Note that more commonly [Ukk85a, MSMME21], just the column i is used to indicate how far along diagonal $k = i - j$ the farthest reaching state can be found. Using $i + j$ leads to more symmetric formulas.) In order to write the recursive formula on the F_{gk}, we need a helper function: $\text{LCP}(i, j)$ returns the length of the longest common prefix between $A_{\geq i}$ and $B_{\geq j}$, which indicates how far we can walk along the diagonal for free starting at $u = \langle i, j\rangle$. We call this *extending* from u. The recursion then starts with $F_{00} = \text{LCP}(0, 0)$ as the farthest state along the main diagonal with cost 0. A *wavefront* is the set of farthest reaching states for a given distance, as shown by black lines in Figure 2.6c. To compute wavefront $F_{g,\bullet}$ in terms of $F_{g-1,\bullet}$, we first find the farthest state at distance g on diagonal k that is *adjacent* to a state at distance $g-1$:

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

From this state, with coordinates $i(X_{gk}) = (X_{gk} + k)/2$ and $j(X_{gk}) = (X_{gk} - k)/2$, we can possibly walk further along the diagonal for free to obtain the farthest reaching point:

$$F_{gk} = X_{gk} + \text{LCP}(i(X_{gk}), j(X_{gk})).$$

The edit distance between two sequences is then the smallest g such that $F_{g,n-m} \geq n + m$.

Time complexity. The total number of farthest reaching states is $O(s^2)$, since there are $2s+1$ diagonal within distance s, and each has at most $s+1$ farthest reaching states. The total time spent on LCP is at most $O(ns)$, since on each of the $2s+1$ diagonals, the LCP calls cover at most n characters in total. Thus, the worst case of this method is $O(ns)$. Nevertheless, Ukkonen observes

[Ukk85a] that in practice the total time needed for LCP can be small, and Myers proves [Mye86] that the LCS-version of the algorithm does run in expected $O(n + s^2)$ when we assume that the input is a random pair of sequences with distance s.

Myers also notes that the LCP can be computed in $O(1)$ by first building (in $O(n+m)$ time) a suffix tree on the input strings and then using an auxiliary data structure to answer lowest-common-ancestor queries, leading to a worst-case $O(n + s^2)$ algorithm. However, this does not perform well in practice.

We remark here that when the divergence $d = s/n$ is fixed at, say, 1%, s^2 still grows quadratically in n, and thus, in practice still method still becomes slow when the inputs become too long.

Space complexity. A naive implementation of the method requires $O(s^2)$ memory to store all values of F_{gk} (on top of the $O(n+m)$ input sequences). If only the distance is needed, only the last front has to be stored and $O(s)$ additional memory suffices. To reduce the $O(s^2)$ memory, Hirschberg's divide-and-conquer technique can also be applied here [Mye86]: we can run two instances of the search in parallel, from the start and end of the alignment graph, until they meet. Then, this meeting point must be on the optimal alignment, and we can recurse into the two sub-problems. These now have distance $s/2$, so that overall, the cost is

$$2 \cdot (s/2)^2 + 4 \cdot (s/4)^2 + 8 \cdot (s/8)^2 \cdots = s^2/2 + s^2/4 + s^2/8 + \cdots < s^2.$$

Applications. Wu et al. [WMMM90] and Papamichail and Papamichail [PP09] apply diagonal transition to align sequences of different lengths, by incorporating the gap-heuristic (Figure 2.5g, Figure 2.6g). Diagonal transition has also been extended to linear and affine costs in the *wavefront alignment* algorithm (WFA) [MSMME21] in a way similar to [Got82], by introducing multiple layers to the graph. Similar to Myers [Mye86], BiWFA [MSEG+23] applies Hirscherg's divide-and-conquer approach [Hir75] to obtain $O(s)$ memory usage (on top of the $O(n+m)$ input).

2.10 Parallelism

So far we have mostly focused on the theoretical time complexity of methods. However, since the introduction of $O(n + s^2)$ diagonal transition around 1985, no further significant breakthroughs in theoretical complexity have been found. Thus, since then, the focus has shifted away from reducing the *number* of computed states and towards computing states *faster* through more efficient implementations and more modern hardware. Most of the developments in this area were first developed for either semi-global or local alignment, but they just as much apply to global alignment.

As Spouge notes [Spo89] in the context of computational volumes:

> The order of computation (row major, column major or antidiagonal) is just a minor detail in most algorithms.

But this decision exactly at the core of most efficient implementations.

SWAR. The first technique in this direction is *microparallelism* [ACG95], nowadays also called SWAR (SIMD within a register), where each (64-bit) computer word is divided into multiple (e.g. 16-bit) parts, and word-size instructions modify all (4) parts in parallel. This can then applied with *inter-sequence parallelism* to search a given query in multiple reference sequences in parallel [ACG95, BYG92, WM92, HFN05, Rog11].

Anti-diagonals. Hughey [Hug96] notes that values along *anti-diagonals* of the DP matrix are not dependent on each other, and thus can be computed in parallel. Wozniak [Woz97] applied SIMD (single instruction, multiple data) instructions for this purpose, which are special CPU instructions that operate on multiple (currently up to 512 bits, for AVX-512) computer words at a time.

Vertical packing. Rognes and Seeberg [RS00, p. 702] also use microparallelism, but use *vertical* instead of anti-diagonal vectors:

> The advantage of this approach is the much-simplified and faster loading of the vector of substitution scores from memory. The disadvantage is that data dependencies within the vector must be handled.

Indeed, when using vertical vectors a *sequence profile* (see below) can be used to quickly determine the (mis)match score of each of the character in the vector. However, the DP cells now depend on each other, and it may be necessarily to (slowly) iterate through the values in the vector to handle insertions corresponding to vertical edges in the edit graph.

Striped SIMD. To work around the dependencies between adjacent states in each vector, Farrar [Far06] introduces an alternative *striped* SIMD scheme where lanes are interleaved with each other. Thus, the query is split into, say, 8 segments that are aligned in parallel (each in one lane of the SIMD vector). In this case, there are still dependencies between adjacent segments, and these are resolved using a separate while loop, for as long as needed. This is used by for example BSAlign [SR24].

Bitpacking. An observation that we have not used so far is that for unit cost edit distance specifically, it can be shown that the difference between distances to adjacent states is always in $\{-1, 0, +1\}$. Myers [Mye99] uses this fact to encode $w = 64$ adjacent differences into two w-bit words: one word in which bit j indicates that the j'th difference is $+1$, and one word in which bit j indicates that the j'th difference is -1. If we additionally know the difference along the top edge, Myers' method can efficiently compute the output differences of a $1 \times w$ rectangle in only 20 instructions.

We call each consecutive non-overlapping chunk of 64 rows a *lane*, so that there are $\lceil m/64 \rceil$ lanes, where the last lane may be padded. As presented originally, for text searching, this method only uses 17 instructions, but some additional instructions are needed to carry the horizontal difference to the next lane when $m > w$.

Currently, Edlib [ŠŠ17] is the most popular tool that implements bitpacking, alongside band doubling and divide-and-conquer, so that it has a complexity of $O(ns/w)$.

The supplement of BitPAl [LHB14, BHL13] introduces an alternative scheme for edit distance based on a different encoding of the $\{-1, 0, +1\}$ values, that also ends up using 20 instructions. We show implementations of both Myers' and BitPAl's method in Figure 2.7.

Profile. The DP recurrence relation depends on the sequences A and B via $\delta(a_i, b_j)$, which is 1 when $a_i \neq b_j$. When using a vertorized method, we would like to query this information efficiently for multiple pairs (i, j) at once. When using vertical vectors, this can be done efficiently using a *profile* [RS00]. For Myers' bitpacking, this looks as follows. For each character c in the alphabet, the bitvector $Eq[c]$ stores for each character b_j of B whether it equals c as a single bit. Then, when the lane for rows $j = 0$ to $j = 64$ is processed in column i, we can simply read the indicator word corresponding to these lanes from the bitvector for $c = a_i$ ($Eq[a_i]$) and directly use it in the bitwise algorithm.

```rust
pub fn compute_block_simd_myers(
    // 0 or 1. Indicates -1 difference on top.
    hp0: &mut Simd<u64, 4>,
    // 0 or 1. Indicates -1 difference on top.
    hm0: &mut Simd<u64, 4>,
    // 64-bit indicator of +1 differences on left.
    vp: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 differences on left.
    vm: &mut Simd<u64, 4>,
    // 64-bit indicator of chars equal to top char.
    eq: Simd<u64, 4>,
) {
    let vx = eq | *vm;
    let eq = eq | *hm0;
    // The addition carries information between rows.
    let hx = (((eq & *vp) + *vp) ^ *vp) | eq;
    let hp = *vm | !(hx | *vp);
    let hm = *vp & hx;
    // Extract the high bit as bottom difference.
    let right_shift = Simd::<u64, 4>::splat(63);
    let hpw = hp >> right_shift;
    let hmw = hm >> right_shift;
    // Insert the top horizontal difference.
    let left_shift = Simd::<u64, 4>::splat(1);
    let hp = (hp << left_shift) | *hp0;
    let hm = (hm << left_shift) | *hm0;
    // Update the input-output parameters.
    *hp0 = hpw;
    *hm0 = hmw;
    *vp = hm | !(vx | hp);
    *vm = hp & vx;
}
```

(a) Myers' bitpacking

```rust
pub fn compute_block_simd_bitpal(
    // 0 or 1. Indicates 0 difference on top.
    hz0: &mut Simd<u64, 4>,
    // 0 or 1. Indicates -1 difference on top.
    hp0: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 differences on left.
    vm: &mut Simd<u64, 4>,
    // 64-bit indicator of -1 and 0 differences on left.
    vmz: &mut Simd<u64, 4>,
    // 64-bit indicator of chars equal to top char.
    eq: Simd<u64, 4>,
) {
    let eq = eq | *vm;
    let ris = !eq;
    let notmi = ris | *vmz;
    let carry = *hp0 | *hz0;
    // The addition carries info between rows.
    let masksum = (notmi + *vmz + carry) & ris;
    let hz = masksum ^ notmi ^ *vm;
    let hp = *vm | (masksum & *vmz);
    // Extract the high bit as bottom difference.
    let right_shift = Simd::<u64, 4>::splat(63);
    let hzw = hz >> right_shift;
    let hpw = hp >> right_shift;
    // Insert the top horizontal difference.
    let left_shift = Simd::<u64, 4>::splat(1);
    let hz = (hz << left_shift) | *hz0;
    let hp = (hp << left_shift) | *hp0;
    // Update the input-output parameters.
    *hz0 = hzw;
    *hp0 = hpw;
    *vm = eq & hp;
    *vmz = hp | (eq & hz);
}
```

(b) Bitpal's bitpacking

Figure 2.7 Bitpacking Rust code for SIMD version of Myers' and Bitpal's bitpacking algorithms that both take 20 instructions.

For SIMD and SWAR methods that use packed integer values (rather than single bits), the same can be done, where we can simply write the values of all $\delta(a_i, b_j)$.

Difference recurrence relations. For more general cost models, such as affine costs, direct bitpacking does not work, since differences between adjacent states can be larger than 1. Still, it is beneficial to consider differences between adjacent states rather than absolute distances: these are typically smaller, so that they require fewer bits to store and more of them can be processed in parallel [SK18]. Suzuki and Kasahara introduce this technique for affine-cost alignment, and this has subsequently been used by KSW2 and BSAlign [SR24].

Blocks. A separate improvement is made by Block aligner [LS23], an approximate aligner that can also handle position-specific scoring matrices. Its main novelty is to divide the computation into large blocks. This results in highly predictable code, and benefits the execution speed, even though some more (redundant) states may be computed.

2.11 LCS and Contours

So far, all pairwise alignment methods we looked at are based on the alignment graph. The longest common subsequence problem also admits different solutions. See e.g. [BHR] for a survey.

Sparse dynamic programming. Instead of finding a minimal-cost path through a graph, we can search for the longest *chain* of matches [Hir75, Hir77, HS77]. Suppose there are r *matches* in total, where each match is a pair (i, j) such that $a_i = b_j$. We can then process these matches from left to right (by increasing i and j), and for each of them determine the longest chain of matches ending in them (Figure 2.6j). By extension, we determine for each state $\langle i, j\rangle$ the length $S(\langle i, j\rangle)$ of the LCS of $A_{<i}$ and $B_{<j}$. Such methods that only consider a subset of vertices of the graph or DP-matrix are using *sparse dynamic programming*, and are reviewed and extended in [EGGI92a, EGGI92b].

Note that S can never decrease as we move right or down the matrix, and this allows to efficiently store the values of a column via a list of *thresholds* of rows where the LCS jumps from g to $g+1$. Then, the value of a cell can be found using binary search, so that the overall algorithm runs in $O((r+n)\lg n)$. While this is slow in general, when there are only few ($r \approx n$) matches, as may be the case when comparing lines of code, this algorithm is much faster than previous quadratic methods.

Contours. The regions of equal S create a set of *contours* (Figure 2.6j), where contour ℓ is the boundary between the regions with $S(u) \geq \ell$ and $S(u) < \ell$. Each contour is determined by a set of *dominant* matches $a_i = b_j$ for which $S(i+1, j+1)$ is larger than both $S(i, j+1)$ and $S(i+1, j)$.

LCSk. We also mention here the LCSk variant, where the task is to maximize the number of length-k matches between the two input strings. This was first introduced around 1982 by Wilbur and Lipman [WL83, WL84], and rediscovered in 2014 [BLS13, PŽŠ14, PKM+17, DG14]. When choosing k sufficiently larger than $\log_\sigma n$, this has the benefit that the number of \$k\$-mer matches between the two strings is typically much smaller than n^2, so that the $O((r+n)\lg n)$ runtime becomes feasible. The drawback, however, is that this not provide an exact solution to the original LCS problem.

Chaining k-mers. A solution to the LCSk problem consist of a sequence of matching \$k\$-mers. Together, these form a *chain*, which is formally defined as a sequence of vertices $u_1, \ldots, u_n$ in a partially ordered set (whose transitive close is a directed acyclic graph), such that $u_1 \leq u_2 \leq \cdots \leq u_n$. Then, LCSk is equivalent to finding the longest chain in the poset of k-mer matches. In this formulation, a score (the length) is maximized. Myers and Miller [MM95] instead consider a version where the cost of a chain is minimized, by using the *gap cost* over the gap between consecutive k-mer matches in the chain.

2.12 Some Tools

There are many aligners that implement $O(nm)$ (semi)-global alignment using numerous of the aforementioned implementation techniques, such as SeqAn [DWRR08], Parasail [Dai16], SWIPE [Rog11], Opal [Šoš15], libssa [Fri15], SWPS3 [SLKD08], and SSW library [ZLGM13].

Dedicated global alignment implementations implementing band-doubling are much rarer, and we list some recent ones here. For more, we refer to the survey of Alser et al. [ARD+21].

KSW2 [Li16] implements banded alignment using the difference recurrence [SK18] with SIMD, and supports (double) affine costs.

Edlib [ŠŠ17] implements band doubling [Ukk85a] using the $g^*(u) + c_{\text{gap}}(u, v_t) \leq t$ computational volume, similar to A* with the gap-heuristic. It uses Myers' bitpacking [Mye99]. For traceback, it uses Hirschberg's *divide-and-conquer* approach [Hir75]: once the distance is found, the alignment

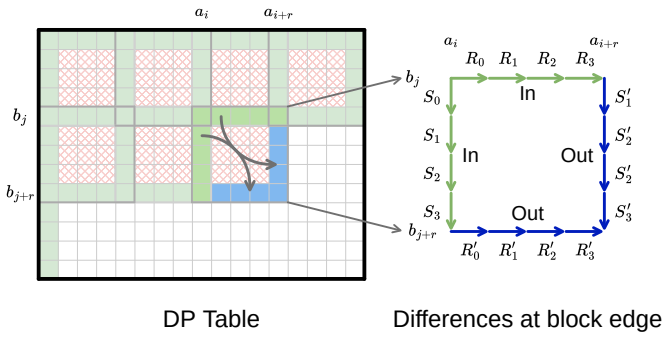


Figure 2.8 In the four Russians method, the $n \times m$ grid is divided into blocks of size $r \times r$. For each block, differences between DP table cells along the top row R and left column S are the *input*, together with the corresponding substrings of A and B. The *output* are the differences along the bottom row R' and right column S'. For each possible input of a block, the corresponding *output* is precomputed, so that the DP table can be filled by using lookups only. Red shaded states are not visited.

is started over from both sides towards the middle column, where a state on the shortest path is determined. This is recursively applied to the left and right halves until the sequences are short enough that quadratic memory can be used.

WFA [MSMME21] builds on the $O(n + s^2)$ diagonal transition method [Ukk85a, Mye86], and extends it to affine costs using a method similar to [Got82].

BiWFA [MSEG+23] is a later version that applies divide-and-conquer [Hir75] to reduce to linear memory usage.

2.13 Subquadratic Methods and Lower Bounds

We end this chapter with a discussion of some more theoretical methods that have a worst case that is slightly better than quadratic.

Lower bounds. Backurs and Indyk [BI18] have shown that unit cost edit distance can not be solved in time $O(n^{2-\delta})$ for any $\delta > 0$, on the condition that the *Strong Exponential Time Hypothesis* (SETH) is true. Soon after, it was also shown that SETH implies that LCS also can not be solved in time $O(n^{2-\delta})$ for any $\delta > 0$ [ABW15].

Four Russians method. The so called *four Russians method* was introduced by [ADKF70]. It is a general method to speed up DP algorithms from n^2 to $n^2/\log n$, provided that entries are integers and all dependencies are 'local'.

This idea was applied to pairwise alignment by Masek [MP80], resulting in the first subquadratic worst-case algorithm for edit distance. It works by partitioning the $n \times m$ matrix in blocks of size $r \times r$, for some $r = \log_k n$, as shown in Figure 2.8. Consider the differences R_i and S_i between adjacent DP cells along the top row (R_i) and left column (S_i) of the block. The core observation is that the differences R'_i and S'_i along the bottom row and right column of the block only depend on R_i, S_i, and the substrings $a_i \cdots a_{i+r}$ and $b_j \cdots b_{j+r}$. This means that for some value of k depending on the alphabet size σ, $r = \log_k n$ is small enough so that we can precompute the values of R' and S'

for all possibilities of $(R, S, a_i \cdots a_{i+r}, b_j \cdots b_{j+r})$ in $O(n^2/r^2)$ time. In practice, r needs to be quite small.

Using these precomputed values, the DP can be sped up by doing a single $O(1)$ lookup for each of the $O(n^2/r^2)$ blocks, for a total runtime of $O(n^2/\log^2 n)$. The runtime was originally reported as $O(n^2/\log n)$, but subsequent papers realized that the r differences along each block boundary fit in a single machine word, so that lookups are indeed $O(1)$ instead of $O(r)$. While this is the only known subquadratic worst-case algorithm, it does not break the $O(n^{2-\delta})$ lower bound, since $\log^2 n$ grows subpolynomial.

Masek's method requires a constant size alphabet. A first extension to general alphabets increased the time to $O(n^2(\log\log n)^2/\log^2(n))$ [BFC08], and this was later improved to $O(n^2 \log\log n/\log^2(n))$ [Gra16]. An algorithm with similar complexity also works for LCS.

Applications. Wu et al. provide an implementation of this method for approximate string matching [WMM96]. They suggest a block size of $1 \times r$, for $r = 5$ or $r = 6$, and provide efficient ways of transitioning from one block to the next.

Nowadays, the bit-parallel technique of Myers [Mye99] has replaced four Russians, since it can compute up to 64 cells in a single step, while not having to wait for (comparatively) slow lookups of the precomputed data.

2.14 Summary

We summarize most of the papers discussed in this section in chronological order in Table 2.1. Not mentioned in the table are the review papers by Smith et al. [SWF81], Kruskal [Kru83], Spouge [Spo91], and Navarro [Nav01], and also Bergroth et al.'s survey on LCS algorithms [BHR]. A more recent review on read-aligners was done by Alser et al. [ARD+21].

Table 2.1 Chronological overview of papers related to exact global pairwise alignment. Parameters are sequence lengths n and m with $n \geq m$. The (edit) distance is s. The number of matching characters or k-mers is r. The length of the LCS is p. $w = 64$ is the word size, and lastly we assume a fixed-size alphabet Σ. Time is worst-case unless noted otherwise, and space usage is to obtain the full alignment. Methods in bold are newly introduced or combined.

Paper	Cost model	Time	Space	Method	Remarks
[Vin68]	no deletions	$O(nm)$	$O(nm)$	DP	different formulation in a different domain, but conceptually similar
[NW70]	**arbitrary**	$O(n^2m)$	$O(nm)$	DP	solves pairwise alignment in polynomial time
[San72]	LCS	$\boldsymbol{O(nm)}$	$O(nm)$	DP	the first quadratic algorithm
[Sel74] and [WF74]	edit distance	$O(nm)$	$O(nm)$	DP	the quadratic algorithm now known as Needleman-Wunch
[Hir75]	LCS	$O(nm)$	$\boldsymbol{O(n)}$	divide-and-conque	introduces linear memory backtracking
[HS77]	LCS	$\boldsymbol{O((r+n)\lg n)}$	$O(r+n)$	thresholds	distance only
[Hir77]	LCS	$\boldsymbol{O(p(m-p)\lg n)}$	$\boldsymbol{O(n+(m-p)^2)}$	contours + band	for similar sequences
[MP80]	edit distance	$\boldsymbol{O(n \cdot \max(1, m/\lg n))}$	$O(n^2/\lg^2 n)$	four-russians	best known complexity; requires finite alphaet
[SWF81]	**affine**	-	-	-	First to suggest affine, in future work.
[Got82]	**affine**	$O(nm)$	$O(nm)$	DP	extends [Sel74] to affine
[AE86]	affine	$O(nm)$	$O(nm)$	DP	Fixes bug in traceback of [Got82]
[NKY82]	LCS	$\boldsymbol{O(n(m-p))}$	$O(n(m-p))$	**DP on thresholds**	improves [Hir77], subsumed by [Mye86]
[WL83, WL84]	LCSk	-	-	**chaining k-mer matches**	Approximate
[Fic84]	Edit distance	$O(nt)$	$O(nt)$	*Bound $g^*(u) \leq t$	Assuming $t \geq s$.
Dijkstra [Dij59]	Edit distance	$O(ns)$	$O(ns)$	**Dijkstra's algorithm**	Implement using $O(1)$ bucket queue
[Ukk85a]	edit distance	$\boldsymbol{O(ms)}$	$O(ns)$	band doubling	first $O(ns)$ algorithm for edit distance
[Ukk85a]	edit distance	$O(n+s^2)$ expected	$\boldsymbol{O(n+s^2)}$	**diagonal transition**	introduces diagonal transition method, requires fixed indel cost
[Mye86]	LCS	$O(n+s^s)$ expected	$O(s)$ working memory	**diagonal transition**, divide-and-conquer	introduces diagonal transition method for LCS, $O(n+s^2)$ expected time
[Mye86]	LCS	$\boldsymbol{O(n+s^2)}$	$O(n)$	suffix tree	better worst case complexity, but slower in practice
[MM88]	affine	$O(nm)$	$O(m+\lg n)$	divide-and-conquer	applies [Hir75] to [Got82] to get linear space
[Spo89]	edit distance	-	-	**A*, computational volumes**	Review paper
[Got90]	**double/more affine**	$O(Lmn)$	$O(nm+Lm)$	DP, L **layers in the graph**	
[WMMM90]	unit cost	$O(n+(s-\lvert n-m\rvert)s)$ exp.	$O(n)$	Diagonal transition, gap-heuristic, divide-and-conquer	
[EGGI92a]	LCSk	$O(n + d\log\log\min(d, nm/d))$		**sparse-dynamic-programming**, contours	d is number of *dominant* matches
[MM95]	LCSk, edit distance	$O(r\log^2 r)$	$O(r\log r)$	**chaining** k-mer matches with **gap cost**	r is number of matches
[Mye99]	unit costs	$O(nm/w)$	$O(m\sigma/w)$	DP, **bitpacking**	
[PP09]	unit costs	$O(n+(s-\lvert n-m\rvert)s)$	$O(s)$	A*, gap heuristic, diagonal transition	
[DG14]	LCSk	$O(n+r\log l)$	$O(n+\min(r, nl))$	thresholds	modifies [HS77] for LCSk
BitPAl [LHB14]	Edit distance	$O(znm/w)$	$O(znm/w)$	Bitpacking, **difference recurrence**	z depends on edit costs
[Gra16]	unit cost/LCS	$O(nm\log\log n/\log^2 n)$	$o(n)$ overhead	Four-russians	General alphabet
Edlib [ŠŠ17]	unit costs	$O(ns/w)$	$O(n)$	Bitpacking, band-doubling, divide-and-conquer	extends Myers' bit-packing to global alignment
libgaba [SK18]	Affine	-	-	**SIMD, affine difference recurrence relation**	Adaptive band; not exact
KSW2 [Li18]	Affine, Double affine	$O(nm/w)$	$O(nm/w)$	Implements [SK18]	w SIMD lanes
WFA [MSMME21]	affine	$O(n+s^2)$ expected	$O(n+s^2)$	diagonal-transition	extends diagonal transition to gap affine [Got82]
WFALM [EP22]	affine	$O(n+s^2)$	$O(n+s^{3/2})$	diagonal-transition, square-root-decomposition	reduces memory usage of WFA by only storing $1/\sqrt{n}$ of fronts
BiWFA [MSEG+23]	affine	$O(n+s^2)$ expected	$O(s)$ working memory	diagonal-transition, divide-and-conquer	applies [Hir75] to WFA to get linear space
Block Aligner [LS23]	Affine; scoring matrix			SIMD, blocks, adaptive band	
TALCO [WYB+24]	Affine			Adaptive band; **traceback convergence**	Resolves trace during alignment, saving memory
BSAlign [SR24]	Affine			striped SIMD, difference recurrence, (adaptive) banded	First to implement these together
A*PA [GKI24]	unit costs	$O(n)$ best case	$O(n)$	A*, **gap-chaining seed heuristic**, **pruning**, diagonal-transition	only for random strings with random errors, with $n \ll \lvert\Sigma\rvert^{1/e}$
A*PA2 [GK24]	unit costs	$O(n)$ best case		DP, A*, blocks, (incremental) band-doubling, SIMD, bitpacking	

3 A*PA: Exact Global Alignment using A* with Chaining Seed Heuristic and Match Pruning

Summary

In this chapter, we introduce *A* Pairwise Aligner*, A*PA. As the name suggests, this aligner uses the A* shortest path algorithm to compute the edit distance and alignment between two sequences.

In particular, this method works by finding a shortest path through the *edit graph*. To do this efficiently, it uses a *heuristic* that can quickly give a lower bound estimate on the edit distance between (suffixes of) the input sequences.

As a starting point, we take the *seed heuristic* [IBV22], that was developed for sequence-to-graph alignment. In short, it splits one of the sequences into short k-mers. Then, every k-mer that does not occur in the other sequence implies the presence of at least one edit, and the edit distance is at least the number of such k-mers absent in the other sequence.

We improve this heuristic in various ways. First, we require that the matches of these k-mers form a *chain*, similar to methods for longest common subsequence (LCS) [Hir75, Hir77, HS77] and LCSk methods [BLS13, PKM$^+$17]. Then, we add a /gap cost/ on the chaining of matches that are not on the same diagonal, like [MM95]. We extend the heuristic to *inexact matches*, so that the lack of a match of a k-mer implies the existence of at least *two* edits. We also extend the A* to be based on the diagonal transition method [Ukk85a, Mye86], rather than on the edit graph directly, so that only *farthest reaching states* are visited.

We further introduce the novel *match pruning*, which dynamically improves the A* heuristic as the search progresses, and causes the method to have near-linear runtime when the error rate is low.

On random sequences of divergence d=4% and length n, the empirical runtime of A*PA scales near-linearly with length up to $n = 10^7$ bp. At $n = 10^7$ bp, A*PA is over 500× faster than competing aligners Edlib and BiWFA. On long (n>500 kbp) reads of a human sample it efficiently aligns sequences with divergence d<10%, leading to 3× median speedup compared to Edlib and BiWFA.

Attribution

This chapter is based on the A*PA paper, "Exact global alignment using A* with chaining seed heuristic and match pruning" [GKI24], which has joint first-authorship with Pesho Ivanov. Large parts of this chapter are copied verbatim or with minor changes from that publication.

A*PA was developed in close collaboration with Pesho Ivanov. Specifically, the seed heuristic is a direct application of his earlier work on A*ix [IBM$^+$20, IBV22]. Pesho Ivanov's contributions predominantly include the extension of the heuristic to the more general seed heuristic with chaining, gaps, and inexact matches. My own contributions predominantly include match pruning, the efficient implementation of the heuristics using contours, and the proofs and experiments.

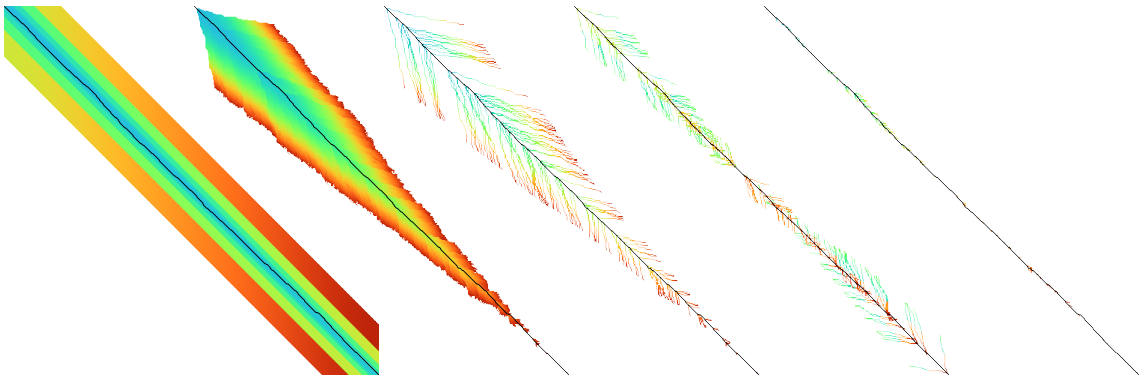


Figure 3.1 Computed states per algorithm. Various optimal alignment algorithms and their implementation are demonstrated on synthetic data (length n=500 bp, divergence d=16%). The colour indicates the order of computation from blue to red. (a) Band-doubling (Edlib), (b) Dijkstra, (c) Diagonal transition/DT (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).

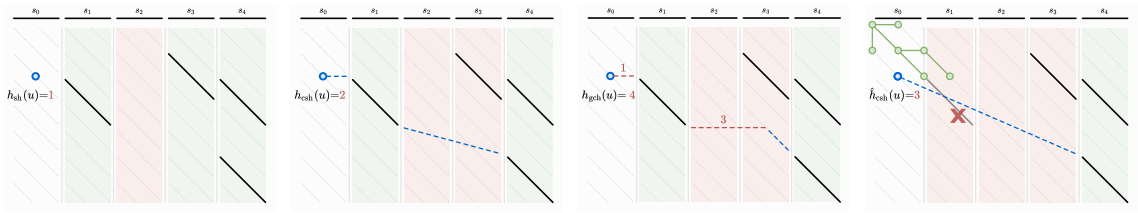
3.1 Overview

We start this chapter with an intuitive overview of A*PA (Section 3.1). In Section 3.2, we briefly re-introduce important concepts, alongside formal notation that will be needed. In the following sections we formally introduce the heuristic (Section 3.3), pruning (Section 3.4), and a method to efficiently evaluate the heuristic (Section 3.6). We end with a comparison against other methods (Section 3.7). Proofs of the main theorems can be found in (Section 3.A).

To align two sequences A and B globally with minimal cost, we use the A* shortest path algorithm from the start to the end of the alignment graph, as first suggested by Hadlock [Had88]. A core part of the A* algorithm is the heuristic function $h(u)$ that provides a lower bound on the remaining distance from the current vertex u. A good heuristic efficiently computes an accurate estimate h, so suboptimal paths get penalized more and A* prioritizes vertices on a shortest path, thus reaching the target quicker. In this paper, we extend the *seed heuristic* of A*ix [IBV22] in several ways to increase its accuracy for long and erroneous sequences.

Seed heuristic (SH). To define the *seed heuristic* h_s, we split A into short, non-overlapping substrings (*seeds*) of fixed length k (Figure 3.2a). 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. Thus, the seed heuristic h_s is the number of remaining seeds (contained in $A_{\geq i}$) that do not match anywhere in B. The seed heuristic is a lower bound on the distance between the remaining suffixes $A_{\geq i}$ and $B_{\geq j}$. In order to compute h_s efficiently, we precompute all *matches* in B for all seeds from A. Where A*ix [IBV22] uses *crumbs* to mark upcoming matches in the graph, we do not need them due to the simpler structure of sequence-to-sequence alignment.

Chaining (CSH). One drawback of the SH is that it may use matches that do not lie together on a path from u to the end, as for example the matches for s_1 and s_3 in Figure 3.2a. In the *chaining*



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

seed heuristic h_{cs} (Section 3.3), we enforce 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 right (Figure 3.2b). 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. When there are many spurious matches (i.e. outside the optimal alignment), chaining improves the accuracy of the heuristic, thus reducing the number of states expanded by A*. To compute CSH efficiently, we subtract the maximal number of matches in a chain starting in the current state from the number of remaining seeds.

Gap costs (GCSH). The CSH penalizes the chaining of two matches by the *seed cost*, the number of skipped seeds in between them. This chaining may skip a different number of letters in A and B, in which case the absolute difference between these lengths (*gap cost*) is a lower bound on the length of a path between the two matches. The *gap-chaining seed heuristic* h_{gcs} (Figure 3.2c) takes the maximum of the gap cost and the seed cost, which significantly improves the accuracy of the heuristic for sequences with long indels.

Inexact matches. To further improve the accuracy of the heuristic for divergent sequences, we use *inexact matches* [WM92, MSSGR12]. For each seed in A, our algorithm now finds all its inexact matches in B with cost at most 1. The lack of a match of a seed then implies that at least r=2 edits are needed to align it. This doubles the *potential* of our heuristic to penalize errors.

Match pruning. In order to further improve the accuracy of our heuristic, we apply the *multiple-path pruning* observation [PM17]: once a shortest path to a vertex u has been found, no other path to u can be shorter. Since we search for a single shortest path, we want to incrementally update our heuristic (similar to Real-Time Adaptive A* [KL06]) to penalize further paths to u. We prove that once A* expands a state u which is at the start or end of a match, indeed it has found

a shortest path to u. Then we can ignore (*prune*) such a match, thus penalizing other paths to u (Figure 3.2d, Section 3.4). Pruning increases the heuristic in states preceding the match, thereby penalizing states preceding the "tip" of the A* search. This reduces the number of expanded states, and leads to near-linear scaling with sequence length (Figure 3.1e).

Diagonal transition (DT). The diagonal transition algorithm only visits so called *farthest reaching* states [Ukk85a, Mye86] along each diagonal and lies at the core of WFA [MSMME21] (Figure 3.1c). We introduce the *diagonal transition* optimization to the A* algorithm that skips states known to be not farthest reaching. This is independent of the A* heuristic and makes the exploration more "hollow", especially speeding up the quadratic behavior of A* in complex regions.

We present an algorithm to efficiently initialize and evaluate these heuristics and optimizations (Sections 3.5 and 3.6), prove the correctness of our methods (Section 3.A), and evaluate and compare their performance to other optimal aligners (Section 3.7).

3.2 Preliminaries

This section provides definitions and notation that are used throughout this chapter. A summary of notation is shown in Table 3.1.

Sequences. The input sequences $A = \overline{a_0 a_1 \dots a_i \dots a_{n-1}}$ and $B = \overline{b_0 b_1 \dots b_j \dots b_{m-1}}$ are over an alphabet Σ with 4 letters. We refer to substrings $\overline{a_i \dots a_{i'-1}}$ as $A_{i\dots i'}$, to prefixes $\overline{a_0 \dots a_{i-1}}$ as $A_{<i}$, and to suffixes $\overline{a_i \dots a_{n-1}}$ as $A_{\geq i}$. The *edit distance* $\mathrm{ed}(A, B)$ is the minimum number of insertions, deletions, and substitutions of single letters needed to convert A into B. The *divergence* is the observed number of errors per letter, $d := \mathrm{ed}(A, B)/n$, whereas the *error rate* e is the number of errors per letter *applied* to a sequence.

Alignment graph. Let *state* $\langle i, j\rangle$ denote the subtask of aligning the prefix $A_{<i}$ to the prefix $B_{<j}$. The *alignment graph* (also called *edit graph*) $G(V, E)$ is a weighted directed graph with vertices $V = \{\langle i, j\rangle \mid 0 \leq i \leq n, 0 \leq j \leq m\}$ 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 u and $h^*(u) := \mathrm{d}(u, v_t)$ for the distance from u to the target.

Seeds and matches. We split the sequence A into a set of consecutive non-overlapping substrings (*seeds*) $\mathcal{S} = \{s_0, s_1, s_2, \dots, s_{\lfloor n/k\rfloor-1}\}$, such that each seed $s_l{=}A_{lk\dots lk+k}$ has length k. After aligning the first i letters of A, our heuristics will only depend on the *remaining* seeds $\mathcal{S}_{\geq i} := \{s_l \in \mathcal{S} \mid lk \geq i\}$ 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 $\mathcal{S}_{u\dots v} = \mathcal{S}_{i\dots i'} = \{s_l \in \mathcal{S} \mid i \leq lk,\, lk + k \leq i'\}$ and an *alignment* of s to a subsequence of B by π_s. The alignments of seed s with sufficiently low cost (Section 3.3) form the set $\mathcal{M}_s$ of *matches*.

Dijkstra and A*. Dijkstra's algorithm [Dij59] finds a shortest path from v_s to v_t by *expanding* (generating all successors) vertices in order of increasing distance $g^*(u)$ from the start. Each vertex to be expanded is chosen from a set of *open* vertices. The A* algorithm [HNR68, HNR72, Pea84], instead directs the search towards a target by expanding vertices in order of increasing $f(u) \coloneqq g(u) + h(u)$, 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 vertices u. A dominant heuristic will usually, but not always [Hol10], expand less vertices. Note that Dijkstra's algorithm is equivalent to A* using a heuristic that is always 0, and that both algorithms require non-negative edge costs. Our variant of the A* algorithm is provided in Algorithm 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: $c_{\text{gap}}(\langle i, j\rangle, \langle i', j'\rangle) \coloneqq |(i'-i)-(j'-j)|$. For $u \preceq v \preceq w$, the gap cost satisfies the triangle inequality $c_{\text{gap}}(u, w) \leq c_{\text{gap}}(u, v) + c_{\text{gap}}(v, w)$.

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 $u \preceq m_1 \preceq \ldots$, 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 $S(u) < \ell$ (Figure 3.3). Note that contour ℓ is completely determined by the set of matches $m \in \mathcal{M}$ for which $S(\text{start}(m)) = \ell$ [Hir77]. Hunt and Szymanski [HS77] give an algorithm to efficiently compute S when $\mathcal{M}$ is the set of single-letter matches between A and B, and Deorowicz and Grabowski [DG14] give an algorithm when $\mathcal{M}$ is the set of exact k-mer matches.

3.3 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 $\mathcal{M}_s = \varnothing$ implies that aligning the seed will incur cost at least r. Let $\mathcal{M} = \bigcup_s \mathcal{M}_s$ denote the set of all matches. With $r = 1$ we allow

Table 3.1 Summary of definitions and notations.

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

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

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: $P\langle i, j\rangle \coloneqq r \cdot \lvert\mathcal{S}_{\geq i}\rvert$.

Chaining matches. Each heuristic limits how matches can be *chained* based on a *partial order* on states. We write $u \preceq_p v$ for the partial order implied by a function p: $p(u) \leq p(v)$. 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 (Section 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 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 3.1** (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 3.2 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 3.2. Our admissibility proofs (Section 3.A.1) are based on $\mathrm{c_m}$ and γ being lower bounds on disjoint parts of the remaining path. Since the more complex h_{gcs} dominates the other heuristics it usually expand fewer states.

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

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

3.4 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 [PM17]. 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 3.2** (Pruning heuristic)**.** *Let E be the set of expanded states during the A* search, and let $\mathcal{M}\backslash E$ be the set of matches that were not pruned, i.e. those matches not starting or ending in an expanded state. We say that $\hat{h} := h^{\mathcal{M}\backslash E}$ is a* pruning heuristic *version of h.*

The hat over the heuristic function ($\hat{h}$) denotes the implicit dependency on the progress of the A*, where at each step a different $h^{\mathcal{M}\backslash E}$ is used. Our modified A* algorithm (Algorithm 1) works for pruning heuristics by ensuring that the f-value of a state is up-to-date before expanding it, and otherwise *reorders* it in the priority queue. Even though match pruning violates the admissibility of our heuristics for some vertices, we prove that A* is sill guaranteed to find a shortest path (Section 3.A.2). To this end, we show that our pruning heuristics are *weakly-admissible heuristics* (Definition 3.7) in the sense that they are admissible on at least one path from v_s to v_t.

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

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

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

3.5 A* and Diagonal Transition

A*. We now present our precise variant of A* [HNR68] that supports match pruning (Algorithm 1). All computed values of g are stored in a hash map, and all *open* states are stored in a bucket queue of tuples $(v, g(v), f(v))$ ordered by increasing f. Line 14 prunes (removes) a match and thereby increases some heuristic values before that match. As a result, some f-values in the priority queue may become outdated. Line 11 solves this problem by checking if the f-value of the state about to be expanded was changed, and if so, line 12 pushes the updated state to the queue, and proceeds by choosing a next best state. This way, we guarantee that the expanded state has minimal updated f. To reconstruct the best alignment we traceback from the target state using the hash map g (not shown).

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

$$X_{gk} := \max(F_{g-1,k-1}{+}1, F_{g-1,k}{+}2, F_{g-1,k+1}{+}1), \tag{3.1}$$

$$F_{gk} = X_{gk} + \mathrm{LCP}\left(A_{\geq(X_{gk}+k)/2}, B_{\geq(X_{gk}-k)/2}\right). \tag{3.2}$$

The base case is $X_{0,0}{=}0$ with default value $F_{gk}{=}{-}\infty$ for $k{>}|g|$, and LCP is the length of the longest common prefix of two strings. Each edge in a traceback path is either a match created by an extension (3.2), or a mismatch starting in a f.r. state (3.1). We call such a path an *f.r. path*.

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

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

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

Implementation. In Algorithm 2 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 2) still applies.

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

3.6 Evaluating the heuristic

We present an algorithm to efficiently compute our heuristics. 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 $\text{score}(m) := r - \text{c}_\text{m}(m)$ and is always positive. The *score of a* $\preceq_p$*-chain* $m_1 \preceq_p \cdots \preceq_p m_l$ is the sum of the scores of the matches in the chain. We define the chain score of a match m as

$$S_p(m) := \max_{m \preceq_p m_1 \preceq_p \cdots \preceq_p m_l \preceq_p v_t} \big\{ \text{score}(m) + \cdots + \text{score}(m_l) \big\}. \tag{3.3}$$

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

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

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

$$S_p(m) = \text{score}(m) + \max_{m \preceq_p m' \preceq v_t} S_p(m'). \tag{3.4}$$

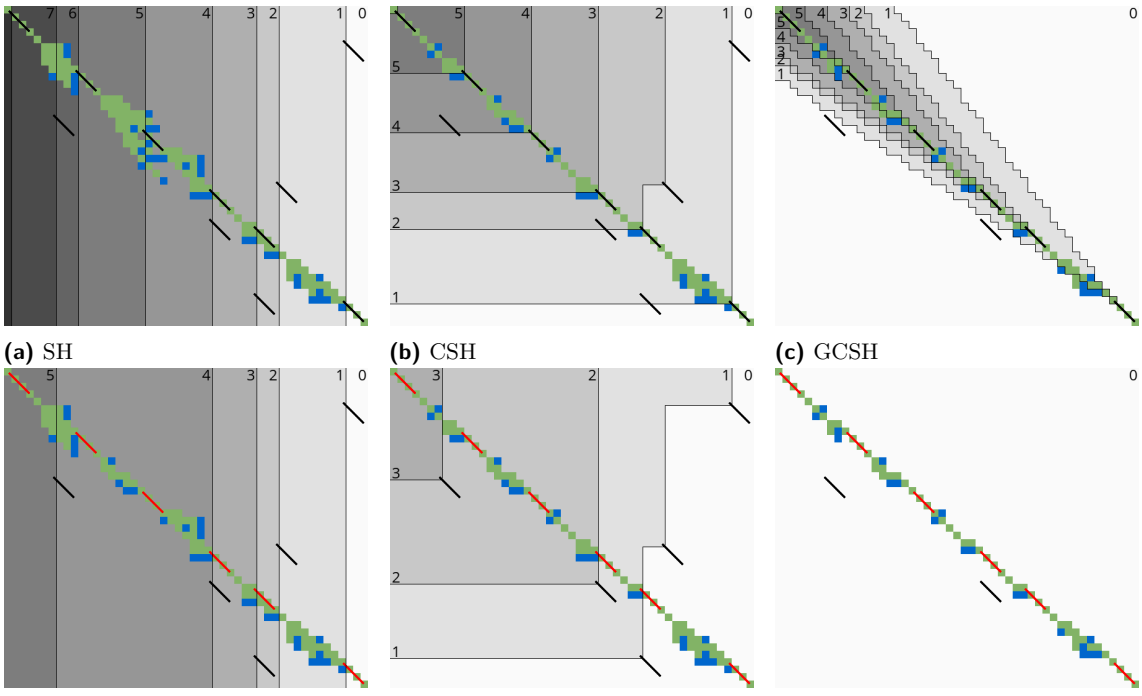
We also define the chain score of a state u as the maximum chain score over succeeding matches m: $S_p(u) = \max_{u \preceq_p m \preceq_p v_t} S_p(m)$, so that Equation (3.4) can be rewritten as $S_p(m) = \text{score}(m) + S_p(\text{end}(m))$.

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

▶ **Theorem 4.** *$h^{\mathcal{M}}_{p,c_{\text{seed}}}(u) = P(u) - S_p(u)$ 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 $\mathcal{L}_\ell$ (Figure 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 5 in Section 3.A.3).

Computing $S_p(u)$. This last observation inspires our algorithm for computing chain scores. For each layer $\mathcal{L}_\ell$, we store the set $L[i]$ of matches having score ℓ: $L[\ell] = \{m \in \mathcal{M} \mid S_p(m) = \ell\}$. The score $S_p(u)$ is then the highest ℓ such that layer $L[\ell]$ contains a match m reachable from u ($u \preceq_p m$). From Lemma 5 we know that $S_p(u) \geq \ell$ if and only if one of the layers $L[\ell']$ for $\ell' \in [\ell, \ell + r)$ contains a match preceded by u. We use this to compute $S_p(u)$ using a binary search over the



(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 $m \not\preceq_T v_t$.

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 $\ell' := S_p(m) \le \ell$ 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 $\ge \ell$. 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 4 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 Section 3.A.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 3.3** (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* score$(m)>1$*, for each* adjacent *pair of existing states $(\langle i,j\pm1\rangle, \langle i',j'\rangle)$ and $(\langle i,j\rangle, \langle i',j'\pm1\rangle)$, there is an* adjacent *match with corresponding start and end, and score at least* 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 $\mathcal{M}$.

In order to compute chains of matches when the gap cost is included, we define the following transformation, that bears some resemblance to the earlier work by Myers and Miller on chaining [MM95].

▶ **Definition 3.4** (Gap transformation). *The partial order $\preceq_T$ on states is induced by comparing both coordinates after the* gap transformation

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

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

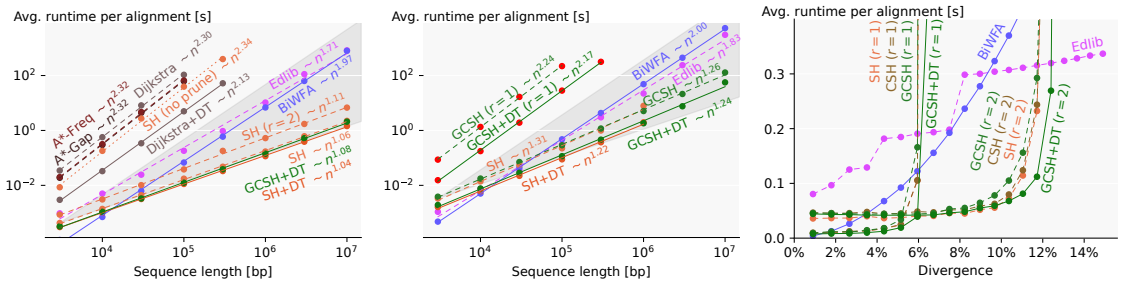
$$h^{\mathcal{M}}_{\text{gcs}}(u) = \begin{cases} P(u) - S_T(u) & \text{if } u \preceq_T v_t, \\ c_{\text{gap}}(u, v_t) & \text{if } u \not\preceq_T v_t. \end{cases}$$

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

3.7 Results

Our algorithm is implemented in the aligner A*PA[2] in Rust. We compare it with state of the art exact aligners on synthetic (Section 3.7.2) and human (Section 3.7.3) data[3] using PaBench[4]. We justify our heuristics and optimizations by comparing their scaling and performance (Section 3.7.7).

[2] github.com/RagnarGrootKoerkamp/astar-pairwise-aligner (tag evals)
[3] github.com/pairwise-alignment/pa-bench/releases/tag/datasets
[4] github.com/pairwise-alignment/pa-bench (tag astarpa-evals)



(a) Runtime by length (d=4%, r=1). **(b)** Runtime by length (d=12%, r=2). **(c)** Runtime by divergence.

Figure 3.4 Runtime comparison on synthetic data (a)(b) Log-log plots comparing variants of our heuristic, including the simplest (SH) and most accurate (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. For d=12%, red dots show where the heuristic potential is less than the edit distance. Missing data points are due to exceeding the 32 GiB memory limit. (c) Runtime scaling with divergence (n=10^5, 10^6 total bp, k=15).

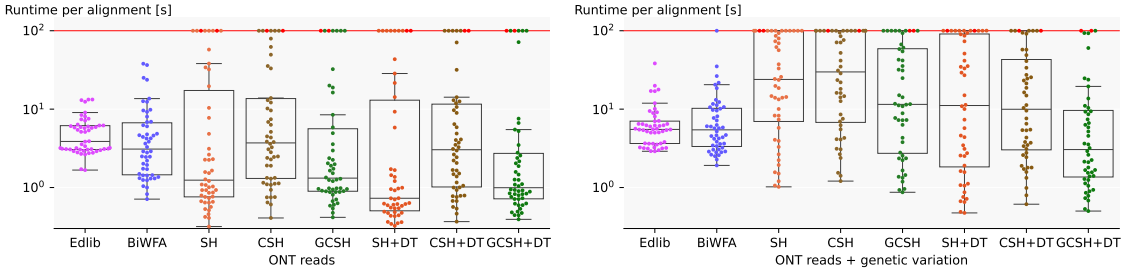


Figure 3.5 Runtime on long human reads. Each dot is an alignment without (left) or with (right) genetic variation. Runtime is capped at 100 s. Boxplots show the three quartiles, and red dots show where the edit distance is larger than the heuristic potential. The median runtime of A*PA (GCSH + DT, k=15, r=2) is 3× (left) and 1.7× (right) faster than Edlib and BiWFA.

3.7.1 Setup

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

Human data. We use two datasets of 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 3.3). The reference genome is CHM13 (v1.1) [NKR+22]. The reads used for each dataset are:

- *ONT*: 50 reads sampled from those used to assemble CHM13.

		Length [kbp]			Divergence [%]			Max gap [kbp]		
Dataset	Cnt	min	mean	max	min	mean	max	min	mean	max
ONT	50	500	594	849	3	6	18	0	0	1
ONT+gen.var.	48	502	632	1,053	4	7	20	0	**1.9**	**42**

Table 3.3 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.

- *ONT with genetic variation*: 48 reads from another human [BDH+19], as used in the BiWFA paper [MSEG+23].

Algorithms and aligners. We compare SH, CSH, and GCSH (all with pruning) as implemented in A*PA to the state-of-the-art exact aligners BiWFA and Edlib. We also compare to Dijkstra's algorithm and A* with previously introduced heuristics (gap cost and character frequencies [Had88], and SH without pruning of A*ix [IBV22]). We exclude SeqAn and Parasail since they are outperformed by WFA and Edlib [MSMME21, ŠŠ17]. We run all aligners with unit edit costs with traceback enabled.

A*PA parameters. Inexact matches (r=2) and short seeds (low k) increase the accuracy of GCSH for divergent sequences, thus reducing the number of expanded states. On the other hand, shorter seeds have more matches, slowing down precomputation and contour updates. A parameter grid search on synthetic data shows that the runtime is generally insensitive to k as long as k is high enough to avoid too many spurious matches ($k \gg \log_4 n$), and the potential is sufficiently larger than edit distance ($k \ll r/d$). For d=4%, exact matches lead to faster runtimes, while d=12% requires r=2 and $k < 2/d = 16.7$. We fix $k = 15$ throughout the evaluations since this is a good choice for both synthetic and human data.

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, and for A*PA include the time to build the heuristic. Best-fit polynomials are calculated via a linear fit in the log-log domain using the least squares method.

3.7.2 Scaling on synthetic data

Runtime scaling with length. We compare our A* heuristics with Edlib, BiWFA, and other heuristics in terms of runtime scaling with n and d (Figure 3.4). As theoretically predicted, Edlib and BiWFA scale quadratically. For small edit distance, Edlib is subquadratic due to the bit-parallel optimization. Dijkstra, A* with the gap heuristic, character frequency heuristic [Had88], or original seed heuristic [IBV22] all scale quadratically. 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 $n^{1.06}$ for

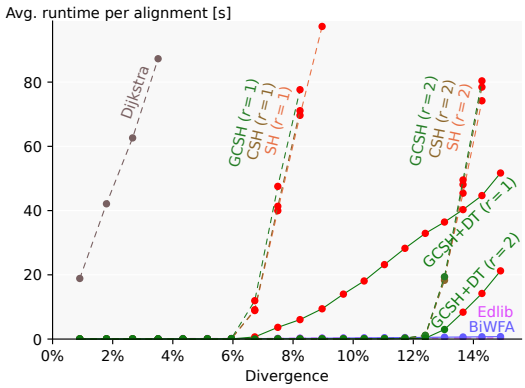


Figure 3.6 Runtime scaling with divergence (linear, synthetic, n=10^5, 10^6 bp total, k=15). The figure shows the same results as Figure 3.4c, but zoomed out to show scaling for high d, where runtime of A*PA degrades from constant in d to linear in d. r=1 and r=2 indicate exact and inexact matches, respectively. Red dots show where the heuristic potential is less than the edit distance. Missing datapoints timed out after 100 s.

$n \leq 10^7$). 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 (Section 3.7.5). GCSH with DT is not exactly linear due to high memory usage and state reordering (Section 3.7.6 shows the time spent on parts of the algorithm).

Performance. A*PA with SH with DT is >500× faster than Edlib and BiWFA for d=4% and n=10^7 (Figure 3.4a). For n=10^6 and $d \leq 12\%$, memory usage is less than 500 MB for all heuristics (Section 3.7.4).

Runtime scaling with divergence Figure 3.4c shows that A*PA has near constant runtime in d as long as the edit distance is sufficiently less than the heuristic potential (i.e. $d \ll r/k$). In this regime, A*PA is faster than both Edlib (linear in d) and BiWFA (quadratic in d). For $1 \leq d \leq 6\%$, exact matches have less overhead than inexact matches, while BiWFA is fastest for $d \leq 1\%$. A*PA becomes linear in d for $d \geq r/k$ (Figure 3.6).

Two modes. Figure 3.6 shows the runtime scaling with divergence for various heuristics. We notice two regimes of operation, depending on whether the heuristic potential P is sufficient to compensate for the edit distance: near-linear in n (constant in d) and quadratic in n (linear in d). The edit distance becomes larger than the potential P around $d = r/k$. For k=15 as in Figure 3.6, the threshold is near $d \approx 1/k = 6.7\%$ for exact matches and near $d \approx 2/k = 13.3\%$ for inexact matches. Every error not accounted for by the heuristic triggers a search "to the side", causing A* to explore $O(n)$ additional states. When using DT, only $O(s - P)$ additional farthest reaching states are explored instead, where s is the edit distance. This leads to observed runtimes of $O(n + n \cdot \max(s - P, 0))$ without DT, and $O(n + \max(s - P, 0)^2)$ with DT. These are similar to Edlib's $O(ns)$ and BiWFA's $O(n + s^2)$, but skipping over the first P errors.

3.7.3 Speedup on human data

We compare runtime (Figure 3.5, Section 3.7.6), and memory usage (Section 3.7.4) 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 contour updates. The runtime of A*PA (GCSH with DT) on ONT reads is less than Edlib and BiWFA in all quartiles, with the median being >3× faster. However, the runtime of A*PA grows rapidly when $d \geq 10\%$, so we set a time limit of 100 seconds per read, causing 6 alignments to time out. In real-world applications, the user would either only get results for a subset of alignments, or could use a different tool to align divergent sequences. With genetic variation, A*PA is 1.7× 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 (Figure 3.9). Since slow alignments dominate the total runtime, Edlib has a lower mean runtime.

3.7.4 Memory usage

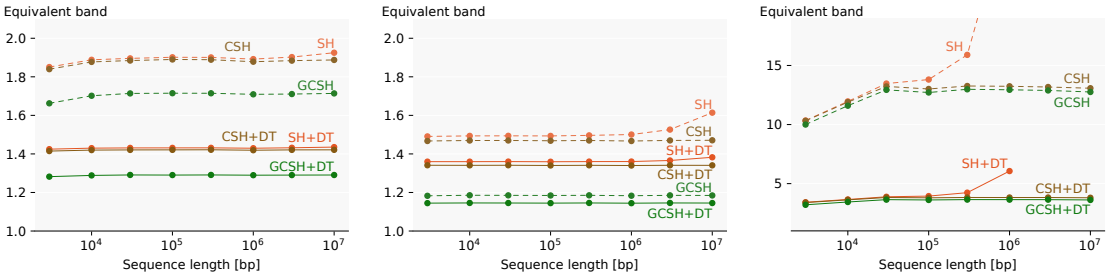
Table 3.4 compares memory usage of aligners on synthetic sequences of length $n=10^6$. Alignments with inexact matches ($d \geq 8\%$) use significantly more memory than those with exact matches because more k-mer hashes need to be stored to find inexact matches. Table 3.5 compares memory usage on the 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 3.4 Memory usage per algorithm (synthetic data, $n=10^6$). Exact matches are used when $d \leq 4\%$, and inexact matches when $d \geq 8\%$.

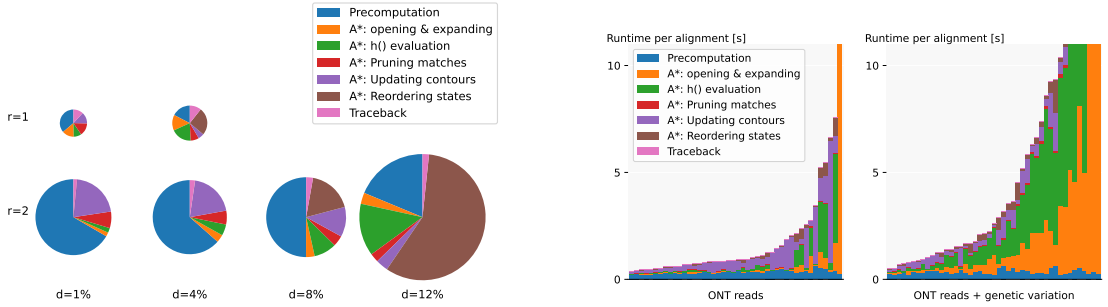
	ONT reads		+ genetic var.	
Aligner	Median	Max	Median	Max
Edlib	2	5	2	6
BiWFA	11	19	15	24
A*PA (GCSH + DT)	160	3,478	270	6,926

Table 3.5 Memory usage [MB] of aligners on human data. Medians are over all alignments; maximums are over alignments not timing out.



(a) d=4%, exact matches. **(b)** d=4%, inexact matches. **(c)** d=12%, inexact matches.

Figure 3.7 Equivalent band scaling with sequence length on synthetic data. (k=15). The equivalent band is the number of expanded states per bp for aligning synthetic sequences. Averages are over total N=10^7 bp.



(a) Synthetic data **(b)** Human data

Figure 3.8 Runtime distributions per stage of A*PA (GCSH with DT) (stages do not overlap). Stage $A*$ includes expanding and opening states. *Pruning matches* includes consistency checks. *Updating contours* includes updating of contours after pruning. (a) On synthetic data (n=10^6 bp, N=10^7 bp total, k=15). The circle area is proportional to the total runtime. Figures for r=1 and d≥8% are skipped due to timeouts (100 s). (b) On human data (r=2). Alignments are sorted by total runtime (timeouts not shown).

3.7.5 Expanded states and equivalent band

The main benefit of an A* heuristic is a lower number of expanded states, which translates to faster runtime. Instead of evaluating the runtime scaling with length (Figure 3.4a), we can judge how well a heuristic approximates the edit distance by directly measuring the *equivalent band* (Figure 3.7) 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 with inexact matches starts to grow around $n \geq 3 \cdot 10^6$ at divergence d=4%, and around $n \geq 3 \cdot 10^5$ 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 lower than CSH due to better accounting for indels. The DT variants expand fewer states by skipping non-farthest reaching states, also lowering the equivalent band.

3.7.6 Runtime profile of A*PA

Figures 3.8a and 3.8b compare the time used by stages of A*PA. On synthetic data with exact matches (r=1), the runtime is spread over all parts of the algorithm. When using inexact matches, precomputation takes a significant fraction of the total time and updating contours becomes slower due to the increased number of matches.

On human data, faster alignments spend a large fraction of their time on the precomputation, followed by the updating of contours after pruning matches. Slower alignments on the other hand are limited by the performance of the A* algorithm, and spend a large fraction of time on opening and expanding states, and evaluating the heuristic.

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

We visualize our techniques on a complex alignment in Figure 3.10.

SH with pruning enables near-linear runtime. Figure 3.4a shows that the addition of match pruning changes the quadratic runtime of SH without pruning to near-linear, giving multiple orders of magnitude speedup.

Inexact matches cope with higher divergence. Inexact matches double the heuristic potential, thereby almost doubling the divergence up to which A*PA is fast (Figure 3.4c). This comes at the cost of a slower precomputation to find all matches.

Chaining copes with spurious matches. While CSH improves on SH for some very slow alignments (Figure 3.5), more often the overhead of computing contours makes it slower than SH.

Gap-chaining copes with indels. GCSH is significantly and consistently faster than SH and CSH on human data, especially for slow alignments (Figure 3.5). GCSH has less overhead over SH than CSH, due to filtering out matches $m \not\preceq v_t$.

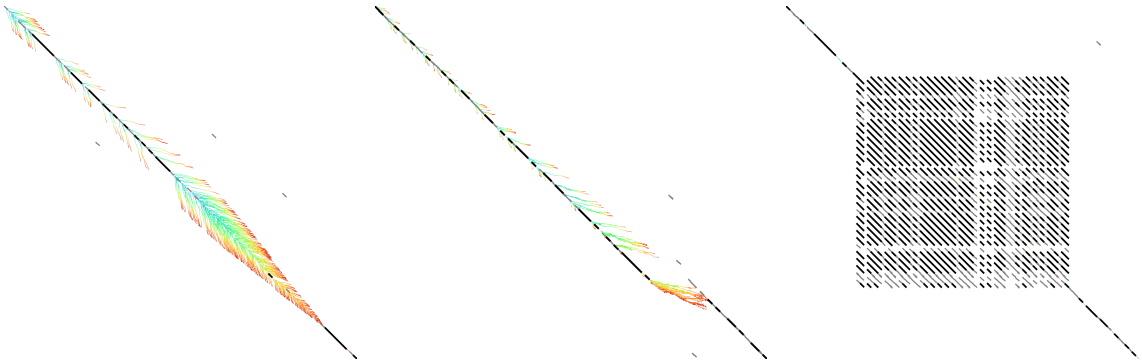
Diagonal transition speeds up quadratic search. DT significantly reduces the number of expanded states when the A* search is quadratic (Figures 3.4a and 3.6). In particular, this results in a big speedup when aligning genetic variation containing big indels (Figure 3.5).

CSH, GCSH, and DT only have a small impact on the uniform synthetic data, where usually either the SH is sufficiently accurate for the entire alignment and runtime is near-linear ($d \ll r/k$), or even GCSH is not strong enough and runtime is quadratic ($d \gg r/k$). On human data however, containing longer indels and small regions of quadratic search, the additional accuracy of GCSH and the reduced number of states explored by DT provide a significant speedup (Section 3.8.1).

3.8 Discussion

Seeds are necessary, matches are optional. The seed heuristic uses 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.

Modes: Near-linear and quadratic. The A* algorithm with a seed heuristic has two modes of operation that we call *near-linear* and *quadratic*. In the near-linear mode A*PA expands few vertices because the heuristic successfully penalizes all edits between the sequences. When the



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

Figure 3.9 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.

divergence is larger than what the heuristic can handle, every edit that is not penalized by the heuristic increases the explored band, leading to a quadratic exploration similar to Dijkstra.

Limitations

1. *Quadratic scaling.* Complex data can trigger a quadratic (Dijkstra-like) search, which nullifies the benefits of A* (Figures 3.9 and 3.10). Regions with high divergence (d≥10%), such as high error rate or long indels, exceed the heuristic potential to direct the search and make the exploration quadratic. Low-complexity regions (e.g. with repeats) result in a quadratic number of matches which also take quadratic time.
2. *Computational overhead of A*.* Computing states sequentially (as in Edlib, BiWFA) is orders of magnitude faster than computing them in random order (as in Dijkstra, A*). A*PA outperforms Edlib and BiWFA (Figure 3.4a) when the sequences are long enough for the linear-scaling to have an effect (n>30 kbp), and there are enough errors (d>1%) to trigger the quadratic behaviour of BiWFA.

Future work

1. *Performance.* We are working on a DP-based version of A*PA that applies computational volumes [Spo89, Spo91], block-based computations [LS23], and a SIMD version of Edlib's bit-parallelization [Mye99]. This has already shown 10× additional speedup on the human data sets and is less sensitive to the input parameters. Independently, the number of matches could be reduced by using variable seed lengths and skipping seeds with many matches.
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 with length of A* is not asymptotic and requires a more thorough theoretical analysis [Med23b].

3.8.1 Comparison of heuristics and techniques

Figure 3.10 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.

3.A Proofs

3.A.1 Admissibility

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

▶ **Definition 3.5** (Start of seed). *A state $\langle i, j\rangle$ is* at the start of some seed *when i is a multiple of the seed length k, or when $i = n$.*

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

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

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

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

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

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

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

◀

▶ **Lemma 2** (Weak consistency). *Let $h \in \{h^{\mathcal{M}}_{\text{s}}, h^{\mathcal{M}}_{\text{cs}}, h^{\mathcal{M}}_{\text{gcs}}\}$ 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 (Figure 3.11). Since the matches and the paths between them are disjoint, $\mathrm{c}_{\text{path}}(\pi^*)$ is at least the cost of the matches $\mathrm{c}_{\text{m}}(m_{i+1}) = \mathrm{d}(\text{start}(m_{i+1}), \text{end}(m_{i+1}))$ plus the cost to chain these matches $\gamma(\text{end}(m_i), \text{start}(m_{i+1})) \leq \mathrm{d}(\text{end}(m_i), \text{start}(m_{i+1}))$. Putting this together:

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

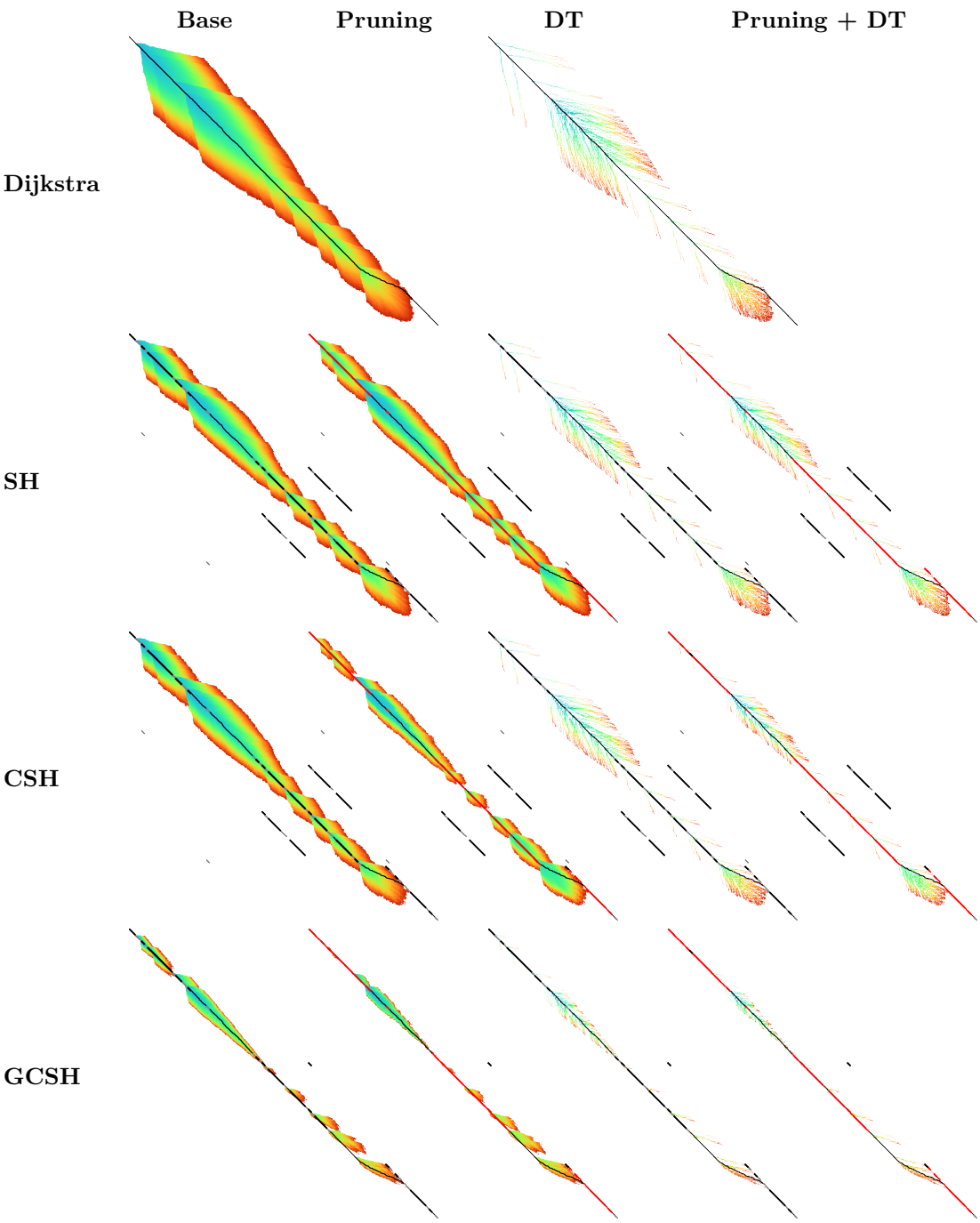


Figure 3.10 Expanded states for various heuristics and techniques, on a sequence containing a noisy region, a repeat, and an indel (n=1000, d=17.5%). The colour shows the order of expanding, from blue to red. The sequences include a highly divergent region, a repeat, and a gap. Matches are shown as **black 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.

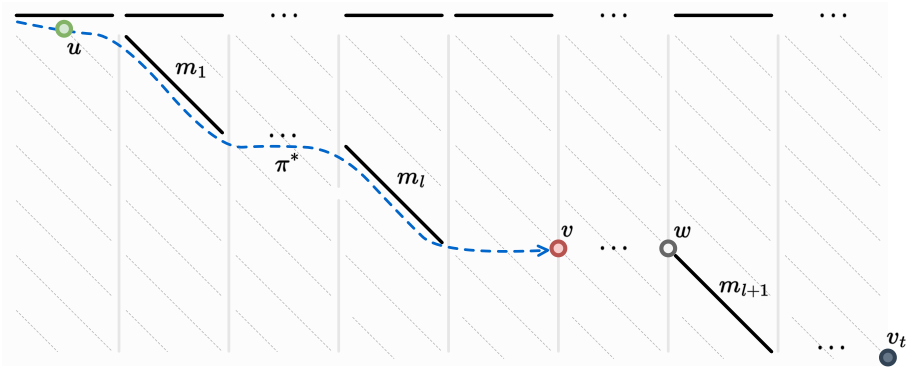


Figure 3.11 Variables of the proof of Lemma 2.

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 $h(v)$ (Definition 3.1) with $w := \text{start}(m_{l+1})$. This chain also minimizes $h(w)$ and thus $h(v) = \gamma(v,w) + h(w)$. We can now bound the cost of the joined chain from u to v and from w to the end and get our result via $\gamma(m_l, w) \leq \gamma(m_l, v) + \gamma(v, w)$ (Lemma 1)

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

◀

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

Proof. We will prove $h_\mathrm{s}^{\mathcal{M}}(u) \overset{(1)}{\leq} h_\mathrm{cs}^{\mathcal{M}}(u) \overset{(2)}{\leq} h_\mathrm{gcs}^{\mathcal{M}}(u) \overset{(3)}{\leq} h^*(u)$, 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 $h_\mathrm{s}^{\mathcal{M}} = h_{\preceq_i, c_\mathrm{seed}}^{\mathcal{M}} \leq h_{\preceq, c_\mathrm{seed}}^{\mathcal{M}} = h_\mathrm{cs}^{\mathcal{M}}$.

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

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

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

◀

3.A.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 2). Then, we show that our pruning heuristics are indeed weakly admissible (Theorem 3).

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

▶ **Definition 3.6** (Fixed vertex). *A vertex u is* fixed *if it is expanded and A* has found a shortest path to it, that is, $g(u) = g^*(u)$.*

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

▶ **Definition 3.7** (Weakly admissible). *A heuristic $\hat{h}$ is* weakly admissible *if at any moment during the A* search there exists a shortest path π^* from v_s to v_t in which all vertices $u \in \pi^*$ after its last fixed vertex n^* satisfy $\hat{h}(u) \le h^*(u)$.*

To prove that A* finds a shortest path when used with a weakly-admissible heuristic, we follow the structure of the original proof by Hart et al. [HNR68]. First we restate their Lemma 1 in our notation with a slightly stronger conclusion that follows directly from their proof.

▶ **Lemma 3** (Lemma 1 of [HNR68]). *For any unfixed vertex n and for any shortest path π^* from v_s to n, there exists an open vertex n' on π^* with $g(n') = g^*(n')$ such that π^* does not contain fixed vertices after n'.*

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

▶ **Corollary 1** (Generalization of Corollary to Lemma 1 of [HNR68]). *Suppose that $\hat{h}$ is weakly admissible, and that A* has not terminated. Then, there exists an open vertex n' on a shortest path from v_s to v_t with $f(n') \le g^*(v_t)$.*

Proof. Let π^* be the shortest path from v_s to v_t given by the weak admissibility of $\hat{h}$ (Definition 3.7). Since A* has not terminated, v_t is not fixed. Substitute $n = v_t$ in Lemma 3 to derive that there exists an open vertex n' on π^* with $g(n') = g^*(n')$. By definition of f we have $f(n') = g(n') + \hat{h}(n')$. Since π^* does not contain any fixed vertices after n', the weak admissibility of $\hat{h}$ implies $\hat{h}(n') \le h^*(n')$. Thus, $f(n') = g(n') + \hat{h}(n') \le g^*(n') + h^*(n') = g^*(v_t)$. ◀

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

Proof. The proof of Theorem 1 of [HNR68] applies, with the remark that instead of an arbitrary shortest path, we use the specific path π^* given by the weak admissibility and the specific vertex n' given by Corollary 1. ◀

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

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

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

Let n^* be the last fixed state on π^* before v, and let $u \in \pi^*$ be the successor of n^*. State u is open because its predecessor n^* is fixed and on a shortest path to u. Let the chain of all matches of cost less than r on π^* between u and v be $u \le m_1 \le \cdots \le m_l \le v$. Since n^* is the last fixed state

on π^*, none of these matches has been pruned, and they are all in $\mathcal{M}\backslash E$ as well. This means we can apply Lemma 2 to get $h(u) \le \mathrm{d}(u,v) + h(v)$, so

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

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

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

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

3.A.3 Computation of the (chaining) seed heuristic

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

Proof. For a chain of matches $\{m_i\} \subseteq \mathcal{M}$, let s_i and t_i be the start and end states of m_i. We translate the terms of our heuristic from costs to potentials and match scores (Section 3.6):

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

The heuristic (Definition 3.1) can then be rewritten as follows:

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

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

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

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

3.A.4 Computation of the gap-chaining seed heuristic

In this section we prove (Theorem 5) that we can change the dependency of GCSH on c_{gs} to c_{seed} by introducing a new partial order $\preceq_T$ on the matches. This way, Theorem 4 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 3.4** (Gap transformation). *The partial order $\preceq_T$ on states is induced by comparing both coordinates after the* gap transformation

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

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

▶ **Lemma 6.** *Let u and v be two states with v at the start of some seed. Then $u \preceq_T v$ if and only if $c_{\text{gap}}(u,v) \le c_{\text{seed}}(u,v)$. 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 7.** *When the set of matches $\mathcal{M}$ is consistent, $h^{\mathcal{M}}_{\text{gcs}}(u)$ can be computed using $\preceq_T$-chains only:*

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

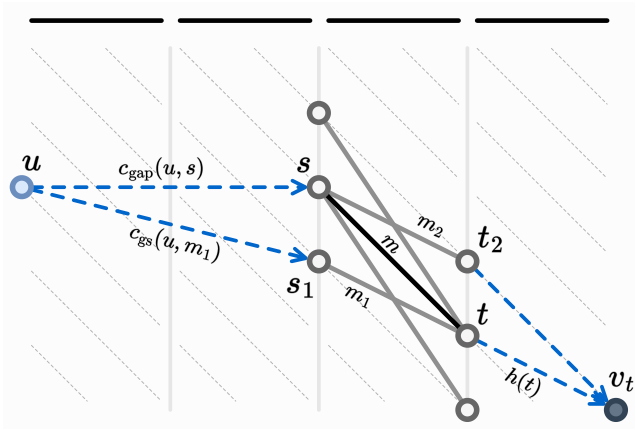


Figure 3.12 Variables in Case 2 of the proof of Lemma 7. Match m has score$(m) = 2$, so it has adjacent matches m_1 and m_2 (gray) with score$(m_i) \geq 1$.

Proof. We write $h := h^{\mathcal{M}}_{\text{gcs}} = h^{\mathcal{M}}_{\preceq,c_{\text{gs}}}$ (Definition 3.1) 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 Definition 3.1, so

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

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

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

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

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

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

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

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

- (priority 0) minimizes $h(u)$ in Equation (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 Equation (3.8).

Part 2.A: $u \preceq_T m$. Let s and t be the begin and end of m (Figure 3.12), and let m' be a match minimizing $h(t)$ in Equation (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 Equation (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 Equation (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 $\text{c}_{\text{m}}(m) = c_{\text{seed}}(s, t) - \text{score}(m)$, so the above simplifies to $c_{\text{gap}}(u, s) < c_{\text{seed}}(u, s) + \text{score}(m)$ and since these are integers $c_{\text{gap}}(u, s) \le c_{\text{seed}}(u, s) + \text{score}(m) - 1$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. Write $h := h^{\mathcal{M}}_{\text{gcs}}$ and $h' := h^{\mathcal{M}}_{\preceq_T, c_{\text{gs}}}$. When $u \not\preceq_T v_t$, $h(u) = c_{\text{gap}}(u, v_t)$ by Lemma 7. Otherwise, when $u \preceq_T v_t$, we have $h(u) = h'(u)$. Let $u \preceq_T m_1 \preceq_T \ldots \preceq_T v_t$ be a $\preceq_T$-chain for h' as in Definition 3.1. All terms in h' satisfy $\text{end}(m_i) \preceq_T \text{start}(m_{i+1})$, so $c_{\text{gap}} \leq c_{\text{seed}}$ and by Lemma 6 $c_{\text{gs}}(m_i, m_{i+1}) = c_{\text{seed}}(m_i, m_{i+1})$. Thus, $h' = h_{\preceq_T, c_{\text{seed}}}$, and $h^{\mathcal{M}}_{\text{gcs}}(u) = P(u) - S_T(u)$ by Theorem 4. ◀

4 A*PA2: Up to 19× Faster Exact Global Alignment

Summary

We now introduce A*PA2, an exact global pairwise aligner with respect to edit distance. The goal of A*PA2 is to unify the near-linear runtime of A*PA on similar sequences with the up to 500× higher efficiency of dynamic programming (DP) based methods. In practice, this means that A*PA2 combines a lot of of existing methods with a very optimized implementation, and then uses the seed heuristic on top for better scaling to long sequences.

Like Edlib, A*PA2 uses Ukkonen's band doubling in combination with Myers' bitpacking. In addition, A*PA2 1) uses large block sizes inspired by Block Aligner, 2) extends this with SIMD (single instruction, multiple data), 3) introduces a new *profile* for efficient computations, 4) introduces a new optimistic technique for traceback based on diagonal transition, 5) avoids recomputation of states where possible, and 6) applies the heuristics developed in A*PA and improves them using *pre-pruning*.

With just engineering optimizations, and no seed-heuristic, A*PA2-simple has complexity $O(ns)$ and is 6× to 8× faster than Edlib for sequences ≥ 10 kbp. A*PA2-full also includes the heuristic and is often near-linear in practice for sequences with small divergence. The average runtime of A*PA2 is 19× faster than the exact aligners BiWFA and Edlib on >500 kbp long ONT (Oxford Nanopore Technologies) reads of a human genome having 6% divergence on average.

Attribution

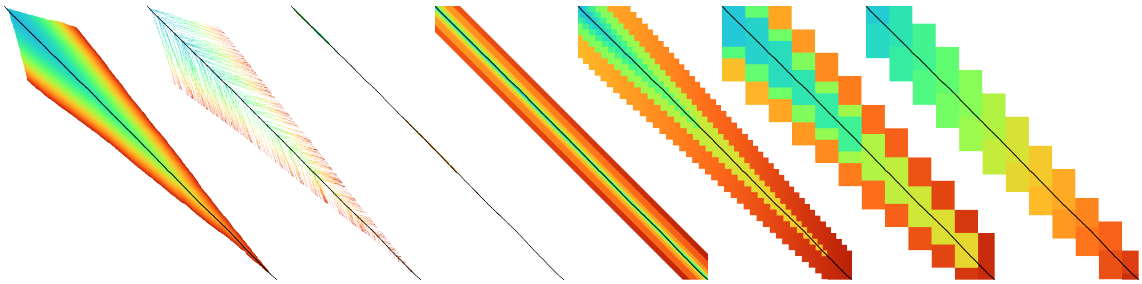
This chapter is based on the A*PA2 paper, "A*PA2: Up to 19× faster exact global alignment" [GK24], which is my own work. Large parts of this chapter are copied verbatim or with minor changes from that publication.

4.1 Introduction

As we saw, A*PA [GKI24] uses the A* shortest path algorithm to speed up alignment and has near-linear runtime when divergence is low. A drawback of A* is that it uses a queue and must store all computed distances. This causes large overhead compared to methods based on dynamic programming (DP), and means that A*PA can take up to 500× more time per visited state than methods based on DP.

In this chapter, we introduce A*PA2, a method that unifies the heuristics and near-linear runtime of A*PA with the efficiency of DP based methods.

As Fickett [Fic84, p. 1] stated 40 years ago and still true today,



(a) Dijkstra **(b)** WFA **(c)** A*PA **(d)** Ukkonen **(e)** Edlib **(f)** **A*PA2 -simple** **(g)** **A*PA2 -full**

Figure 4.1 Alignment of two sequences of length 3000 bp with 20% divergence using different methods. Coloured pixels correspond to visited states in the edit graph or dynamic programming matrix, and the blue to red gradient indicates the order of computation. The black path indicates an optimal alignment. (a) Dijkstra is the classical shortest path algorithm. (b) WFA uses the diagonal transition algorithm. (c) A*PA with the gap-chaining seed heuristic. (d) Ukkonen's method uses band doubling. (e) Edlib adds the gap heuristic and bitpacking. (f) A*PA2-simple additionally computes blocks of 256 columns at a time, and (g) A*PA2-full applies the heuristics of A*PA. Figure 4.12 shows the same methods on a more complicated alignment.

> at present one must choose between an algorithm which gives the best alignment but is expensive, and an algorithm which is fast but may not give the best alignment.

In this chapter we narrow this gap and show that A*PA2 is nearly as fast as approximate methods.

4.1.1 Contributions

We introduce A*PA2, an exact global pairwise sequence aligner with respect to edit distance.

In A*PA2, we combine multiple existing techniques and introduce a number of new ideas to obtain up to 19× speedup over existing single-threaded exact aligners. A*PA2 is often faster and never much slower than approximate methods.

As a starting point, we take the band doubling algorithm implemented by Edlib [ŠŠ17] using bitpacking [Mye99]. First, we speed up the implementation (points 1., 2., 3.). Then, we reduce the amount of work done (4., 5.). Lastly, we apply A* heuristics (6.).

1. **Block-based computation** Edlib (Figure 4.1e) computes one column of the DP matrix at a time, and for each column decides which range (subset of rows) of states to compute. We significantly reduce this overhead by processing blocks of 256 columns at a time (Figure 4.1f), taking inspiration from Block Aligner [LS23]. Correspondingly, we only store states of the DP-matrix at block boundaries, reducing memory usage.
2. **SIMD** We speed up the computation of each block by using 256bit SIMD, allowing the processing of 4 computer words in parallel.
3. **Novel encoding** We introduce a novel encoding of the input sequence to speed up SIMD operations by comparing characters bit-by-bit and avoiding slow `gather` instructions. This limits the current implementation to alphabets of size 4.
4. **Incremental doubling** Both the band doubling method of Ukkonen [Ukk85a] and Edlib recompute states after doubling the threshold. We avoid this by using the theory behind the A* algorithm, extending the incremental doubling of Fickett [Fic84] to blocks and arbitrary heuristics.

5. **Traceback** For the traceback, we optimistically use the diagonal transition method [Ukk85a, Mye86, MSMME21] within each block with a strong adaptive heuristic, only falling back to a full recomputation of the block when needed.
6. **A*** We apply the gap-chaining seed heuristic (GCSH) of A*PA [GKI24] (Figures 4.1c and 4.1g), and improve it using *pre-pruning*. This technique discards most of the *spurious* (off-path) matches ahead of time.

4.2 Methods

Conceptually, A*PA2 builds on Edlib. First we describe how we make the implementation more efficient using SIMD and blocks. Then, we modify the algorithm itself by using a new traceback method and avoiding unnecessary recomputation of states. On top of that, we apply the A*PA heuristics for further speed gains on large and complex alignments, at the cost of larger precomputation time to build the heuristic.

At the core, A*PA2 uses band doubling with the $g^*(u) + h(u) \leq t$ computational volume, in combination with bitpacking. That is, in each *iteration* of t we compute the distance to all states with $g^*(u) + h(u) \leq t$. In its simple form, A*PA2-simple, we use the gap heuristic, like Edlib does. The initial value for the tested distance t is the value of the heuristic at the start, and in the ith iteration $t_i := h(\langle 0, 0 \rangle) + B \cdot 2^i$, where B is the block size introduced below.

Section 4.2.1 to Section 4.2.6 describe our new methods, while Section 4.2.7 and Section 4.2.9 show the mathematical details of the algorithm.

4.2.1 Blocks

Instead of determining the range of rows to be computed for each column individually, we determine it once per *block* of $B = 256$ consecutive columns. This computes some extra states, but reduces the overhead by a lot. (From here on, B stands for the block size, and not for the sequence B to be aligned.) Within each block, we iterate over *lanes* of $w = 64$ rows at a time, and for each lane compute all B columns before moving on to the next lane.

Section 4.2.7 explains in detail how the range of rows to be computed is determined.

4.2.2 Memory

Where Edlib does not initially store intermediate values and uses meet-in-the-middle to find the alignment, A*PA2 stores the distance to all states at the end of *each* block, encoded as the distance to the top-right state of the block and the bit-encoded vertical differences along the right-most column. This simplifies the traceback method (see Section 4.2.5), and has sufficiently small memory usage to be practical.

4.2.3 SIMD

While it is tempting to use a SIMD vector as a single $W = 256$-bit word, the four $w = 64$-bit words (SIMD lanes) are dependent on each other and require manual work to shift bits between the lanes. Instead, we let each 256-bit AVX2 SIMD vector represent four 64-bit words (lanes) that are anti-diagonally staggered as in Figure 4.2a. This is similar to the original anti-diagonal tiling introduced by Wozniak [Woz97], but using units of w-bit words instead of single characters. This idea was already introduced in 2014 by the author of Edlib in a GitHub issue

(`github.com/Martinsos/edlib/issues/5`), but to our knowledge has never been implemented in either EDLIB or elsewhere.

We further improve instruction-level-parallelism (ILP) by processing 8 lanes at a time using two SIMD vectors in parallel, spanning a total of 512 rows (Figure 4.2a).

When the number of remaining lanes in a block to be computed is ℓ, we process 8 lanes in parallel as long as $\ell \geq 8$. If there are remaining lanes, we end with another 8-lane ($5 \leq \ell < 8$) or 4-lane ($1 \leq \ell \leq 4$) iteration that optionally includes some padding lanes at the bottom. In case the horizontal differences along the original bottom row are needed (as required by incremental doubling Section 4.2.9), we can not use padding and instead fall back to trying a 4-lane SIMD ($\ell \geq 4$), a 2-lane SIMD ($\ell \geq 2$), and lastly a scalar iteration ($\ell \geq 1$).

4.2.4 SIMD-friendly sequence profile

A drawback of anti-diagonal tiling is that each lane of a SIMD vector corresponds to a different column with character a_i that needs to be looked up in the profile $Eq[a_i][\ell]$. While SIMD can do multiple lookups in parallel using `gather` instructions, these instructions are not always efficient. Thus, we introduce the following alternative scheme. Let $b = \lceil \log_2(\sigma) \rceil$ be the number of bits needed to encode each character, with $b = 2$ for DNA. For each lane, the new profile Eq' stores b words as an $\lceil m/w \rceil \times b$ array $Eq'[\ell][p]$. Each word $0 \leq p < b$ stores the negation of the pth bit of each character in its lane. To check which characters in lane ℓ contain character c with bit representation $\overline{c_{b-1} \dots c_0}$, we precompute b words $C_0 = \overline{c_0 \dots c_0}$ to $C_{b-1} = \overline{c_{b-1} \dots c_{b-1}}$ and then compute $\bigwedge_{j=0}^{b-1}(C_j \oplus Eq'[\ell][j])$, where $\oplus$ denotes the xor operation. As an example take $b = 2$ and a lane with $w = 8$ characters $(0, 1, 2, 2, 3, 3, 3, 3)$. Then $Eq'[\ell][0] = \overline{00001101}$ (the negation of indicating odd positions) and $Eq'[\ell][1] = \overline{00000011}$, keeping in mind that bits are shown in reverse order in this notation. If the column now contains character $c = 2 = \overline{10}$ we initialize $C_0 = \overline{00000000}$ and $C_1 = \overline{11111111}$ and compute

$$(C_0 \oplus Eq'[\ell][0]) \wedge (C_1 \oplus Eq'[\ell][1]) = \overline{00001101} \wedge \overline{11111100} = \overline{00001100},$$

indicating that 0-based positions 2 and 3 contain character 2. This naturally extends to SIMD vectors, where each lane is initialized with its own constants.

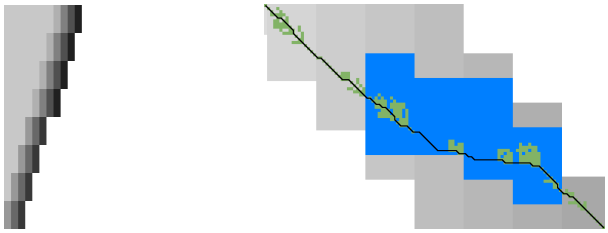
4.2.5 Traceback

The traceback stage takes as input the computed vertical differences at the end of each block of columns. We iteratively work backwards through the blocks. When tracing the block covering columns i to $i + B$, we know the distances $g(\langle i, j \rangle)$ to the states in column i at the start of the block, and a state $u = \langle i + B, j \rangle$ at distance $g^*(u)$ in column $i + B$ at the end of the block that is on an optimal path. The goal is to find an optimal path from column i to u.

A naive approach is to simply recompute the entire block of columns while storing distances to all states. Here we consider two more efficient methods.

Optimistic block computation Instead of computing the full range of rows for this column, a first insight is that only rows up to j are needed, since an optimal path to $u = \langle i + B, j \rangle$ can never go below row j.

Secondly, the path crosses $B = 256$ columns, and so we optimistically assume that it will be contained in rows $j - 256 - 64 = j - 320$ to j. Thus, we first recompute this range of rows (rounded out to multiples of $w = 64$) from left to right while storing intermediate values, as shown in blue in



(a) SIMD **(b)** Traceback

Figure 4.2 (a) SIMD processing of two times 4 lanes in parallel. This example uses lanes of 4 instead of 64 rows. First the top-left triangle is computed lane by lane, and then 8-lane diagonals are computed by using two 4-lane SIMD vectors in parallel. (b) Computed blocks are shown in grey. States expanded by the diagonal transition traceback in each block are shown in green. When the distance in a block is too large, a part of the block is fully recomputed as fallback, as shown in blue. In regions with low divergence, diagonal transition is sufficient to trace the path, and only in noisy regions the algorithm falls back to recomputing full blocks.

Figure 4.2b. If the distance to u computed this way equals $g^*(u)$, there is a shortest path contained within the computed rows and we trace it one state at a time. Otherwise, we repeatedly try again with double the number of lanes until success. The exponential search ensures low overhead and good average case performance.

Optimistic diagonal transition traceback (DTT) A second improvement uses the *diagonal transition* algorithm backwards from u. We simply run the unmodified algorithm on the reverse graph covering columns i to $i + B$ and rows 0 to j. Whenever a state v in column i is reached, say at distance d from u, we check whether $g(v) + d = g^*(u)$, and continue until a v is found for which this holds. We then know that v lies on a shortest path and we can find the path from v to u via a usual traceback on the diagonal transition algorithm, as shown in green in Figure 4.2b.

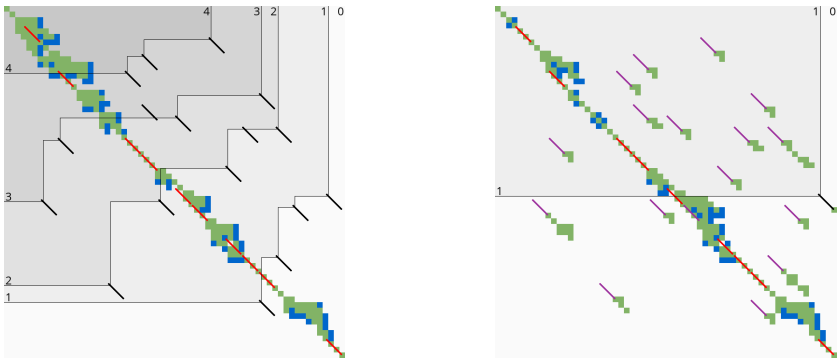
As a further optimization, when no suitable v is found after trying all states at distance $\leq g = 40$, we abort the DTT and fall back to the block doubling described above. Another optimization is the WF-adaptive heuristic introduced by WFA [MSMME21]: all states that lag more than $x = 10$ behind the furthest reaching diagonal are dropped. Lastly, we abort early when it takes cost $> g/2$ to cross half the columns. Both g and x are experimentally determined, see Figure 4.11.

4.2.6 A* and pruning

Edlib already uses a simple *gap cost* heuristic that gives a lower bound on the number of insertions and deletions on a path from each state to the end. We replace this by the much stronger gap-chaining seed heuristic (GCSH) introduced in A*PA, with two modifications.

Bulk pruning In A*PA, matches are *pruned* as soon as a shortest path to their start has been found. This helps to penalize states *before* (left of) the match. Each iteration of A*PA2 works left-to-right only, so that pruning of matches does not affect the current iteration. Thus, we collect all matches to be pruned at the end of each iteration, and update the contours in one right-to-left sweep. To keep the computational volume valid after pruning, we ensure that the range of computed rows in each column never shrinks.

Pre-pruning Here we introduce an independent optimization that also applies to the original A*PA method. Each of the heuristics h introduced in A*PA [GKI24] depends on the set of *matches* $\mathcal{M}$ between seeds in A and k-mers of B.



(a) Without pre-pruning **(b)** With pre-pruning

Figure 4.3 Effect of pre-pruning on chaining seed heuristic (CSH) contours. The left shows contours and layers of the heuristic at the end of an A*PA alignment, after matches (black diagonals) on the path have been pruned (red diagonals). The right shows pre-pruned matches in purple and the states visited during pre-pruning in green. After pre-pruning, almost no off-path matches remain. This decreases the number of contours, making the heuristic stronger, and simplifies contours, making the heuristic faster to evaluate.

Now consider a seed s_i with an exact match m. The existence of the match is a 'promise' that seed s_i can be crossed for free. (Seeds without match require at least 1 edit.) When m can not be extended into an alignment of s_i and s_{i+1} of cost less than 2, we can amortize this cost over the two seeds and regard m as a 'false promise', since crossing the two seeds takes cost at least 2. Thus, we remove m, making the heuristic more accurate.

More generally, we try to extend each match m into an alignment of seeds s_i up to the start of s_{i+q} for all $q \le p = 14$. If all extensions have cost $\ge q$, then m falsely promised that s_i to s_{i+q} can be crossed for cost $< q$ and we *pre-prune* (remove) m.

The extension of each match is done by running the diagonal transition algorithm from its end. Any furthest reaching states that are at distance $\ge q$ while at most q seeds have been covered are dropped, and the match is pre-pruned when no active states are left.

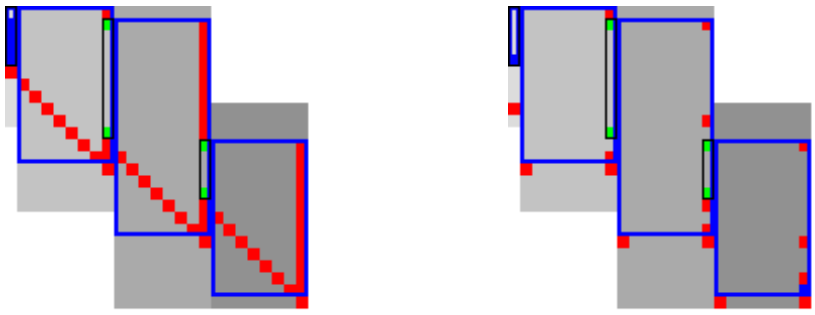
As shown in Figure 4.3b, the effect is that the number of off-path matches is significantly reduced. This makes contours simpler and hence faster to initialize, update, and query, and it increases the value of the heuristic.

Pre-pruning lowers number of remaining matches and allows using $k = 12$ instead of $k = 15$, further improving the heuristic. For $k = 12$, $p = 14$ was experimentally determined to remove nearly all off-path matches, while not taking significant time (Figure 4.11).

4.2.7 Determining the rows to compute

For each block, from column i to $i + B$, we only compute those rows that can possibly contain states on a path/alignment of cost at most t. Intuitively, we try to 'trap' the alignment inside a wall of states that can not lie on a path of length at most t, i.e., that must have $f^*(u) > t$, as can be seen in red in Figure 4.4a. We determine this range of rows in two steps.

Step 1: Fixed range First, we determine the *fixed range* at the end of the preceding block. I.e., we find the topmost and bottommost states $\langle i, j_{start}\rangle$ and $\langle i, j_{end}\rangle$ with $f(u) = g(u) + h(u) \le t$, as shown in green in Figure 4.4. All in-between states $u = \langle i, j\rangle$ with $j_{start} \le j \le j_{end}$ are then *fixed*, meaning that the correct distance has been found and $g(u) = g^*(u)$. One way to find j_{start} and



(a) Simple **(b)** Sparse

Figure 4.4 Detail of computed ranges (a) Starting with a fixed range between two green states (black rectangle), first, the bottom of the to be computed region (blue rectangle) is computed (red diagonal, with $f_l(u) > t$). Then, the region is rounded out to multiples of w and computed (grey background). Lastly, the last column is shrunk (red, $f(u) > t$) until states with $f(u) \leq t$ are found (green), that determine the fixed range (black rectangle). The last block has no fixed states in its right column, so t must be increased. (b) The sparse variant uses much fewer heuristic invocations.

j_{end} is by simply iterating inward from the start/end of the computed range and skipping all states with $f(u) = g(u) + h(u) > t$, as indicated by the red columns in Figure 4.4a.

Step 2: End of computed range Then, we find the bottommost state $v = \langle i + B, j'_{end} \rangle$ at the end of the to-be-computed block that can possibly lie on a path of length $\leq t$. We then compute rows j_{start} to j'_{end} in columns i to $i + B$, rounding $[j_{start}, j'_{end}]$ *out* to the previous/next multiple of the word size $w = 64$.

To determine j'_{end}, let $u = \langle i, j_{end} \rangle$ be the bottommost fixed state in column i with $f(u) \leq t$. Let $v = \langle i', j' \rangle$ be a state in the current block ($i \leq i' \leq i + B$) that is strictly below the diagonal of u. Suppose v lies on a path of length $\leq t$. This path must cross column i in or above u, since states u' below u have $f^*(u') > t$. The distance to v is now at least $\min_{j \leq j_{end}} \left(g^*(\langle i, j \rangle) + c_{\text{gap}}(\langle i, j \rangle, v) \right) \geq g^*(u) + c_{\text{gap}}(u, v)$, and thus we define

$$f_l(v) := g^*(u) + c_{\text{gap}}(u, v) + h(v) \leq f^*(v)$$

as a lower bound on the length of the shortest path through v, assuming v is strictly below the diagonal of u and $f^*(v) \leq t$. When $f_l(v) > t$, this implies $f^*(v) > t$ and also $f^*(v') > t$ for all v' below v. The end of the range is now computed by finding the bottommost state v in each column for which f_l is at most t, using the following algorithm:

1. Start with $v = \langle i', j' \rangle = u = \langle i, j_{end} \rangle$.
2. While the below-neighbour $v' = \langle i', j' + 1 \rangle$ of v has $f_l(v) \leq t$, increment j'.
3. Go to the next column by incrementing i' and j' by 1 and repeat step 2, until $i' = i + B$.

The row j'_{end} of the last v we find in this way is the bottommost state in column $i + B$ that can possibly have $f(v) \leq t$, and hence this is end of the range we compute.

In Figure 4.4a, we see that $f(v)$ is evaluated at a diagonal of states just below the bottommost fixed (green) state u at the end of the preceding black, and that the to-be-computed range (indicated in blue) includes exactly all states above this diagonal.

4.2.8 Sparse Heuristic Invocation

A drawback of the previous method is that it requires a large number of calls to f and hence to the heuristic h: roughly one per column and one per row. Here we present a *sparse* version that uses

fewer calls to f than the method of Section 4.2.7, as shown in Figure 4.4b.

We first state two very similar lemmas, and then give the updated steps.

▶ **Lemma 8.** *When h is admissible and $f(u) > t + 2D$, then $f^*(u') > t$ for all u' within distance $d(u, u') \leq D$ from u.*

Proof. Since adjacent states differ in distance by $\{-1, 0, +1\}$, we have $g(u') \geq g(u) - d(u, u') \geq g(u) - D$ and $h^*(u') \geq h^*(u) - d(u, u') \geq h^*(u) - D$. Now suppose that $f^*(u') \leq t$. Then u' is fixed and we have $g(u') = g^*(u')$, and since h is admissible $h(u) \leq h^*(u)$. Thus:

$$\begin{aligned} t + 2D < f(u) &= g(u) + h(u) \\ &\leq g(u) + h^*(u) \leq g(u') + h^*(u') + 2D \\ &= g^*(u') + h^*(u') + 2D = f^*(u') + 2D \leq t + 2D. \end{aligned}$$

This is a contradiction, so we must have $f^*(u') > t$, as required. ◀

▶ **Lemma 9.** *When h is admissible, v is below the diagonal of a computed state u, and $f_l(v) = g^*(u) + c_{\text{gap}}(u, v) + h(v) > t + 2D$, then $f^*(v') > t$ when v has distance $d(v, v') \leq D$ from u.*

Proof. We have $c_{\text{gap}}(u, v') \geq c_{\text{gap}}(u, v) - d(v, v') \geq c_{\text{gap}}(u, v) - D$, and $h^*(v') \geq h^*(v) - D$. Now suppose that $f^*(v') \leq t$. Then we know that $g^*(v) \geq g^*(u) + c_{\text{gap}}(u, v)$, and we still have $h(v) \leq h^*(v)$, $g^*(v') \geq g^*(v) - D$, and $h^*(v') \geq h^*(v) - D$. It follows that

$$\begin{aligned} t + 2D < f_l(v) &= g^*(u) + c_{\text{gap}}(u, v) + h(v) \\ &\leq g^*(v) + h^*(v) \\ &\leq g^*(v') + h^*(v') + 2D = f^*(v') \leq t + 2D, \end{aligned}$$

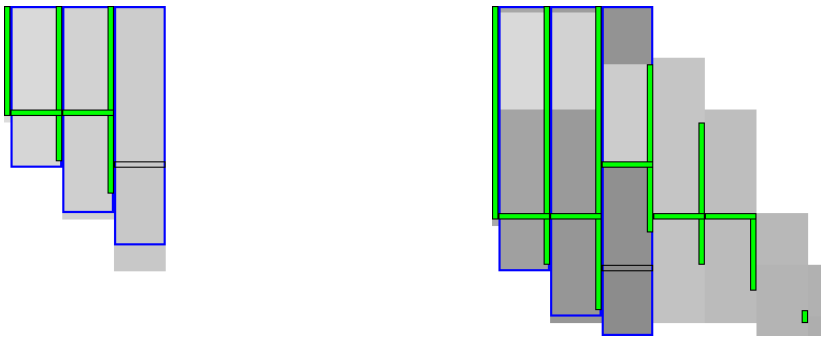
which is a contradiction, so we conclude that $f^*(v') > t$, as required. ◀

Step 1': Sparse fixed range To find the first row j_{start} with $f(\langle i, j_{start}\rangle) \leq t$, start with $j = r_{start}$, and increment j by $\lceil (f(v) - t)/2 \rceil$ as long as $f(v) > t$, since none of the intermediate states can lie on a path of length $\leq t$ by Lemma 8. The last row is found in the same way by going up from r_{end}. As seen in Figure 4.4b, this sparse variant significantly reduces the number of evaluations of the heuristic in the right-most columns of each block.

Step 2': Sparse end of computed range Instead of considering one column at a time, we now first make a big jump down and then jump to the right.

1. Start with $v = \langle i', j' \rangle = u + \langle 1, B + 1 \rangle = \langle i + 1, j_{end} + B + 1 \rangle$.
2. If $f_l(v) \leq t$, increase j' (go down) by 8.
3. If $f_l(v) > t$, increase i' (go right) by $\lceil (f_l(v) - t)/2 \rceil$, but do not exceed column $i + B$.
4. Repeat from step 2, until $i' = i + B$.
5. While $f_l(v) > t$, decrease j' (go up) by $\lceil (f_l(v) - t)/2 \rceil$, but do not go above the diagonal of u.

The resulting v is again the bottommost state in column $i + B$ that can potentially have $f(t) \leq t$, and its row is the last row that will be computed.



(a) First iteration **(b)** Second iteration

Figure 4.5 Incremental doubling detail Blue rectangles show the ranges required to be computed, and grey the computed blocks. Vertical green rectangles show the fixed range at the end of each block, and green horizontal rectangles a fixed row j_f of states inside some blocks. In both figures the third block was just computed, in the (a) first and (b) second iteration. The black empty rectangle indicates the new candidate j'_f for the fixed horizontal region. In (a), the computation is split into two parts, above and below j'_f. In (b), the computation is split into three parts (dark grey): above the reusable region, between the old j_f and the new j'_f, and below the new j'_f.

4.2.9 Incremental doubling

When band doubling doubles the threshold from t to $2t$, it simply recomputes the distance to all states. On the other hand, Dijkstra visits states in increasing order of distance, and the distance to a state is correct (*fixed*) as soon as a state is expanded.

Indeed, band doubling algorithm can also avoid recomputations. After completing the iteration for t, it is guaranteed that the distance is fixed to all states that satisfy $f(u) \leq t$. In fact, a stronger result holds: in any column, the distance is fixed for all states *between* the topmost and bottommost state in that column with $f(u) \leq t$.

In each block, we would like to skip some rows preceding j_{end}, the end of the fixed range in its first column. To be able to do this, we must store the horizontal differences along row j_{end} so that we can continue from there in the next iteration. In practice, we choose row j_f (for *fixed*) as the last row at a lane boundary before j_{end}, as indicated in Figure 4.5 by a horizontal black rectangle. In the first iteration (Figure 4.5a), reusing values is not yet possible, so the computation of the block is split into two parts: one above j_f, to extract the horizontal differences along row j_f, and the remainder below j_f.

In the second and further iterations (Figure 4.5b), the values at j_f are reused and each block is split into three parts. The first part computes all lanes covering states before the start of the fixed range (green column) at the end of the block. We then skip the lanes up to the previous j_f, since the values at both the bottom and right of this region are already fixed. Then, we compute the lanes between the old j_f and its new value j'_f. Lastly we compute the lanes from j'_f to the end.

4.3 Results

A*PA2 is available at `github.com/RagnarGrootKoerkamp/astar-pairwise-aligner` and written in Rust. We compare it against other aligners on real datasets, report the impact of the individual techniques we introduced, and measure time and memory usage.

Dataset	Source	Cnt	Length [kbp]			Divergence [%]			Max gap [kbp]	
			min	mean	max	min	mean	max	mean	max
SARS-CoV-2	A*PA2	10,000	27	30	30	0	**1.5**	13	0	1
ONT 1 kbp	WFA	12,500	0	1	1	0	10	23	0	0
ONT 10 kbp	BiWFA	10,000	0	11	50	3	12	19	0	3
ONT >500 kbp	A*PA	50	500	594	849	3	6	17	0	1
ONT >500 kbp + gv	BiWFA	48	502	632	1,053	4	7	18	**1.9**	42

Table 4.1 Dataset statistics All but the first dataset are ONT reads. The dataset with genetic variation (gv) also includes long gaps, while the SARS-CoV-2 dataset stands out for having only 1.5% divergence on average. **Cnt**: number of sequence pairs. **Max gap**: longest gap in the reconstructed alignment.

4.3.1 Setup

Datasets We benchmark on five datasets containing real sequences of varying length and divergence, as listed in detail in Table 4.1. They can be downloaded from `github.com/pairwise-alignment/pa-bench/releases/tag/datasets`.

Four datasets containing Oxford Nanopore Technologies (ONT) reads are reused from the WFA, BiWFA, and A*PA evaluations [MSMME21, MSEG+23, GKI24]. Of these, two '>500 kbp' and '>500 kbp with genetic variation' datasets have divergence (error rate, or more precisely, edit distance divided by length) $6-7\%$, while two '1 kbp' and '10 kbp' datasets are filtered for sequences of length <1 kbp and <50 kbp and have average divergence 11% and average sequence length 800 bp and 11 kbp.

A SARS-CoV-2 dataset was newly generated by downloading 500 MB of viral sequences from the COVID-19 Data Portal, `covid19dataportal.org` [HLR+21], filtering out non-ACTG characters, and selecting 10000 random pairs. This dataset has average divergence 1.5% and average length 30 kbp.

For each set, we sorted all sequence pairs by edit distance and split them into 50 files each containing multiple pairs, so that the first file contains the 2% of pairs with the lowest divergence. Reported runtimes are averaged over the sequences in each file.

Algorithms and aligners We benchmark A*PA2 against state-of-the-art exact aligners Edlib, BiWFA, and A*PA. We further compare against the approximate aligners WFA-Adaptive [MSMME21] and Block Aligner. For WFA-Adaptive we use default parameters $(10, 50, 10)$, dropping states that lag behind by more than 50. For Block Aligner we use block sizes from 0.1% to 1% of the input size. Block Aligner only supports affine costs so we use gap-open cost 1 instead of 0.

We compare two versions of A*PA2. *A*PA2-simple* uses all engineering optimizations (bitpacking, SIMD, blocks, new traceback) and uses the simple gap-heuristic. *A*PA2-full* additionally uses more complicated techniques: incremental-doubling, and the gap-chaining seed heuristic introduced by A*PA with pre-pruning.

Parameters For A*PA2, we fix block size $B = 256$. For A*PA2-full, we use the gap-chaining seed heuristic (GCSH) of A*PA with exact matches and seed length $k = 12$. We pre-prune matches by looking ahead up to $p = 14$ seeds. Parameters were determined experimentally, see Figure 4.11 for a comparison. For most parameters, the runtime is not very sensitive to the exact value. For A*PA, we use the original inexact matches with seed length $k = 15$ by default, and only change this for the low-divergence SARS-CoV-2 dataset and 4% divergence synthetic data, where we use exact matches ($r = 1$).

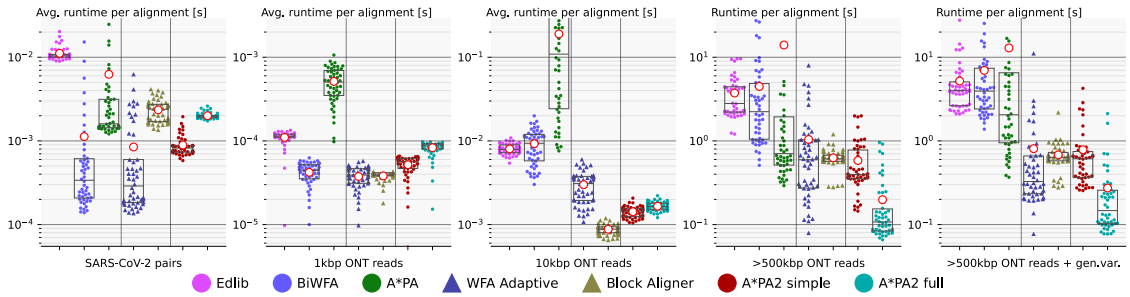


Figure 4.6 Runtime comparison (log) Each dot shows the running time of a single alignment (right two plots) or the average runtime over 2% of the input pairs (left three plots). Box plots show the three quartiles, and the red circled dot shows the average running time over all alignments. For A*PA, exact matches are used for the SARS-CoV-2 dataset. Some A*PA alignments ≥ 10 kbp time out, and the shown average is a lower bound on the true average. Approximate aligners WFA Adaptive and Block Aligner are indicated with triangles. On the >500 kbp reads, A*PA2-full is 19× faster than other exact methods.

	SARS-CoV-2	ONT			
Aligner	[ms]	1 kbp [ms]	10 kbp [ms]	>500 kbp [s]	>500 kbp + gv [s]
Edlib	11.14	0.110	8.0	3.74	5.20
BiWFA	1.13	0.042	9.3	4.47	6.96
A*PA	6.25	0.514	>190.1	>14.01	>12.92
WFA-Adaptive	0.85	0.038	3.0	1.04	0.81
Block Aligner	2.35	0.038	0.9	0.63	0.68
A*PA2-simple	0.89	0.052	1.4	0.58	0.78
A*PA2-full	2.00	0.083	1.7	0.20	0.27
Speedup [×]	1.3	0.81	5.6	18.8	19.0

Table 4.2 Average runtime per sequence of each aligner on each dataset. Cells marked with > are a lower bound due to timeouts. Speedup is reported as the fastest A*PA2 variant compared to the fastest of Edlib, BiWFA, and A*PA. WFA-Adaptive and Block Aligner are approximate aligners.

Execution We ran all benchmarks using PaBench (`github.com/pairwise-alignment/pa-bench`) on Arch Linux on an `Intel Core i7-10750H` with 64GB of memory and 6 cores, with hyper-threading disabled, frequency boost disabled, and CPU power saving features disabled. The CPU frequency is fixed to 3.6GHz and we run 1 single-threaded job at a time with niceness −20. Reported running times are the average wall-clock time per alignment and exclude the time to read data from disk for more stable measurements. For A*PA and A*PA2-full, reported times do include the time to find matches and initialize the heuristic.

4.3.2 Comparison with other aligners

Speedup on real data Figure 4.6 compares the running time of aligners on real datasets. Table 4.2 contains a corresponding table. On the >500 kbp ONT reads, A*PA2-full is 19× faster than Edlib, BiWFA, and A*PA in average running time, and using the gap-chaining seed heuristic in A*PA2-full provides 3× speedup over A*PA2-simple.

On shorter sequences, the overhead of initializing the heuristic in A*PA2-full is large, and

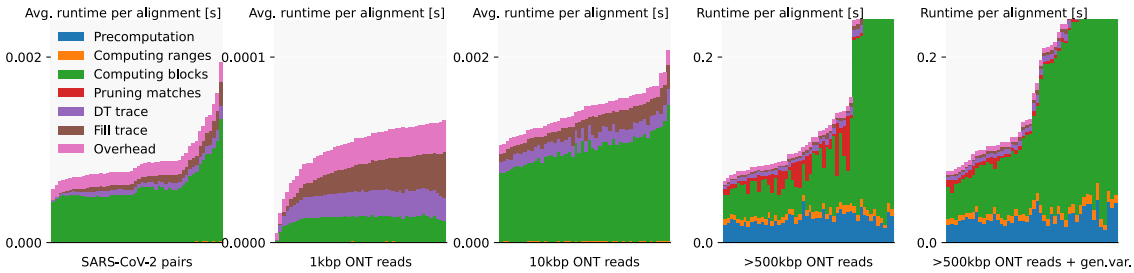


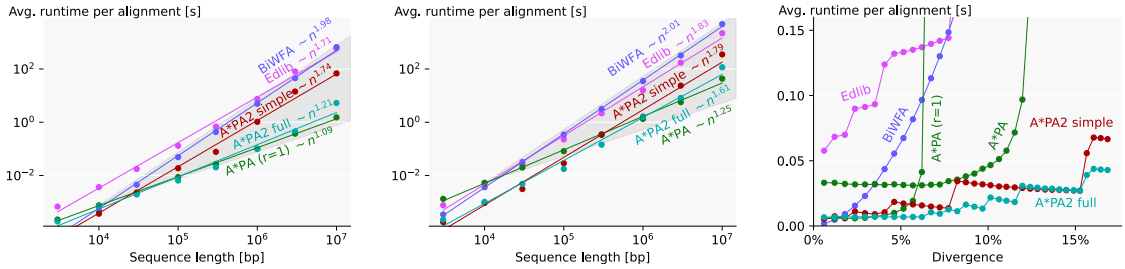
Figure 4.7 Runtime profile of A*PA2 using A*PA2-simple for short sequences and A*PA2-full for the two rightmost >500 kbp datasets. Each column corresponds to a (set of) alignment(s), which are sorted by total runtime. *Overhead* is the part of the runtime not measured in one of the other parts and includes the time to build the profile. For >500 kbp long sequences, A*PA2-full spends most of its time computing blocks, followed by the initialization of the heuristic. For very short sequences of 1 kbp, up to half the time is spent on tracing an optimal alignment.

	SARS-CoV-2	ONT			
Aligner		1 kbp	10 kbp	>500 kbp	>500 kbp + gv
WFA-Adaptive	92%	93%	49%	60%	4%
Block Aligner	34%	85%	53%	96%	50%

Table 4.3 Percentage of correctly aligned reads by approximate aligners. The accuracy of WFA-Adaptive drops a lot for the >500 kbp dataset with genetic variation, since these alignments contain gaps of thousands of basepairs, much larger than the 50 bp cutoff after which trailing diagonals are dropped.

Memory [MB]	SARS-CoV-2		ONT							
			1 kbp		10 kbp		>500 kbp		>500 kbp + gv	
Aligner	median	max	median	max	median	max	median	max	median	max
Edlib	0	0	0	0	0	0	0	0	0	0
BiWFA	0	0	0	0	0	0	4	11	0	2
A*PA	0	236	0	0	228	873	84	3,453	158	6,868
WFA-Adaptive	0	11	0	0	0	0	0	0	0	0
Block Aligner	0	16	0	0	0	3	583	1,189	610	2,171
A*PA2 simple	2	5	0	0	4	6	0	55	2	164
A*PA2 full	0	0	0	0	0	0	30	82	6	141

Table 4.4 Memory usage of aligners, measured as the increase in `max_rss` before and after aligning a pair of sequences.



(a) Scaling with length ($d = 4\%$) **(b)** Scaling with length ($d = 12\%$) **(c)** Scaling with divergence

Figure 4.8 Runtime comparison on synthetic data (a)(b) Log-log plot of average running time of aligners on synthetic sequences of increasing length with 4% divergence and 12% divergence. A*PA uses exact matches (indicated by $r = 1$) for $d = 4\%$ and inexact matches for $d = 12\%$. For sequences of length n, averages are over $10^7/n$ pairs. Lines are fitted in the log-log domain. The region between linear and quadratic growth is shaded in grey. (c) Average running time of aligners over 10 sequences of length 100 kbp with varying uniform divergence.

A*PA2-simple is faster. For the 10 kbp dataset, A*PA2-simple is 5.6× faster than other exact methods. For the shortest (1 kbp ONT reads) and most similar sequences (SARS-CoV-2 with 1% divergence), BiWFA is usually faster than Edlib and A*PA2-simple. In these cases, the overhead of using 256 wide blocks is relatively large compared to the edit distance $s \leq 500$ in combination with BiWFA's $O(s^2 + n)$ expected running time.

Comparison with approximate aligners For the smallest datasets, BiWFA is about as fast as the approximate methods WFA Adaptive and Block Aligner, while for the largest datasets A*PA2-full is significantly faster. Only on the dataset of 10 kbp ONT reads is Block Aligner 1.6× faster than A*PA2, but it only finds the correct edit distance for 53% of the alignments. All accuracy numbers can be found in Table 4.3.

Scaling with length Figures 4.8a and 4.8b compare the runtime of aligners on synthetic random sequences of increasing length and constant uniform divergence. BiWFA's runtime is quadratic and is fast for sequences up to 3000 bp. As expected, A*PA2-simple has very similar scaling to Edlib but is faster by a constant factor around 7.5×. A*PA2-full includes the gap-chaining seed heuristic used by A*PA, resulting in comparable speed and near-linear scaling for both of them when $d = 4\%$. For more divergent sequences, A*PA2-full is faster than A*PA since initializing the A*PA heuristic with inexact matches is relatively slow. The reason that A*PA2-full is slower than A*PA for sequences of length 10 Mbp is that A*PA2-full uses shorter seed length $k = 12$ instead of $k = 15$. In most cases, pre-pruning is fast enough to handle the extra matches this causes, but when n approaches $4^{12} \approx 16 \cdot 10^6$, this becomes a bottleneck.

Scaling with divergence Figure 4.8c compares the runtime of aligners on synthetic sequences of increasing divergence. BiWFA's runtime grows quadratically, while Edlib grows linearly and jumps up each time another doubling of the threshold is required. A*PA is fast until the maximum potential is reached at 6% resp. 12% and then becomes very slow. A*PA2-simple behaves similar to Edlib and jumps up each time another doubling of the threshold is needed, but is around 8× faster. A*PA2-full outperforms BiWFA for divergence ≥ 2% and A*PA for divergence ≥ 4%. The runtime of A*PA2-full is near-constant up to divergence 7% due to the gap-chaining seed heuristic

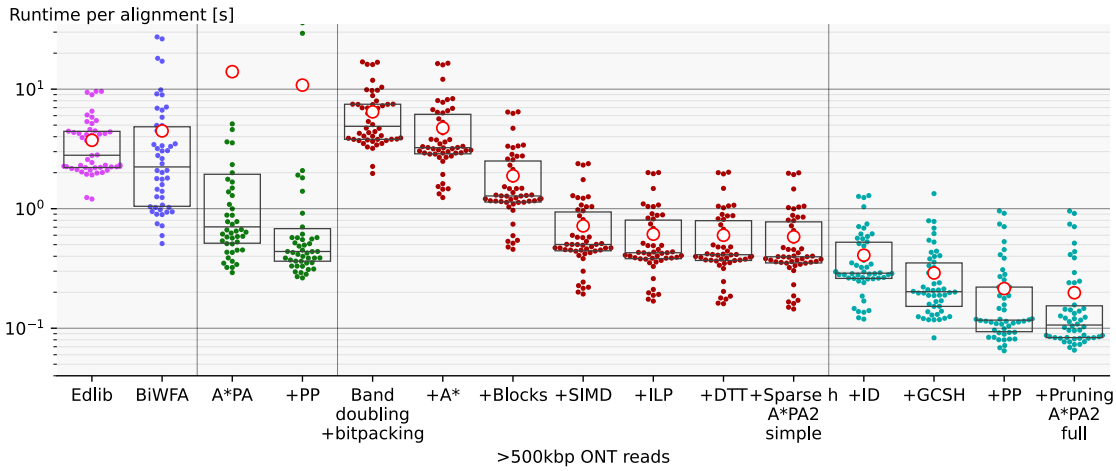


Figure 4.9 Effect of adding features Box plots showing the performance improvements of A*PA2 on long sequences when incrementally adding new methods one-by-one. A*PA2-simple corresponds to the middle red columns, and A*PA2-full corresponds to the rightmost blue columns.

which can correct for up to $1/k = 1/12 = 8.3\%$ of divergence, while A*PA2-simple starts to slow down at lower divergence. The graph of A*PA2 has a negative slope when the number of doublings is fixed, because too low thresholds are rejected more quickly when divergence is higher.

Memory usage of A*PA2 on >500 kbp sequences is at most 200 MB and only 30 MB in median, down from 6868 MB and 158 MB for A*PA. On other datasets and for Edlib and BiWFA, memory usage is at most 11 MB, and usually < 1 MB (Table 4.4).

4.3.3 Effects of methods

Figure 4.9 shows the effect of one-by-one adding improvements to A*PA2 on >500 kbp long sequences, roughly in order of importance, starting with Ukkonen's band doubling method using Myers' bitpacking. Figure 4.10 instead shows the effect of removing each improvement from the final method. We first change to the $g^*(u) + c_{\text{gap}}(u, v_t)$ computational volume (+A*), making it comparable to Edlib. Then we process blocks of 256 columns at a time and only store differences at block boundaries, giving 2.5× speedup. Adding SIMD gives another 2.7× speedup, and instruction level parallelism (ILP) provides a further small improvement. The diagonal transition traceback (DTT) and sparse heuristic computation do not improve performance of A*PA2-simple much on long sequences, but their removal can be seen to slow it down for shorter sequences in the ablation (Figure 4.10). Incremental doubling (ID), the gap-chaining seed heuristic (GCSH), pre-pruning (PP), and the pruning of A*PA give another 3× speedup on average and 4× speedup in the first quantile.

Figure 4.7 shows that A*PA2 typically spends most of its time computing blocks. For short 1 kbp long sequences, half the time is spent on traceback, and for the >500 kbp sequences, A*PA2-full spends around a quarter of time on initializing the heuristic.

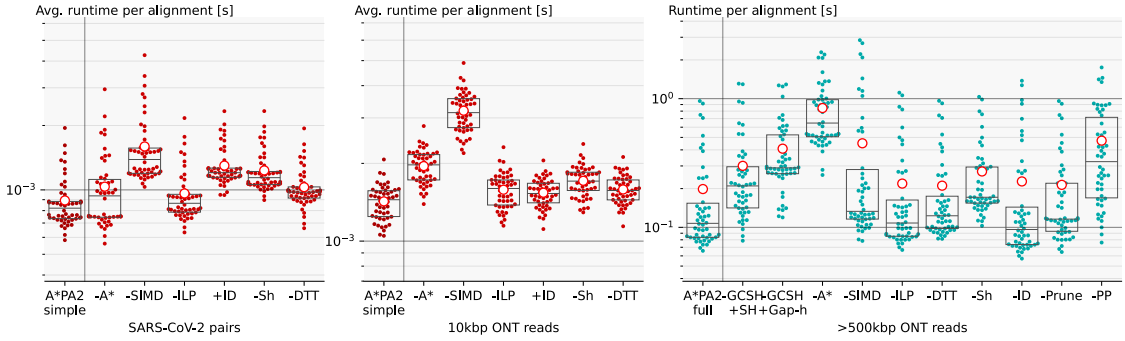


Figure 4.10 Ablation Box plots showing how the performance of A*PA2-simple (left, middle) and A*PA2-full (right) changes when removing (-) or adding (+) features. ILP: instruction level parallelims, ID: incremental doubling, Sh: sparse heuristic evaluation, DTT: diagonal transition traceback, PP: pre-pruning. Note that adding incremental doubling to A*PA2-simple slows down simple alignments, while A*PA2-full benefits from it for larger alignments.

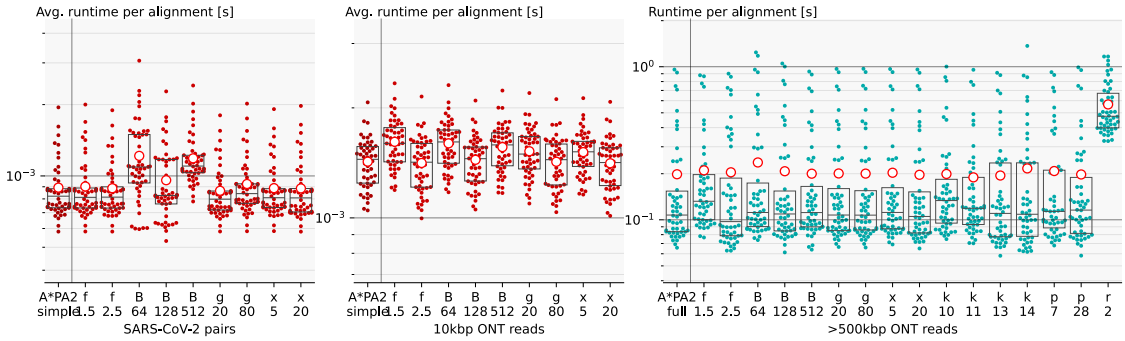


Figure 4.11 Changing parameters Runtime of A*PA2-simple (left, middle) and A*PA2-full (right) with one parameter modified. Default parameters are seed length $k = 12$, pre-pruning look-ahead $p = 14$, growth factor $f = 2$, block size $B = 256$, max DTT traceback cost $g = 40$, and dropping diagonals that lag $x = 10$ behind during traceback. Running time is not very sensitive with regards to most parameters. Of note are using inexact matches ($r = 2$) for the heuristic, which take significantly longer to find, larger seed length k, which decreases the strength of the heuristic, and smaller block sizes ($B = 128$ and $B = 64$), which induce more overhead.

4.4 Discussion

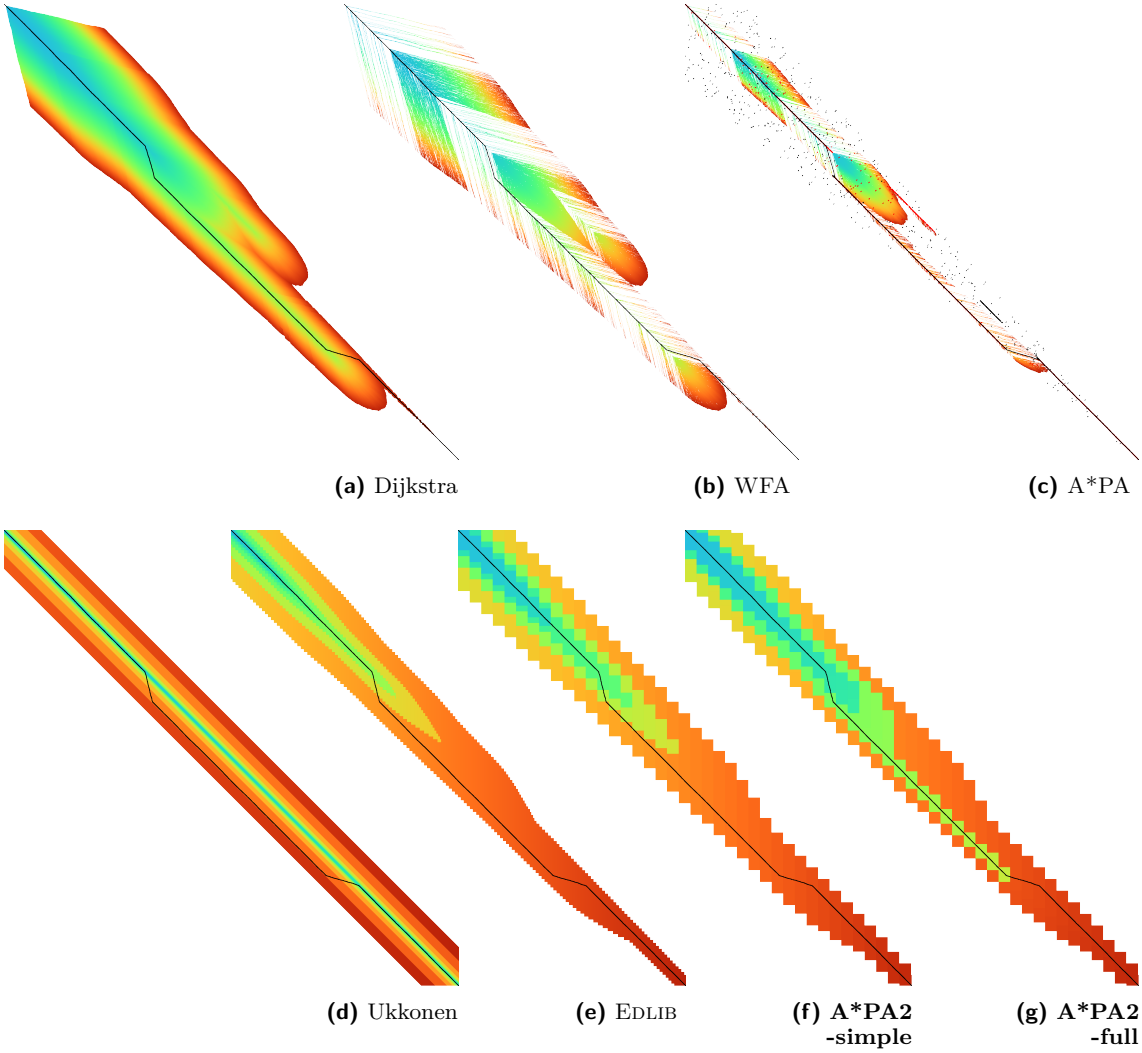
We have shown that by incorporating many existing techniques and by writing highly optimized code, A*PA2 achieves 19× speedup over other methods when aligning >500 kbp ONT reads with 6% divergence, 5.6× speedup for sequences of average length 11 kbp, and only a slight slowdown over BiWFA for very short (<1 kbp) and very similar (<2% divergence) sequences. A*PA2's speed is also comparable to approximate aligners, and is faster for long sequences, thereby nearly closing the gap between approximate and exact methods. A*PA2-simple has similar $O(ns)$ scaling behaviour as Edlib in both length and divergence, but with a 6× to 8× better constant. A*PA2-full achieves the best of both: the near-linear scaling with length of A*PA when divergence is small, and the efficiency of Edlib.

Limitations

1. The main limitation of A*PA2-full is that the heuristic requires finding all matches between the two input sequences, which can take long compared to the alignment itself.
2. For sequences with divergence <2%, the diagonal transition algorithm (BiWFA) is very sparse, and computing full blocks in A*PA2 has considerable overhead.
3. The new sequence profile only supports sequences over alphabet size 4, so DNA sequences containing e.g. `N` characters must either be cleaned or fall back to a slower profile.

Future work

1. When divergence is very low (< 1%), the block-based DP is relatively slow, and performance could be improved by using A* with diagonal transition, possibly per block.
2. Currently A*PA2 is completely unaware of the type of sequences it aligns. Using an upper bound on the edit distance, either known or found using a non-exact method, could avoid trying overly large thresholds and smoothen the curve in Figure 4.8c.
3. It should be possible to extend A*PA2 to semi-global alignment, and to incorporate the ideas of A*PA and A*PA2 back into the Astarix' sequence-to-graph alignment.
4. Extending A*PA2 to affine cost models should also be possible. This will require adjusting the gap-chaining seed heuristic, and changing the computation of the blocks from a bitpacking approach to one of the SIMD-based methods for affine costs.
5. Lastly, TALCO [WYB+24] provides an interesting idea: it may be possible start traceback while still computing blocks, thereby saving memory.



(a) Dijkstra **(b)** WFA **(c)** A*PA

(d) Ukkonen **(e)** Edlib **(f)** **A*PA2 -simple** **(g)** **A*PA2 -full**

Figure 4.12 Alignment of two sequences of length 10 kbp with 17% divergence using the same methods as in Figure 4.1. The optimal alignment contains similar regions, noisy regions, indels, and repeats (shown by the black matches and red pruned matches in (c)). A*PA computes a subset of the states of WFA. A*PA2-full computes more states than A*PA, but is more efficient.

5 Semi-Global Alignment and Mapping

Summary

So far, we have considered only algorithms for *global* alignment. In this chapter, we consider *semi-global* alignment and its variants instead, where a pattern (query) is searched in a longer string (reference). There are many flavours of semi-global alignment, depending on the (relative) sizes of the inputs. We list these variants, and introduce some common approaches to solve this problem.

We then extend A*PA into A*Map as a new tool to solve these problems.

Attribution

All text in this chapter is my own. Some ideas on semi-global alignment are based on early discussions with Pesho Ivanov. Parts of this chapter form the basis for Sassy [BGK25], a tool for approximate string matching of short strings developed together with Rick Beeloo.

In this chapter, we will look at the problem of *semi-global alignment*, as a next step after *global alignment* (Chapter 2). In fact, we will quickly see that there is no such thing as "just" global alignment. Rather, there are many variants that have applications in different domains.

In Section 5.1 we survey the different types of semi-global alignment. Then, in Figure 5.3, we adapt the bitpacking and SIMD method of A*PA2 for $O(nm)$ text searching. In Section 5.3, we extend the gap-chaining seed heuristic of A*PA for semi-global alignment and hence mapping. We use this as the *seeding* and *chaining* parts of the classic *seed-chain-extend* framework. Then we run a separate alignment to find the optimal alignment.

5.1 Variants of semi-global alignment

As we saw in Chapter 3, there are many different types of pairwise alignment (Figure 5.1). Whereas *global* alignment is used when two sequences are to be fully aligned, when that is not the case, there are many different variants. In *semi-global* alignment, we align a sequence to a subset of a longer sequence. While theoretically speaking this captures the full problem, in practice, there are many variants of semi-global alignment that admit different algorithms. In particular, solutions to the problem depend a lot on the absolute and relative size of the two sequences being aligned.

Pairwise semi-global alignment. In this chapter, we will use *(pairwise) semi-global alignment* to refer to instances where the two sequences to be aligned have comparable length. In this case, the number of skipped characters at the start and end of the longer sequence should be small, and not

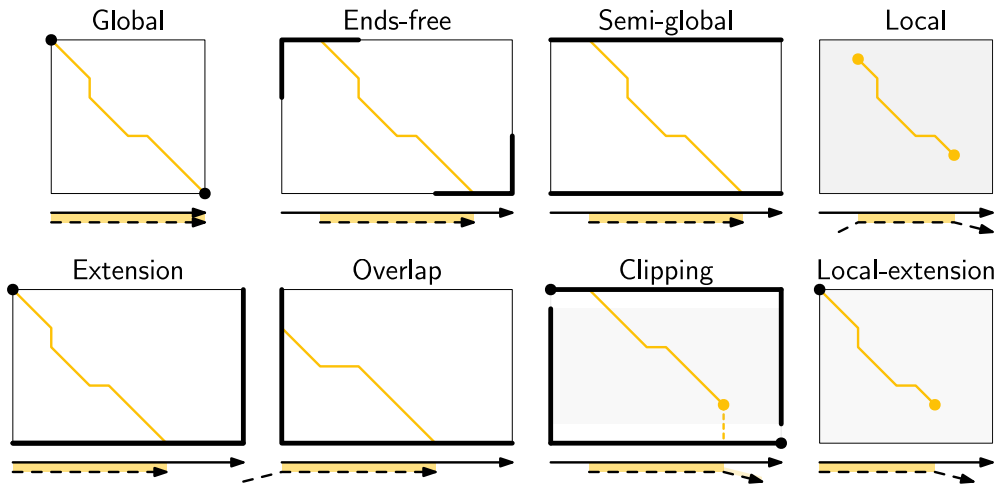


Figure 5.1 Different types of pairwise alignment. Bold vertices and edges indicate where each type of alignment may start and end. Local-extension alignment can end anywhere, whereas local alignment can also start anywhere. Like clipping, these modes require a *score* for matching characters, while in other cases, a *cost model* (with cost-0 matches) suffices. This figure is based on an earlier version that was made in collaboration with Pesho Ivanov.

much more than the edit distance between the two sequences. In this case, we assume that the alignment is *one-off*, i.e., that the both sequences are only aligned once, and that only a single best alignment is needed. Instances of this problem can have length anywhere from 100bp to 100kbp. Pairwise semi-global alignment is especially useful when a pattern has already been approximately located in a text, and a final alignment is done to obtain the precise alignment.

Since the two sequences have similar length, the DP matrix is nearly square, and it suffices to compute states near "the" diagonal. Thus, semi-global alignment is similar to *ends-free* and global alignment, and the classic methods for global alignment can be applied.

Approximate string matching (text searching). In the remaining cases, one of the two sequences (the *text*) is significantly longer than the other (the *pattern*). When additionally the pattern is short, with a length on the order of the machine word size (64 classically, or 256 with SIMD), we classify the instance as *approximate*[1] *string matching* or *text searching*. Here, we are typically interested in not just one optimal alignment, but in finding all sufficiently good alignments, with a cost below some (fixed) threshold. One of the reasons we reasons we consider this a separate problem is that there is a truly vast amount of literature on optimizing approximate string matching. See e.g. [Ukk85b, LV89, CL92, BYG92, WM92, ACG95, BYN96, HFN05, Rog11] for a small selection[2].

Initial solutions for this problem include computing the full matrix ($O(nm)$) and using various types of bitpacking ($O(n\lceil m/w \rceil)$) and parallellization. Additionally, when patterns with up to k errors are to be found, only a subset of the matrix has to be computed and $O(n\lceil k/w \rceil)$ expected time is sufficient [Ukk85b, CL92, Mye99] (Figure 5.3).

One characteristic of nearly all methods for text searching is that they scan the entire text on

[1] Here, *approximate* means that we look for *inexact* matches with a number of mutations.
[2] See `https://curiouscoding.nl/posts/approximate-string-matching` for a longer overview of relevant papers.

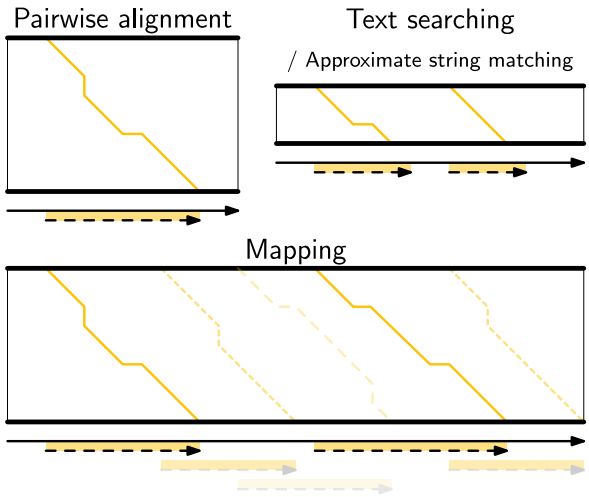


Figure 5.2 Depending on the absolute and relative size of the two sequences, instances of semi-global alignment fall into three categories: pairwise alignment, text searching, and mapping.

each search, and thus require at least $\Omega(n)$ time. Typical applications are searching for a text in a set of files (as can be done by `grep`), and extracting tags and/or barcodes from multiplexed ONT (Oxford Nanopore Technologies) reads.

Mapping. The primary characteristic of *mapping* when compared to approximate string matching is that for mapping, many strings are searched in a single text. Thus, $\Omega(n)$ time per search is not sufficient, especially when the reference has a size anywhere from megabytes to gigabytes. Often, patterns are *long reads* with a length on the order of 50kbp, but also *short reads* of length 100bp can be mapped. To avoid the $\Omega(n)$ per search, the reference is *indexed*, so that regions similar to the pattern can efficiently be identified. Usually, this leads to *approximate* methods, where the reported alignments are not guaranteed to have a minimal cost.

As with approximate string searching, one way to use mapping is by asking for all alignments with cost below some threshold. Another option is to ask only for the few best alignments. In practice, another mode is to first find the cost s of the best alignment (assuming its cost is below some threshold), and then find all alignments up to e.g. cost $s' = 1.1 \cdot s$.

The classic method applied here is *seed-chain-extend* [Li18]. This first finds matches between the sequences, then finds *chains* of these matches, and then fills the gaps in between consecutive matches using relatively small alignments.

5.2 Fast text searching

In A*PA2 (Chapter 4), we developed a block-based method for pairwise alignment. At the core, these blocks are computed using a fast SIMD-based implementation of the bitpacking algorithm of Myers [Mye99, CL92]. So far, we have only used this as a building block for global alignment, but now we will use this to directly support $O(n\lceil m/w \rceil)$ text searching.

In the basis, this requires two changes. First, we ensure that the alignment can start anywhere in the text by changing the horizontal differences along the top row of the matrix from 1 (as used by global alignment) to 0, as indicated by the bold lines in Figure 5.3.

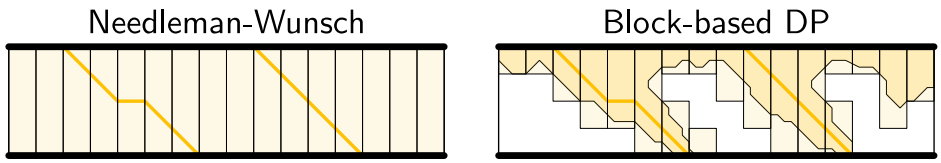


Figure 5.3 Text searching is the problem of finding a typically short (length $O(w)$) pattern in a longer text. The left shows how the classical Needleman-Wunsch algorithm fills the entire matrix column by column. On the right (adapted from [Mye99]), we search for all alignments with cost $\leq k$, and states at distance $\leq k$ are highlighted. The block-based approach only computes blocks that contain at least one state at distance $\leq k$, and takes $O(n\lceil k/w \rceil)$ time in expectation on random strings [CL92].

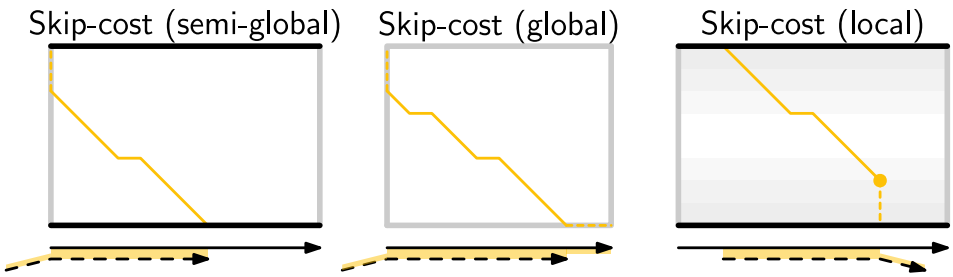


Figure 5.4 By default, global alignment uses a cost of 1 along all edges of the matrix, while semi-global alignment and overlap/ends-free/extension variants have a cost of 0 along some edge. When a pattern only partially overlaps the text, as shown on the left, it may be preferable to have a *skip-cost* α for each unmatched character that is in between 0 and 1. This can also be applied to global alignment (replacing ends-free alignment), and can be an alternative to local alignment.

Secondly, the alignment may end anywhere, and the user may be interested more than just a single best alignment. To support this, we do not only report the score in the bottom right of the DP matrix, but we return a list of all scores along the bottom row. Based on this, the user can decide which scores are sufficiently low to find a full alignment.

Tracing. Once the user decides which scores at the bottom of the matrix are sufficiently low, a traceback be started from those positions. To save time and memory, the initial computation of the matrix only returns the output scores and does not store all nm values. Thus, to find an alignment ending in column i, we recompute the matrix from column $i - 2m$ to column i and store all values for each column. We then do a usual trace through this matrix from $\langle i, m \rangle$ until we reach the top row ($j = 0$).

5.2.1 Skip-cost for overlap alignments

In some applications, it may happen that the pattern is present, but cut off at either its start or end, as shown on the left in Figure 5.4. For example when a read was cut short, or when aligning reads against an incomplete assembly [AKBP24]. In a classical semi-global alignment, the unmatched start of the pattern would incur a cost of 1 per unmatched character, but this may make the total cost of the pattern go above the threshold. Instead, overlap alignment could be used (Figure 5.1), but this requires a bonus for matches, since otherwise the cheapest way to align the pattern could be to skip nearly all of its characters. Ends-free alignment solves this by only allowing a limited number of characters to be skipped. Still, this is suboptimal: when the pattern matches once in

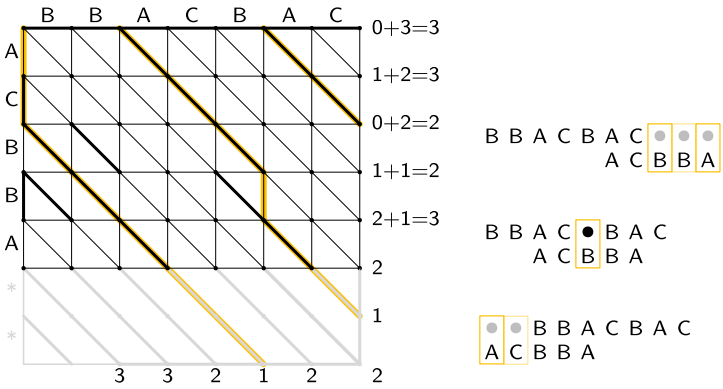


Figure 5.5 Example of computing a semi-global alignment with a skip-cost of $\alpha = 1/2$. In the first column the graph, edges of cost 1 and 0 alternate. On the bottom, the graph is extended with matches until a multiple of the block size is reached. On the right, the final score in row j is increased by $\lceil \alpha(m-j) \rceil = \lceil (m-j)/2 \rceil$ to obtain the score including skip-cost. Three alignments are highlighted and shown, with edits highlighted. Only half of the skipped characters (rounded up) incurs a cost.

full, and once at the start of the sequence with 50% overlap, the scores of these two alignments are not directly comparable. In fact, the overlapping alignment has a benefit because it only pays for mismatches in half its length.

To solve this, we introduce the skip cost[3] $0 \le \alpha \le 1$, which is the cost paid for each character at the start and/or end of the pattern that is not aligned because it extends outside the text. This concept can also be applied to global-alignment variants such as ends-free and overlap (Figure 5.4, middle), so that skipping characters in both sequences has a (not necessarily equal) cost.

In practice, it is not practical to handle fractional costs, especially in the case of edit distance where the distance between adjacent states must be 0 or 1. To avoid this, we can initialize the first and last column (and row, for global alignment) with a mix of zeros and ones, so that the *fraction* of ones is approximately α, as shown in Figure 5.5 for $\alpha = 0.5$.

Applying the skip-cost. In Figure 5.6, we show an example output when using a skip-cost of $\alpha \in \{0, 0.5, 1\}$ for the alignment as shown in Figure 5.7. Using $\alpha = 1$ corresponds to classical semi-global alignment (thin black), and we see that this correctly detects that the pattern matches in the middle of the sequence, ending at position 300, with a cost around 20. However, the occurrences overlapping the start and end of the text are completely missed. Overlap alignment, which corresponds to $\alpha = 0$ (bold black) *does* have local minima at position 50 and 650 (indicating the pattern extends 50 characters beyond the text). The drawback of these minima is that there are also *global* minima at positions 0 and 700 where the pattern is completely disjoint from the text, so that some additional logic is needed to separate these cases. We see that in regions where the pattern does not match, the alignment has a score around 50, or 0.5 per character. Thus, we choose $\alpha = 0.5$ per skipped character. Using this (yellow), we recover clear local minima at positions 50 and 650, while the cost converges back to 50 as the overlap shrinks to 0.

[3] I would not be surprised if this has been done before. There are many tools applying similar techniques (either via local alignment or a clipping cost), but as far as I am aware, the technique as stated here has not been applied before.

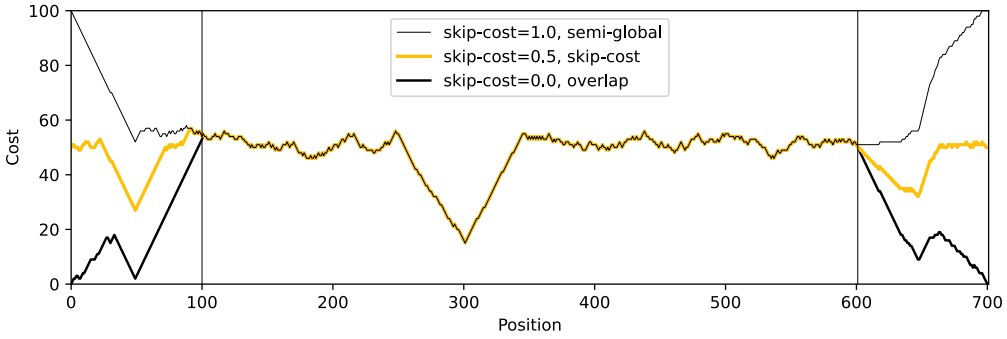


Figure 5.6 Example of the output of the skip-cost alignment when aligning a length-100 pattern onto a length-600 text (as shown in Figure 5.7). Graphs are shown for $\alpha = 1$, corresponding to classical semi-global alignment, $\alpha = 0.5$, corresponding to the skip-cost introduced here, and $\alpha = 0$, corresponding to an overlap alignment. Vertical lines indicate the region inside of which the pattern fully matches within the text, and where the cost of the alignment does not depend on the skip-cost α.

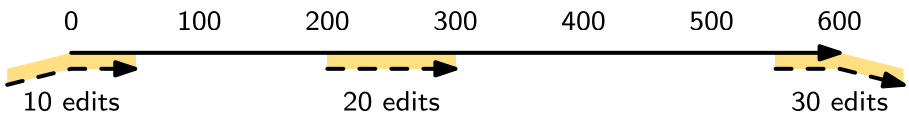


Figure 5.7 The setup of the alignment results shown in Figure 5.6. A random pattern of length 100 is generated and overlaid on a length 600 text 3 times: once in the middle, and twice with a 50 base overlap at the start/end of the sequence. Before inserting the pattern into the text, a different number of mutations is applied to the full length-100 pattern.

5.2.2 Results

We benchmark the throughput of the search function in Figure 5.8, where we measure how long it takes (per text character) to align a pattern against a text. For Edlib [ŠŠ17], we use the *infix* method for semi-global alignment and ask it to report the distance only, and likewise for our method, we measure only the time needed to compute the output distances. Experiments are run on an Intel `i7-10750H` with AVX2, running at a fixed CPU frequency of 2.6 GHz.

As can be seen, both methods take as long for pattern length 32 as for 64, since they pad to 64 bit values. Our SIMD-based method has constant performance up to patterns of length 256, and then grows linearly with the pattern length. Edlib starts to grow at its word size $w = 64$ instead. On very divergent sequences (black), indeed the growth is linear, and even slightly worse because of redundant band doubling. For more similar sequences (grey), when the pattern is present in the text with a small divergence, band doubling reduces the part of the matrix that needs to be computed. Especially when the pattern can be found with a divergence of 1%, this makes the performance nearly independent of the pattern length, as also predicted by Myers' complexity of $O(n\lceil k/w\rceil) = O(n\lceil 0.01m/64\rceil) = O(n\lceil m/6400\rceil)$ [Mye99] and shown in Figure 5.3.

For shorter texts, on the order of the pattern length (not shown), there is an additional 50% to 100% overhead on the time per character that is spent on preprocessing the pattern.

When also tracing the optimal alignment, Edlib needs another 5-10% of time, while our method needs an additional 10-20%.

For patterns of length 128 to 256, our method ends up around 1.7× to 2.0× faster than Edlib. In practical terms, this implies that a pattern of length up to 256 bp can be found in a 1 kbp read

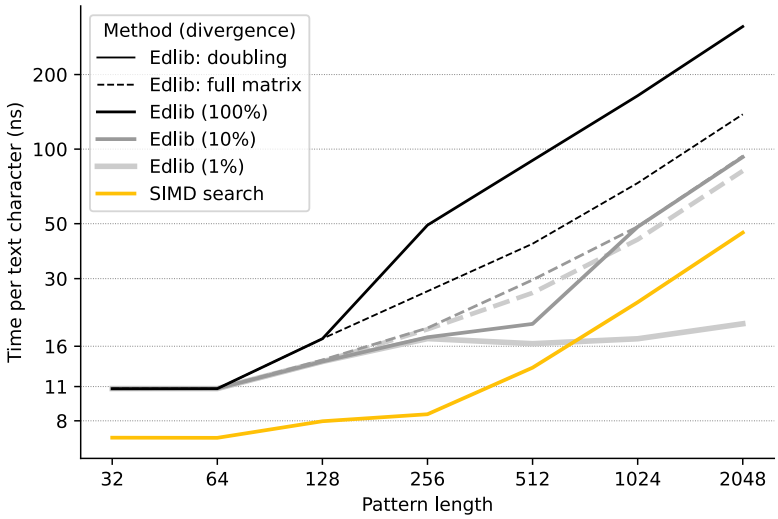


Figure 5.8 Log-log plot of the time to align a pattern of length m against a text of length 50 kbp, in nanoseconds per base of the text. Only the time needed to compute the minimal distance is reported, excluding alignment/traceback. Our SIMD search method (yellow) always computes the entire matrix. Edlib, on the other hand, by default uses a band doubling approach (solid lines). Disabling this via a fixed high threshold is shown dashed.

in 13 micro seconds (75000 searches per second) or in a 50 kbp text in 440 micro seconds (2200 searches per second). Or alternatively, in one second, nearly 100 Mbp of text can be searched.

Future work. Currently, we only implement a naive $O(n\lceil m/w \rceil)$ method that always computes the entire matrix. For sequences of length greater than 256, most of the matrix below the first 256 rows can likely be skipped, and this should provide a significant speedup.

5.3 Mapping using A*Map

The problem of *mapping* differs from text searching considered so far in a few ways. First, the text (*reference*) is fixed and is reused for many alignments. It can be anywhere from megabases to gigabases in size. Secondly, the patterns (*reads*) being mapped can have length 100 bp (*short reads*) up to 50 kbp (*long reads*). To enable efficient mapping, most tools build an *index* on the reference, and then query this for each read to be mapped. In practice, such methods are often *approximate*, in that they are not guaranteed to find a minimal-cost alignment. They work using *seed-chain-extend*: *seeds*[4] (usually k-mer matches) are found via the index. Then these are joined into *chains*, and the best chains are *extended* into a full alignment, as shown in Figure 5.9.

In the remainder of this section, we briefly review strategies for the three parts, *seeding*, *chaining*, and *extending*.

[4] We somewhat interchangeably use *seeds* and *matches* here. To me, a *seed* is a conceptual anchor that can be extended into an alignment. A *match* is the specific type of anchor we use: our seeds are usually matches between k-mers.

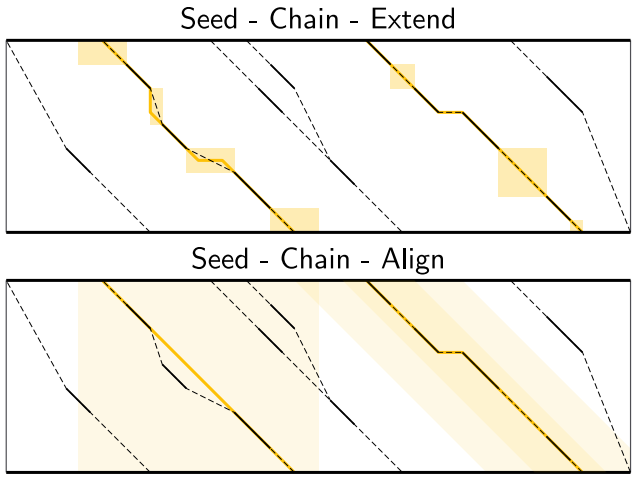


Figure 5.9 An example of the *seed-chain-extend* method for mapping. First, *seeds* (black diagonals) are found, which are short matches between the two sequences. Then, these seeds are *chained* into *chains* (dashed lines). Each seed and each chain is scored based on the number of seeds in the chain and their relative positions. The chains with the highest scores are selected as candidate alignments. Then, short alignments are done to fill the gaps between the seeds and *extend* the chain into a full alignment. A drawback of seed-chain-extend is that it may not return optimal alignments. Instead, a full semi-global alignment could be done around the chain to obtain an exact alignment, leading to *seed-chain-align*. The bottom left shows a semi-global alignment using Needleman-Wunsch, and the bottom-right showh a semi-global alignment using band-doubling.

A*Map builds on the same paradigm, and we review how A*PA's *gap-chaining seed heuristic* can be applied here, and how A*PA and A*PA2 can be modified for *exact* mapping and semi-global alignment. Note that in A*Map, we replace the usual *extend* phase by a more thorough semi-global alignment that covers the full chain at once. This way, we can guarantee that optimal alignments are found.

5.3.1 Seeding

There are various strategies for seeding alignments.

Minimizers. The most popular mapper, minimap2 [Li18], uses *minimizers* (Part II). By default, it uses k-mer size k from 15 to 19 and window size w from 10 to 19, to extract one out of each w consecutive k-mers. It first finds all minimizers of the reference and builds an index that maps each k-mer to the locations where it occurs as a minimizer. Then, the minimizer k-mers for each query are determined, and these are looked up in the index to find the k-mer matches that seed the alignment.

k-min-mers. A different approach is taken by mapquick [ESM+23], which is a mapper designed for highly similar sequences. Here, *k-min-mers* are used to seed the alignment. These are chains of 2 to 15 consecutive 31-mers. This way, each k-min-mer spans a much larger portion of the sequence, and fewer matches are needed to recover sufficiently good chains.

k-mers. In the *seed heuristic* in A*ix [IBV22] and A*PA [GKI24] (Chapter 3), plain k-mer matches are used. A drawback of this approach is that it creates more matches, since there are more k-mers

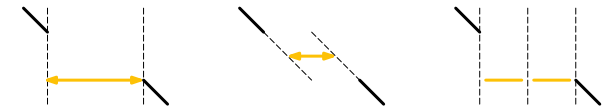


Figure 5.10 There are different models to give costs and scores to chains. Here we show three possible costs that can be given to the connection between

than minimizers. The main benefit, on the other hand, is that it leads to an *exact* algorithm. For other seeding methods, a lack of matches does not imply a (good) lower bound on the minimal edit distance between consecutive matches, as we will see in Section 5.3.2.

Maximal-exact-matches. *Maximal-exact-matches* are a variant where k-mer matches are extended on either side as long as the two sequences match. This is similar to the seeding used by BLAST [AGM+90].

Maximal-unique-matches. Yet another method is to seed the alignment using *maximal-unique-matches*, also known as MUMs. These are substrings of the query and reference that occur exactly once in each string, and that can not be extended into a longer matching substring. Thus, these matches consider global information, rather than just considering local matches. This technique is used by MUMmer [DKF+99, MDP+18];

5.3.2 Chaining

After finding all the seed matches, the next step is to find candidate regions where the query could align. This is done by finding *chains* consisting of multiple matches, and giving each chain a cost or score. Specifically, a chain is a sequence of seeds that can occur together in an alignment.

As for seeding, there are many different methods to score chains.

LCSk. A simple method of scoring chains is to assume that the seeds are disjoint k-mer matches, and simply maximize the number of k-mers in the chain. This is also known as the LCSk metric. [BLS13]. Like the plain LCS, this score focuses only on matches, and disregards the mismatches and indels in between.

LCSk++. An extension of LCSk is LCSk++ [PŽŠ14]. This method allows matches of arbitrary length, and maximizes the total length of the matches.

Anchored edit distance. As with edit distance, we can consider a *cost* equivalent of the *score* given by the LCSk++ metric. This is the *anchored edit distance* [JGT22], where the focus in again on the mismatches and indels rather than the matches. As shown in Figure 5.10, the cost of joining two seeds is the maximum of the horizontal and vertical gap between them.

Gap cost. We already saw that the gap cost [Ukk85a]. is used a lot for pairwise alignment, and it is also useful as a cost for chaining matches: we can lower bound the cost of the alignment between two consecutive matches by the minimal number of horizontal or vertical steps needed to join them (Figure 5.10). Indeed, minimap2 [Li18] also uses a chaining score based on the gap cost. In fact, minimap2 uses a concave function of the size of the gap as actual distance, so that longer gaps are penalized relatively less than short gaps, to admit e.g. splicing alignments.

Seed heuristic (SH). The *seed heuristic*, introduced by A*ix [IBV22, GKI24], provides a second, independent lower bound on the edit distance between two matches. We first find *all* k-mer matches. Then, say that there is a gap of $\Delta_i \times \Delta_j$ bases between two matches in our chain. Assuming that there no in-between matches, we know that there is no k-mer match in the path joining the two matches. Thus, we must incur an error at least every k steps, for at least $\max(\lfloor \Delta_i/k \rfloor, \lfloor \Delta_j/k \rfloor)$ errors. (If we assume that the two initial matches are already maximally extended, we could replace the $\lfloor \cdot \rfloor$ by a $\lceil \cdot \rceil$.) In practice, the seed heuristic is implemented by splitting the reference sequence into adjacent disjoint k-mers, and only matches of those k-mers are found. Then, the distance between consecutive matches is always a multiple of k, and the minimal cost to join them is simply the number of skipped k-mers, as shown in Figure 5.10.

Gap-chaining seed heuristic (GCSH). In A*PA, we extended the seed heuristic into the gap-chaining seed heuristic. Conceptually, this simply takes the maximum of the gap-cost and the seed heuristic cost, since the maximum of two lower bounds is still a lower bound. The main theoretical result of A*PA (Theorem 5, Lemma 7) is the following:

> In an optimal path, two matches can only be chained if the gap cost between them is at most the value of the seed heuristic between them.

Thus, two matches that are d diagonals apart may only be chained if there are at least $k \cdot d$ columns between them. This puts a strong limitation on how far chains can "stray away" from their diagonal. In A*PA, we provide an efficient $r \lg r$ algorithm for chaining r matches that is equivalent to the solution for LCS [Hir77]. It works by first applying a suitable *transformation* to the coordinates of the matches, followed by a plain LCS algorithm.

The main benefit of the GCSH is that it gives mathematical guarantees. Suppose we are doing a global alignment between two sequences of length n (the one that is split into $\ell = \lfloor n/k \rfloor$ k-mer *seeds*) and m. If there is an alignment of cost s, then we know for sure that there is also a chain of cost $\leq s$. Thus, to find all alignments of cost up to s, we only have to consider all chains with cost up to s.

5.3.3 Aligning

After all matches have been chained and sufficiently good candidate chains have been determined, this chain can be extended into an alignment. Minimap2 uses the KSW2 algorithm [SK18] to do an approximate (banded) alignment to fill the gaps between matches. Other methods such as mapquick completely the alignment phase completely and only report the location and/or score of the chain.

A drawback of extending a chain is that the optimal alignment may not completely follow the chain, as exemplified in the bottom-left alignment in Figure 5.9. Instead, we can run a semi-global alignment around the chain using any of the global alignment methods discussed in Chapter 3, such as a plain Needleman-Wunsch DP or band doubling. Indeed, we can also use A*PA or A*PA2 for this semi-global alignment.

Updating GCSH for semi-global alignment. For global alignment we can simply count the number of seeds that is still to be covered to get to the end of the first sequence (the reference). In particular, when $x = n - i$ characters of the first remain, we need to still cross and pay for $x/k - O(1)$ seeds. With semi-global alignment, we can end the alignment anywhere, and avoid crossing all seeds. If there are still y bases of the *pattern* remaining, it turns out this will need a cost of at least $y/(k+1) - O(1)$. This division by $k+1$ rather than k could be avoided by replacing the role of the

pattern and reference, and splitting the pattern into m/k seeds, but that turns out to be inefficient when it comes to indexing all k-mers. By splitting the reference into k-mers, we only need to index $1/k$ of its k-mers, so that this index is much smaller.

Secondly, in A*PA and A*PA2 we filter away all matches for which the their gap cost to the end ($\langle n, m\rangle$) is *larger* than the seed heuristic cost to the end, since these can provably never be part of a chain. With semi-global alignment, chains can end anywhere, and thus this filter does not apply anymore.

Semi-global alignment using A*PA and A*PA2. We additionally make some modifications to A*PA and A*PA2. First, the alignment can start anywhere along the top of the grid, and so we do not only push the root state $\langle 0, 0\rangle$ on the A*PA priority queue, but we push all states along the top row for which the heuristic has a local minima. From there, we expand sideways as needed, both to the right *and to the left*. For A*PA2, we similarly make sure to cover all start positions with sufficiently low value of the heuristic.

Similarly, the alignment may end anywhere on the bottom row, and so the termination condition is changed accordingly. Also during the traceback, we ensure that this is stopped as soon as the top row is reached, rather than the top-left state.

5.3.4 A*Map

While it would be possible to using A*PA or A*PA2 directly as a mapping algorithm, this is inefficient because the index on reference k-mers is not reused between alignments. Thus, we develop A*Map as a dedicated mapper. As discussed, this consists of three components:

- Seeding using k-mer matches: a static index is built containing exactly every k'th k-mer of the reference, and all query k-mers are looked up in this to find their matches in the reference. Matches are sorted using an efficient radix sort.
- Chaining using the gap-chaining seed heuristic (GCSH): all r matches are transformed as done by A*PA, and then an efficient implementation of the $r \lg r$ LCS chaining algorithm is used. All chains with a cost below some fixed threshold t are candidates for alignment.
- Candidate chains are semi-global aligned using A*PA2 with band doubling. The best score is tracked and returned. To ensure the alignment is contained in the subsequence of the reference that is semi-globally aligned, a small buffer is added before the first match and after the last match, as shown in Figure 5.9.

When the goal is to find all alignments with divergence up to $d = 4\%$, one must use a value of k somewhat below $0.9(1/d - 1) = 0.9(1/4\% - 1) = 21.6$ to accommodate spurious matches, and to ensure that candidate chains contain at least one tenth of the maximum possible number of matches (i.e., chains should have length at least $0.1 \cdot m/k$). In this case, $k = 20$ would be a good choice. Generally, smaller k is preferred to improve the quality of the heuristic, but we also need $k > \log_4 n$ to ensure that the number of spurious matches remains limited.

5.3.5 Results

In Table 5.1, we compare A*Map against minimap2 on synthetic long read data. We use chromosome 1 as the reference, which has length around 235 Mbp. From this, we sample 1000 random reads of length 50 kbp. Then, we apply a varying number of uniform random mutations to these strings 1%, 3%, and 5%, to obtain divergences of 0.9%, 2.7%, and 4.4%.

Table 5.1 Results of aligning 1000 random subsequence of chromosome 1, with varying divergence. The first row shows the time in seconds to index the 235 Mbp chromosome, and remaining rows show the total time to map the 1000 reads. For minimap2, we try various default configurations, while for A*Map we use $k = 20$ and $k = 28$. (*): For $k = 28$, the alignments found with 4.4% divergence are not guaranteed to be exact, since k is larger than $1/d = 1/4.4\%$, and indeed, 3 reads remain unmapped. The PacBio mode uses homopolymer compression.

Divergence	Minimap			A*Map	
	PacBio (+hpc) $k = 19$, $w = 10$	ONT $k = 15$, $w = 10$	HIFI $k = w = 19$	$k = 20$	$k = 28$
Indexing	8.7	10.5	7.4	1.2	1.0
0.9%	31.2	48.1	23.0	26.9	**8.7**
2.7%	31.4	50.6	23.3	24.6	**12.9**
4.4%	28.8	46.6	**21.8**	22.8	(*) 15.4

We run both methods on a single thread. For minimap2, we run with `-x map-pb`, `-x map-ont` (default), and `-x map-hifi`. For A*Map, we use A*PA2 with plain band doubling [Ukk85a] for the semi-global alignments.

Experiments are run on an Intel `i7-10750H` with AVX2, running at a fixed CPU frequency of 2.6 GHz.

A*Map analysis. When $k = 20$, a bottleneck of A*Map is the large number of k-mer matches: 200000 on average per mapped read. For divergence 2.7%, 3 seconds are spent collecting matches, 4.5 seconds are needed to sort them, and 5 seconds to chain them. Aligning the most likely chain for each read takes a total of 8 seconds. On average, there are 2.6 candidate chains per pattern. It appears that this is mostly due to reads falling into highly repetitive regions (e.g. in the centromere), where many overlapping starting positions for the semi-global alignment are considered. These (on average) 1.6 additional alignments per read take a total of 3.8 seconds.

When $k = 28$, the number of matches is significantly reduced, to only 30000 per read. This reduces the time spent sorting matches to 0.6 s, and the time for building contours to 0.8 s. Also, the number of candidate chains drops from 2.6 to 1.4 per read, since the larger k increases sensitivity. The total time for aligning the best scoring chains is still 8 seconds.

Comparison. Compared to minimap2, A*Map is significantly faster at indexing the text, since it only needs to build a hashtable on every k'th k-mer. Minimap2, on the other hand, has to compute all minimizers. Nevertheless, minimap2's indexing could probably be sped up by using SimdMinimizers (Chapter 11). On this data, minimap2 works best with the HIFI preset, with $k = w = 19$. For divergence 0.9%, A*Map is 2.6× faster, and for divergence 2.7%, A*Map is 1.8× faster. To put these results into perspective, on data with $< 1\%$ divergence, mapquick [ESM+23] was shown to be more than an order of magnitude faster than minimap2.

Nevertheless, the main feature of A*Map is that it is able to guarantee exact results, where one can prove that no alignment below the threshold is missed. In cases where this is important, A*Map is a viable alternative to minimap2.

Future work. Currently, there are a few limitations. First, the semi-global alignment is independent of the preceding chaining. It could be beneficial to reuse the chains to build a heuristic, to reduce the size of the subsequent alignment. However, initial experiments show that the overhead of

evaluating the heuristic quickly grows compared to simply computing more states. Alternatively, it may be possible to develop an exact alignment method that is *bottom-up* (like the usual extending) by building on ideas such as *local pruning* introduced by A*PA2.

A second issue is the large number of matches, and the time needed to query *all* k-mers of the read. One way to speed this up is to swap the roles of the query and the reference, so that only every k'th query has to be looked up. However, that comes at the cost of a $k\times$ larger index. Alternatively, fine-tuning the value of k so that it is small enough for the given error rate and as large as possible to reduce false positive matches could also help. In parallel, it may be possible to build an efficient index on inexact matches of length $2k$, so that there are simply fewer resulting matches that have to be sorted and chained.

Part II

Low Density Minimizers

6 Theory of Sampling Schemes

Summary

In this chapter, we introduce *minimizers*, and more general *sampling schemes*. These are methods to sample a subset of k-mers from a text in a deterministic way, such that at least one k-mer is sampled from every string of length $k + w - 1$, for some *window guarantee* w.

A key property of such schemes is their *density*: how few k-mers can be sampled while still respecting the window guarantee. Applications benefit from a lower density, since this usually means indices on the text are smaller.

We cover a number of theoretical results from the literature, and explain how the density of a minimizer scheme can be computed. This will serve as the background for the following two chapters, that go deeper into lower bounds on sampling scheme density and practical sampling schemes.

Attribution

In this chapter, we survey existing literature on minimizers. Small parts of this chapter are loosely based on the background sections of the mod-minimizer papers [GKP24, GKLP25], written with Giulio Ermanno Pibiri and Daniel Liu, and the density lower bound paper [KGKM+24], written with Bryce Kille and others. Specifically some definitions and theorem statements are copied verbatim.

6.1 Introduction

As does nearly every paper in bioinformatics, we will start here with the remark that the amount of biological data has been and still is increasing rapidly over time. Thus, both faster and more space efficient algorithms are needed to analyse this data. One popular method for sequence analysis is to consider a sequence as its set of *k-mers*, i.e., the set of substrings of length k, which often has a value up to 31. Then, two sequences can be compared by comparing these sets, rather than by doing a costly alignment between them. For example, when comparing two human genomes, there will likely be relocations and inversions of long segments of DNA, so that a global alignment is not the best way forward. By instead only considering k-mers, we only consider the local structure, so that the changes in global structure do not affect the analysis. Then one could estimate the similarity of these sets using min-hash [Bro97, KGTP23]. Or one could seed an alignment by finding k-mer matches between the two sequences, as done by minimap [Li16].

A drawback of considering the full set of k-mers is that consecutive k-mers overlap, and thus that there is a lot of redundancy. To compress the set of k-mers, it would be preferable to only sample every k'th k-mer, so that each base is covered only exactly once. That, however, doesn't work: if we are given two sequences, but one is shifted by one due to an inserted base at the

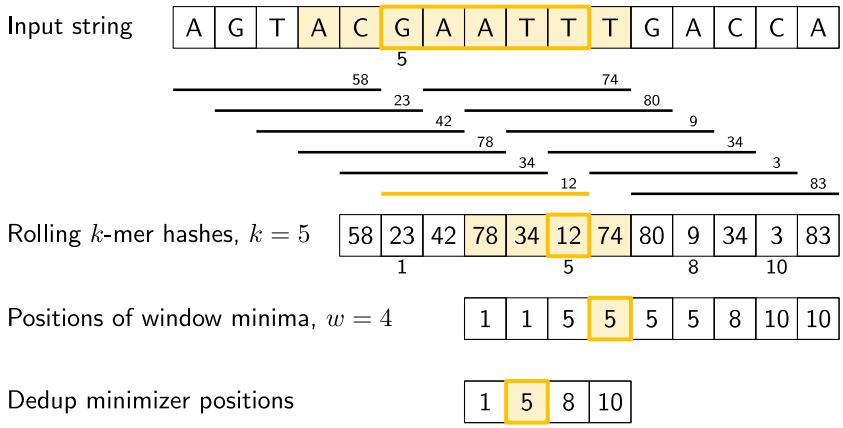


Figure 6.1 An example of the random minimizer. First all k-mers of length $k = 5$ are hashed using a random hash function, as shown by the black lines with their hash written above them. Then, in each window of $w = 4$ consecutive k-mer hashes, as e.g. the window highlighted in yellow, the (position of) the smallest k-mer hash is recorded. Then, the (positions of) the smallest k-mers in all windows form the set of minimizers. This figure is taken from [GKM25] and made in collaboration with Igor Martayan.

very start, no common k-mers will be sampled. To fix this, we would like a *locally consistent* sampling method, that decides whether or not to sample a k-mer only based on the k-mer and some surrounding context. One option is to use all k-mers that start with an `A`, or to use all k-mers whose (lexicographic or hashed) value is sufficiently small. Indeed, this approach is sufficient for some applications.

A drawback of the methods so far is that there could be long stretches of input without any sampled k-mers. This can cause long gaps when seeding alignments. Also when building text indices, such as a sparse suffix array [GR17, ALP23] or a minimizer-space De Bruijn graph [EBC21], we usually need a guarantee that at least one k-mer is sampled every, say, w positions.

Sampling scheme. This brings us to *local sampling schemes* that satisfy a *window guarantee*: given a window of w k-mers, consisting of $w + k - 1$ characters, a sampling scheme returns one of the k-mers as the one to be sampled. If we now slide this window along the input, we collect the set of sampled k-mers. Ideally the scheme ensures that the same k-mer is sampled from adjacent windows, so that this set is much smaller than the total number of k-mers.

Given such a sampling scheme, we can build a *locality sensitive hash*: given a string of length $w + k - 1$, we can hash it to its sampled k-mer. This way, adjacent strings are likely to have the same hash, allowing for efficient indices such as LPHash and SSHash [PSL23, Pib22].

Problem statement. The goal now is to find k-mer sampling schemes that satisfy the w window guarantee, while overall sampling as few k-mers as possible.

This is formalized in the *density*: the expected fraction of k-mers sampled by a scheme. The window guarantee imposes a trivial lower bound of $1/w$, since at least every w'th k-mer must be sampled.

The most common practical scheme is the *random minimizer* [RHH+04, SWA03], which hashes all k-mers in the window, and chooses the smallest one, as shown in Figure 6.1. This has a density of $2/(w + 1)$, roughly twice as large as the trivial lower bound.

Now, you may ask whether it's worth the effort to *at best* save a factor two of memory and speed, and you would not be alone. I asked myself this same question long before becoming interested in minimizers (at RECOMB 2022):

> Why would we even care about better minimizer? We have this simple and fast random minimizer that's only at most 2× away from optimal. Why would anyone invest time in optimizing this by maybe 25%? There are so much bigger gains possible elsewhere.

Even worse, in practice, most schemes do not (and as we will see, can not) give more than a 20% saving in density on top of the very simple random minimizer.

As we will prove in Chapter 7, when $k \leq w + 1$, we can never expect more than around 25% lower density than the random minimizer, and thus, existing schemes are already relatively close to optimal from a practical point of view. Given this new lower bound, maybe the answer is that, indeed, we should stop searching for further schemes.

Nevertheless, there is a lot of pretty maths to be found, both in the lower bounds, and in the many sampling schemes we will review and develop ourselves. Furthermore, as we will see with the mod-minimizer, when $k > w$, we can often achieve this 25% lower density using only slightly more code, so nearly for free. Likewise, for smaller $k \geq w$, we will see that choosing a specific order of the alphabet can improve the density of the random minimizer by over 10%, while *simplifying* the code. Thus, even though improvements may seem small, searching for simple sampling schemes with near-optimal density is a fruitfull research area.

6.2 Overview

This part on minimizers is split into four chapters.

Here, in Chapter 6, we review the existing theory of sampling schemes. This covers, for example, *local*, *forward*, and *minimizer* schemes, the density of the random minimizer, and a number of theoretical results on the optimal asymptotic density of schemes as $k \to \infty$ or $w \to \infty$.

In Chapter 7, we go over existing lower bounds. We start at the original one of Schleimer et al. [SWA03], that turns out to make overly strong assumptions, and end with the new nearly tight lower bound of [KGKM$^+$24].

Then, in Chapter 8, we summarize and compare existing sampling schemes, and introduce two new sampling schemes: the *open-closed* minimizer and the general *mod-minimizer* [GKP24, GKLP25]. The most important result is that the mod-minimizer achieves density $1/w$ in the limit where $k \to \infty$, and that this convergence is close to optimal.

Lastly, in Chapter 9, we consider the special case where $k = 1$. Here, we first introduce the *bd-anchor*, and then improve this into the sus-anchor with anti-lexicographic sorting. This has density very close to optimal for alphabet size $\sigma = 4$, and raises the question whether perfectly optimal selection schemes can be constructed.

Many of the existing papers on sampling schemes touch more than one of these topics, as they both develop some new theory and introduce a new sampling scheme. These results are thus split over the applicable sections.

Previous reviews. This chapter is written from a relatively theoretical perspective, and focuses on the design of low-density sampling schemes. The review of Zheng et al. [ZMK23] takes a more practical approach and compares applications, implementations, and metrics other than just the density. It limits itself to *minimizers*, rather than more general local or forward schemes. A second

review of Ndiaye et al. [NPBnF+24] focuses specifically on applications, and details how minimizers are used in various tools and domains of bioinformatics.

6.3 Theory of sampling schemes

The theory of minimizer schemes started with two independent papers proposing roughly the same idea: winnowing [SWA03] in 2003 and minimizers [RHH+04] in 2004. At the core, the presented ideas are very similar: to deterministically sample a k-mer out of each window of w consecutive k-mers by choosing the "smallest" one, according to either a random or lexicographic order. The *window guarantee* is a core property of minimizers: it guarantees that consecutive minimizers are never too far away from each other. Further, these schemes are *local*: whether a k-mer is sampled as a minimizer only depends on a small surrounding context of $w-1$ characters, and not on any external context. This enables the use of minimizers for locality sensitive hashing [PSL23, Pib22], since the minimizer is a deterministic key (hash) that is often shared between adjacent windows.

While the winnowing paper was published first, the 'minimizer' terminology is the one that appears to be used most these days. Apart from terminology, also notations tend to differ between different papers. Here we fix things as follows.

6.4 Notation

Throughout this chapter, we use the following notation. For $n \in \mathbb{N}$, we write $[n] := \{0, \ldots, n-1\}$. The alphabet is $\Sigma = [\sigma]$ and has size $\sigma = 2^{O(1)}$, so that each character can be represented with a constant number of bits. For all evaluations we will use the size-4 DNA alphabet, but for examples we usually use the plain `ABC..XYZ` alphabet. Given a string $S \in \Sigma^*$, we write $S[i..j)$ for the sub-string starting at the i'th character, up to (and not including) the j'th character, where both i and j are 0-based indices. A k-mer is any (sub)string of length k.

In the context of minimizer schemes, we have a *window guarantee* w indicating that at least one every w k-mers must be sampled. A *window* is a string containing exactly w k-mers, and hence consists of $\ell := w + k - 1$ characters. We will later also use *contexts*, which are sequences containing two windows and thus of length $w + k$.

6.5 Types of sampling schemes

▶ **Definition 6.1** (Window). *Given parameters w and k, a* window *is a string containing exactly w k-mers, i.e., of length $\ell = w + k - 1$.*

▶ **Definition 6.2** (Local sampling scheme). *For $w \geq 1$ and $k \geq 0$, a* local scheme *is a function $f : \Sigma^\ell \to [w]$. Given a window W, it* samples *the k-mer $W[f(W)..f(W)+k)$.*

In practice, we usually require $w \geq 2$ and $k \geq 1$, as some theorems break down at either $w = 1$ or $k = 0$ (even though theoretically those parameters make sense: we can either sample every position when $w = 1$, or for $k = 0$ sample the empty substring following the final character). When $k \geq w$, such a scheme ensures that every single character in the input is covered by at least one sampled k-mer.

▶ **Definition 6.3** (Forward sampling scheme). *A local scheme is* forward *when for any* context C *of length $\ell + 1$ containing windows $W = C[0..\ell)$ and $W' = C[1..\ell+1)$, it holds that $f(W) \leq f(W') + 1$.*

Forward scheme have the property that as the window W slides through an input string S, the position in S of the sampled k-mer never decreases.

▶ **Definition 6.4** (Order). *An order $\mathcal{O}_k$ on k-mers is a function $\mathcal{O}_k : \Sigma^k \to \mathbb{R}$, such that for $x, y \in \Sigma^k$, $x \leq_{\mathcal{O}_k} y$ if and only if $\mathcal{O}_k(x) \leq \mathcal{O}_k(y)$.*

▶ **Definition 6.5** (Minimizer scheme). *A* minimizer scheme *is defined by a total order $\mathcal{O}_k$ on k-mer and samples the leftmost minimal k-mer in a window W, which is called the* minimizer*:*

$$f(W) \coloneqq \operatorname{argmin}_{i \in [w]} \mathcal{O}_k(W[i..i+k)).$$

Minimizer schemes are always forward, and thus we have the following hierarchy

$$\text{minimizer schemes} \subseteq \text{forward schemes} \subseteq \text{local schemes}.$$

There are two particularly common minimizer schemes, the *lexicographic* minimizer and the *random* minimizer.

▶ **Definition 6.6** (Lexicographic minimizer [RHH+04]). *The* lexicographic minimizer *is the minimizer scheme that sorts all k-mers lexicographically.*

▶ **Definition 6.7** (Random minimizer [SWA03]). *The* random minimizer *is the minimizer scheme with a uniform random total order $\mathcal{O}_k$.*

Following [ZKM21a], we also define a *selection* scheme, as opposed to a *sampling* scheme. Note though that this distinction is not usually made in other literature.

▶ **Definition 6.8** (Selection scheme [ZKM21a]). *A* selection scheme *is a sampling scheme with $k = 1$, and thus samples any position in a window of length $w + k - 1 = w$. Like sampling schemes, selection schemes can be either local or forward.*

We will consistently use *select* when $k = 1$, and *sample* when k is arbitrary. When $k = 1$, we also call the sampled position an *anchor*, following bd-anchors [LPS23]. Note that a *minimizer selection scheme* is not considered, as sampling the smallest character can not have density below $1/\sigma$.

▶ **Definition 6.9** (Particular density). *Given a string S of length n, let $W_i \coloneqq S[i..i+\ell)$ for $i \in [n-\ell+1]$. A sampling scheme f then samples the k-mers starting at positions $M \coloneqq \{i + f(W_i) \mid i \in [n-\ell+1]\}$. The* particular density *of f on S is the fraction of sampled k-mers: $|M|/(n-k+1)$.*

▶ **Definition 6.10** (Density). *The* density *of a sampling f is defined as the expected particular density on a string S consisting of i.i.d. random characters of Σ in the limit where $n \to \infty$.*

Since all our schemes must sample at least one k-mer from every w consecutive positions, they naturally have a lower bound on density of $1/w$.

As we will see, for sufficiently large k the density of the random minimizer is $2/(w+1) + o(1/w)$. There is also the notion of *density factor* [MPB+17], which is defined as $(w+1) \cdot d(f)$. Thus, random minimizers have a density factor of 2. While this is convenient, we refrain from using density factors here, because it would be more natural to relate the density to the lower bound of $1/w$ instead, and use $w \cdot d(f)$. Specifically, as defined, the density factor can never reach the natural lower bound of 1, because $(w+1) \cdot \frac{1}{w} = 1 + 1/w > 1$.

Now that we have defined the density, the natural question to ask is:[1]

[1] Definitions and theorems that are newly introduced in this thesis or corresponding papers, as well as related (open) problems are highlighted with a yellow triangle.

▶ **Problem 6.11** (Optimal density). *What is the lowest density that can be achieved by a minimizer, forward, or local scheme?*

Since the classes of forward and local schemes are larger, they can possibly achieve lower densities, but by how much? The ideal is to answer some of these questions by proving a lower bound and providing a scheme that has density equal to this lower bound, ideally for all parameters, but otherwise for a subset. We can also ask what happens when $w \to \infty$ (for k fixed), or when $k \to \infty$ (for w fixed)? And how does this depend on the alphabet size? Or maybe we can not quite make schemes that *exactly* match the lower bound, but we *can* make schemes that are within 1% of the lower bound, or that are asymptotically a factor $1 + o(1)$ away.

There are also different parameter regimes to consider: small $k = 1$ or $k < \log_\sigma(w)$, slightly larger $k \leq 10$, and more practical k up to ≈ 30, or even larger k in theory. Similarly, we can consider small $w \leq 10$, but also $w \approx 1000$ is used in practice. The alphabet size will usually be $\sigma = 4$, but also this can vary and can be $\sigma = 256$ for ASCII input.

If we do find (near) optimal schemes, we would like these to be *pure* in some way: ideally we can provide a simple analysis of their density, as opposed to only being able to compute it without any additional understanding. This somewhat rules out solutions found by brute force approaches, as they often do not provide insight into why they work well. This motivates the following definition.

▶ **Definition 6.12** (Pure sampling scheme). *A sampling scheme is* pure *when it can be implemented in* $O(\text{poly}(w + k))$ *time and space.*

6.6 Computing the density

The density of a sampling scheme is defined as the expected particular density on an infinitely long string. In practice, we can approximate it closely by simply computing the particular density on a sufficiently long random string of, for example, 10 million characters.

The following theorem forms the basis for computing the density of schemes exactly:

▶ **Definition 6.13** ((Charged) context [MPB+17, Lem. 4][ZKM20]). *For forward schemes, a* context *is a string of length $c = w + k$, consisting of two overlapping windows.*

For a sampling scheme f, a context C is charged *when two different positions are sampled from the first and second window, i.e., $f(C[0..w + k - 1)) \neq 1 + f(C[1..w + k))$.*

For a *local* scheme, a context has length $2w + k - 1$ instead [ZKM21a, Section 3.1][KGKM+24, Section 3.2], and is charged when the last window samples a k-mer not sampled by *any* of the previous contained windows. This larger context is necessary because a local scheme can jump backwards. In practice, this

As a small variant on this a *window* is charged when it is the first window to sample a k-mer.

▶ **Theorem 6.14** (Computing density using contexts [SWA03]). *The density of a forward scheme equals the probability that, in a uniform random context of length $c = w + k$, two different k-mers are sampled from the two windows, i.e., the probability that the context is charged.*

Thus, the density can be computed exactly as the fraction of all σ^{w+k} contexts that is charged.

We can also approximate the density by sampling sufficiently many random contexts. A somewhat more efficient method is to use a De Bruijn sequence instead. A De Bruijn sequence of order c is any circular sequence of length σ^c that contains every sequence of length c exactly once [DB46]. We have the following theorem:

▶ **Theorem 6.15** (Computing density using the De Bruijn sequence [MPB+17, Lem. 4]). *The density of any forward scheme equals its particular density on an order $c = w + k$ De Bruijn sequence. For* local *schemes, the order $c = 2w + k - 2$ De Bruijn sequence must be used instead.*

Another approach, that follows from the first, is by considering cycles of length c, rather than just strings of length c.

▶ **Theorem 6.16** (Computing density using cycles). *The density of any forward scheme equals its average particular density over all cyclic strings of order $c = w + k$ for forward schemes and $c = 2w + k - 2$ for local schemes.*

6.7 The density of random minimizers

As a warm-up, we will compute the density of the random minimizer. We mostly follow the presentation of [ZKM20].

We start by analysing when a context is charged [ZKM20, Lemma 1].

▶ **Theorem 6.17** (Charged contexts of minimizers [ZKM20, Lem. 1]). *For a minimizer scheme, a context is charged if and only if the smallest k-mer in the context is either the very first, at position 0, or the very last, at position w.*

Proof. The context contains $w + 1$ k-mers, the first w of which are in the first window, say W, and the last w of which are in the second window, say W'.

When the (leftmost) overall smallest k-mer is either the very first or very last k-mer, the window containing it chooses that k-mer, and the other window must necessarily sample a different k-mer. On the other hand, when the smallest k-mer is not the very first or very last, it is contained in both windows, and both windows will sample it. ◀

Before computing the actual density, we need to bound the probability that a window contains two identical k-mers.

▶ **Theorem 6.18** (Duplicate (k)-mers [ZKM20, Lem. 9]). *For any $\varepsilon > 0$, if $k > (3 + \varepsilon) \log_\sigma(c)$, the probability that a random context of c k-mers contains two identical k-mers is $o(1/c)$.*

Proof (sketch). For any two non-overlapping k-mers in the window, the probability that they are equal is $\sigma^{-k} \leq 1/c^{3+\varepsilon} = o(1/c^3)$. It can be seen that the same holds when two k-mers overlap by $d > 0$ characters.

There are c^2 pairs of k-mers, so by the union bound, the probability that any two k-mers are equal is $o(1/c)$. ◀

In practice, $k > (2 + \varepsilon) \log_\sigma(c)$ seems to be sufficient, but this has not been proven yet. Even stronger, for most applications of the lemma, $k > (1 + \varepsilon) \log_\sigma(c)$ appears sufficient.

This leads us to the density of the random minimizer, which is a more refined version of the simple density of $2/(w + 1)$ that was obtained in [SWA03] and [RHH+04].

▶ **Theorem 6.19** (Random minimizer density [ZKM20, Thm. 3]). *For $k > (3 + \varepsilon) \log_\sigma(w + 1)$, the density of the random minimizer is*

$$\frac{2}{w+1} + o(1/w).$$

Proof. Consider a uniform random context C of $w+k$ characters and $w+1$ k-mers. When all these k-mers are distinct, the smallest one is the first or last with probability $2/(w+1)$. When the k-mers are not all distinct, this happens with probability $o(1/w)$, so that the overall density is bounded by $2/(w+1)+o(1/w)$. ◀

Using a more precise analysis, it can be shown that for sufficiently large k, the random minimizer has, in fact, a density slightly *below* $2/(w+1)$. Marçais et al. [MPB+17] show this using universal hitting sets. Golan and Shur [GS25, Theorem 4] show that the density of the random minimizer is less than 2 for all sufficiently large $k \geq w \geq w_0$, where w_0 is a constant that may depend on the alphabet size σ.

It was originally conjectured that the density of $2/(w+1)$ is the best one can do [SWA03], but this has been refuted by newer methods, starting with DOCKS [OPM+17, MPB+17]. (Although it must be remarked that the original conjecture is for a more restricted class of "local" schemes than as defined here.)

6.8 Universal hitting sets

Universal hitting sets are an alternative way to generate minimizer schemes.

▶ **Definition 6.20** (Universal hitting set [OPM+16, OPM+17]). *A* Universal hitting set *(UHS) U is an "unavoidable" set of k-mers, so that every window of length $\ell = k+w-1$ contains at least one k-mer from the set.*

Universal hitting sets are an example of a *context-free* scheme [Edg21], where each k-mer is sampled only if it is part of the UHS:

▶ **Definition 6.21** (Context free scheme). *A* context-free *scheme decides for each k-mer independently (without surrounding context) whether to sample it or not.*

There is a tight correspondence between universal hitting sets and minimizer schemes:

▶ **Definition 6.22** (Compatible minimizer scheme [MPB+17, Sec. 3.3][MDK18, Sec. 2.1.5][ZKM21a]). *Given a universal hitting set U on k-mers, a* compatible *minimizer scheme uses an order $\mathcal{O}_k$ that orders all elements of U before all elements not in U.*

The density of a compatible minimizer scheme is closely related to the size of the universal hitting set.

▶ **Theorem 6.23** (Compatible minimizer density [MDK18, Lem. 1]). *When a minimizer scheme f is compatible with a UHS U, its density satisfies*

$$d(f) \leq |U|/\sigma^k.$$

Proof (sketch). Consider a De Bruijn sequence of order $c = w+k$. This contains each c-mer exactly once, and each k-mer exactly σ^w times. Thus, the number of k-mers in U in the De Bruijn sequence is $|U| \cdot \sigma^w$.

Suppose the minimizer scheme samples s distinct k-mers in the De Bruijn sequence. Since U is an UHS, $s \leq |U| \cdot \sigma^w$. The density of f is the fraction of sampled k-mers,

$$d(f) = s/\sigma^c \leq |U| \cdot \sigma^w/\sigma^{w+k} = |U|/\sigma^k.$$

◀

From this, it follows that creating smaller universal hitting sets typically leads to better minimizer schemes.

Lastly, Marçais et al. [MPB+17] introduce the *sparsity* of a universal hitting set U as the fraction of contexts of $w+k$ characters that contain exactly one k-mer from U. Then, the density of a corresponding minimizer scheme can be computed as $(1 - \text{Sparsity}(U)) \cdot \frac{2}{w+1}$.

Minimum decycling set. Where a universal hitting set is a set of k-mers such that every length $w+k-1$ window contains a k-mer in the UHS, a *minimum decycling set* (MDS) is a smallest set of k-mers that hits every *infinitely long* string. Equivalently, if we take the complete De Bruijn graph of order k and remove all nodes in the MDS from it, this should leave a graph without cycles. It can be seen that the number of *pure cycles* (corresponding to the rotations of some string of length k) in the De Bruijn graph is a lower bound on the size of an MDS, and indeed this lower bound can be reached.

Mykkeltveit MDS. One construction of an MDS is by Mykkeltveit [Myk72]. To construct this set $\mathcal{D}_k$, k-mers are first embedded into the complex plane via a character-weighted sum of the k k'th roots of unity ω_k: a k-mer K is mapped to $x = \sum_i K_i \cdot \omega_k^i$. This way, shifting a k-mer by one position corresponds to a rotation, followed by the addition or subtraction of a real number. Based on this, $\mathcal{D}_k$ consists of those k-mers whose embedding x corresponds to the first clockwise rotation with positive imaginary part, i.e., such that $\pi - 2\pi/k \leq \arg(x) < \pi$.

6.9 Asymptotic results

In Table 6.1, we summarize a few theoretical results on the asymptotic density of minimizer, forward, and local schemes as $k \to \infty$ or $w \to \infty$. Some of these results will be covered more in-depth later.

Table 6.1 Summary of asymptotic density results. The $k \to \infty$ column shows that the best density in this case is $1/w$. This was previously achieved by the rot-minimizer [MDK18], but the new mod-minimizer converges much faster. When $w \to \infty$, bd-anchors [LPS23] are a close-to-optimal local scheme. The sus-anchors we introduce are *forward* and conjectured to have this same asymptotic density.

Class	$k \to \infty$	$w \to \infty$, lower bound	$w \to \infty$, best
Minimizer	$1/w$, rot-mini	$1/\sigma^k$	$1/\sigma^k$
new	mod-mini		
Forward	$1/w$, rot-mini	$1/w$	$O(\ln(w)/w)$
new	mod-mini	$2/(w+k)$	$(2+o(1))/(w+1)$, conjectured
Local	$1/w$, rot-mini	$1/w$	$(2+o(1))/(w+1)$
new	mod-mini	$1.5/(w+\max(k-2,1))$	

When $k \to \infty$, both the rot-minimizer [MDK18] and the new mod-minimizer (Chapter 8) achieve optimal density $1/w$.

Slightly simplified, the *rot-minimizer* ranks k-mers by the sum of the characters in positions 0 (mod w), so that for $w = 2$, it would sum every other character of the k-mer. Then, it samples the k-mer for which this sum is maximal.

When $w \to \infty$, minimizer schemes have a big limitation. Since they only consider the k-mers, when $w \gg \sigma^k$, almost every window will contain the smallest k-mer. Thus, we obtain:

▶ **Theorem 6.24** (Large-w minimizer scheme [MDK18, Thm. 2]). *For any* minimizer *scheme f, the density is at least $1/\sigma^k$, and converges to this as $w \to \infty$.*

This implies that as $w \to \infty$, fixed-k minimizer schemes can never reach the optimal density of $1/w$. On the other hand, this lower bound does not hold for forward and local schemes. For forward schemes, we can use the lower bound of [KGKM+24, Theorem 1] to get $2/(w+k)$ (Chapter 7). For local schemes, [KGKM+24, Remark 7] applies and with $k' = \max(k, 3)$ we get the bound $1.5/(w + \max(k-2, 1))$.

From the other side, we have:

▶ **Theorem 6.25** (Forward-density for $w \to \infty$ [MDK18, Prop. 7]). *There exists a forward scheme with density $O(1/\sqrt{w})$ for k fixed and $w \to \infty$.*

Proof (sketch). Consider $k' = \log_\sigma \sqrt{w}$. For sufficiently large w we have $k' \geq k$ and we consider any minimizer scheme on k'-mers with window size $w' = w + k - k' \leq w$. Asymptotically, this has density $O(1/\sqrt{w})$. ◀

Later, this was improved to:

▶ **Theorem 6.26** (Forward-density for $w \to \infty$ improved [ZKM21a, Thm. 2]). *There exists a forward scheme with density $O(\ln(w)/w)$ for k fixed and $w \to \infty$.*

Proof (sketch). Let $w' = k' = w/2$, so that $w' + k' - 1 = w - 1 \leq w + k - 1$. We'll build a UHS on k'-mers with window guarantee w'. Set $d = \lfloor \log_\sigma(k'/\ln k')) \rfloor - 1$. Let U be the set of k'-mers that either start with 0^d, or else do not contain 0^d at all. The bulk of the proof goes into showing that this set has size $O(\ln(k')/k') \cdot \sigma^{k'}$. Every string of length $w' + k' - 1 = w - 1$ will either contain 0^d somewhere in its first w' positions, or else the length-$k' = w'$ prefix does not contain 0^d and is in U. Thus, U is a UHS with window guarantee w'. We conclude that the density of a compatible minimizer scheme is bounded by $O(\ln(k')/k') = O(\ln(w)/w)$. ◀

But this is still not optimal: reduced bd-anchors [LPS23, Lemma 6] (Chapter 9) are a *local* scheme with $k = 1$ and density $(2 + o(1))/(w+1)$.

We further improve on this using sus-anchors (Chapter 9), which is a *forward* scheme, we conjecture, also hsa density $(2 + o(1))/(w+1)$ as $w \to \infty$.

While it may seem from Table 6.1 that local schemes are not better than forward schemes, there *are* parameters for which local schemes achieve strictly better density [MDK18, KGKM+24]. Unfortunately, there currently is not good theory of local schemes, and these improved schemes were found by solving an integer linear program (ILP) for small parameters. Lower bounds on local scheme density for small k and w are also not nearly as tight as for forward schemes.

6.10 Variants

There are several variations on sampling schemes that generalize in different ways.

Global schemes drop the requirement that whether a k-mer is sampled only depends on a local context. Examples are minhash [Bro97] and more general FracMinHash [IBR+22], both sampling the smallest k-mers of an entire string.

On strings with many repeated characters, all k-mers have the same hash, and hence all k-mers are sampled. *Robust winnowing* [SWA03] prevents this by sampling the rightmost minimal k-mer by default, unless the minimizer of the previous window has the same hash, in which case that one is "reused".

Min-mers [KGTP23] are a second variant, where instead of choosing a single k-mer from a window, s k-mers are chosen instead, typically from a window that is s times longer.

Finimizers [ABP25] are *variable length* minimizers that ensure that the frequency of the minimizers is below some threshold.

For DNA, it is often not know to which strand a give sequence belongs. Thus, any analysis should be invariant under taking the reverse complement. In this case, *canonical minimizers* can be used.

▶ **Definition 6.27** (Canonical sampling scheme)**.** *A sampling scheme f is* canonical *when for all windows W and their reverse complement* $\mathrm{rc}(W)$*, it holds that*

$$f(\mathrm{rc}(W)) = w - 1 - f(W).$$

One way to turn any minimizer scheme into a canonical minimizer scheme is by using the order $\mathcal{O}_k^{\mathrm{rc}}(x) = \min(\mathcal{O}_k(x), \mathcal{O}_k(\mathrm{rc}(x)))$ [RHH+04, MCVB16] or $\mathcal{O}_k^{\mathrm{rc}}(x) = \mathcal{O}_k(x) + \mathcal{O}_k(\mathrm{rc}(x))$ [KWN+22, GKM25]. Still, this leaves the problem of whether to sample the leftmost or rightmost occurrence of a k-mer in case of ties. This can be solved using the technique of the *refined minimizer* [PR24, GKM25]: ensure that $w + k - 1$ is odd, and pick the strand with the highest count of `GT` bases. Wittler [Wit23] shows a way to encode canonical k-mers that saves one bit. Lastly, Marçais et al. [MEK24] provide a way to turn context-free methods into a canonical version.

The *conservation* of a scheme is the expected fraction of bases covered by sampled k-mers [Edg21]. Shaw and Yu [SY21] generalize sampling schemes to *k-mer selection methods* that allow to sample *any subset* of k-mers from the input string, and *local selection methods* that return any *subset* of k-mers from a window. Both these papers focus on context-free schemes, as such k-mers are better preserved.

There is also the problem to minimize the particular density on a given input string. For example, some works change the order of the `ACGT` DNA alphabet to make `C` the smallest character, as it is often occurs less frequently in DNA sequences [RHH+04]. Other works in this direction are DeepMinimizer [HZK22], minimizers based on polar sets [ZKM21b], and [CLJ+14], the last of which presents a minimizer scheme that orders k-mers such that rare k-mers are preferred over more common ones.

7 Lower Bounds on Sampling Scheme Density

Summary

In this chapter, we look into lower bounds on the density of sampling schemes. We first go over previous lower bounds, including Schleimer et al.'s original one [SWA03] and the "fixed" version by Marçais et al. [MDK18]. Then, we first give a novel near-tight lower bound [KGKM+24], that significantly improves significantly over previous bounds. This new lower bound is the first to show that density $2/(w+1)$ is optimal for $k = 1$. Additionally, an search for optimal schemes for small parameters using integer linear programming (ILP) shows that the lower bound is tight whenever $k \equiv 1 \pmod{w}$.

Attribution

The background in this chapter is newly written, and loosely based on the appendix of the mod-minimizer paper [GKP24] that was coauthored with Giulio Ermanno Pibiri. The proof of the improved version of the lower bound of Marçais et al. is taken from there. The main result of this chapter, a novel near-tight lower bound, is based on the paper "A near-tight lower bound on the density of forward sampling schemes" [KGKM+24], that was written with Bryce Kille and others, and has shared first-authorship between Bryce Kille and myself.

Slightly different versions of the density lower bound were independently discovered by Bryce Kille and myself: I discovered the simpler version for arbitrary w, while Bryce Kille discovered the tighter version for $w = 2$. A first version of the ILP was also implemented by both of us independently, and Bryce Kille optimized this into a faster version.

The starting point for this section is the trivial lower bound:

▶ **Theorem 7.1** (Trivial lower bound). *For any local, forward, or minimizer scheme f, the density is at least $1/w$.*

Naturally, all proofs of tighter lower bounds use the fact that at least one k-mer must be sampled every w positions. All theorems apply it in a slightly different context though.

This was first improved by Schleimer et al. [SWA03] to approximately $1.5/(w+1)$, although using assumptions that are too strong in practice (Section 7.1). Marçais et al. [MDK18] give a weaker version that *does* hold for all forward schemes, of just above $1.5/(w+k)$ (Section 7.2). At the core, it considers two windows spaced apart by $w+k$ positions. The first window than has a minimizer, and with probability $1/2$, a second additional minimizer is needed to 'bridge the gap' to the second window. In the appendix of [GKP24], Groot Koerkamp and Pibiri improve this further to $1.5/(w+k-0.5)$ by using a slightly more precise analysis (Section 7.3). Because of the

similarity of these three proofs, we only provide sketches of the first two, followed by a full proof of the strongest version.

Still, these bounds appeared very far from tight, given that e.g. for $k = 1$ the best schemes do not go below $2/(w+1)$, which is much larger than $1.5/(w+0.5)$. For a large part, Kille and Groot Koerkamp et al. [KGKM+24] resolved this by a new near-tight lower bound of $\lceil (w+k)/k \rceil / (w+k)$ (Section 7.4). This bound looks at cycles of length $w+k$, and uses that at least $\lceil (w+k)/k \rceil$ minimizers must be sampled on such a cycle. They also prove a slightly improved version that is the first lower bound to be *exactly* tight for a subset of parameters.

We end this section with a comparison of the lower bounds to each other, and to optimal schemes found using integer linear programming (Section 7.5).

7.1 Schleimer et al.'s bound

The first improvement over the trivial lower bound was already given in the paper that first introduced minimizers:

▶ **Theorem 7.2** (Lower bound when hashing (k)-mers [SWA03, Thm. 1]). *Consider a w-tuple of uniform random independent hashes of the k-mers in a tuple. Now let S be any function that samples a k-mer based on these w hashes. Then, S has density at least*

$$d(S) \geq \frac{1.5 + \frac{1}{2w}}{w+1}.$$

Proof (sketch). Let W_i and W_{i+w+1} be the windows of w k-mers starting at positions i and $i+w+1$ in a long uniform random string. Since W_i and W_{i+w+1} do not share any k-mers, the hashes of the k-mers in W_i are independent of the hashes of the k-mers in W_{i+w+1}. Now, we can look at the probability distributions X and X' of the sampled position in the two windows. Since the hashes are independent, these distributions are simply the same, $X \sim X'$. There are $(i+w+1+X') - (i+X) - 1 = w + (X'-X)$ "skipped" k-mers between the two sampled k-mers. When $X \leq X'$, this is $\geq w$, which means that at least one additional k-mer must be sampled in this gap. It is easy to see that $\mathbb{P}[X \leq X'] \geq 1/2$, and using Cauchy-Schwartz this can be improved to $\mathbb{P}[X \leq X'] \geq 1/2 + 1/(2w)$. Thus, out of the expected $w+1$ k-mers from position $i+X$ to $i+w+1+X'$ (exclusive), we sample at least $1 + 1/2 + 1/(2w)$ in expectation, giving the result. ◀

Unfortunately, this lower-bound assumes that k-mers are hashed before being processed them further using a potentially "smart" algorithm S. This class of schemes was introduced as *local algorithms*, and thus caused some confusion (see e.g. [MPB+17]) in that it was also believed to be a lower bound on the more general *local schemes* as we defined them. This inconsistency was first noticed by Marçais et al. [MDK18], who introduces a "fixed" version of the theorem.

7.2 Marçais et al.'s bound

Marçais et al. [MDK18] give a weaker variant of the theorem of Schleimer et al. [SWA03] that *does* hold for all forward schemes:

▶ **Theorem 7.3** (Lower bound for forward schemes). *Any forward scheme f has density at least*

$$d(f) \geq \frac{1.5 + \max\left(0, \left\lfloor \frac{k-w}{w} \right\rfloor\right) + \frac{1}{2w}}{w+k}.$$

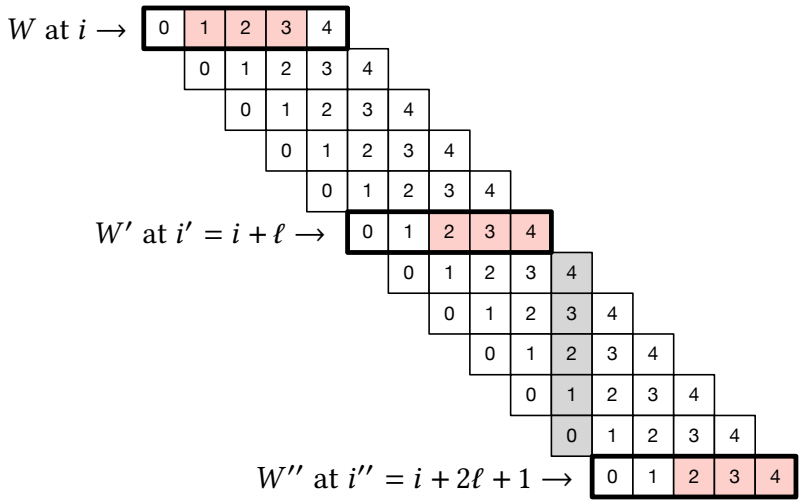


Figure 7.1 Setting used in proving the lower bound. In this example, we use $w = k = 3$, so $\ell = w + k - 1 = 5$. Red boxes indicate the sampled k-mer in windows W, W', and W'' that are highlighted with a ticker stroke.

Proof (sketch). The proof is very comparable to the one of Schleimer et al. [SWA03]. Again, we consider two windows in a long uniform random string. This time, however, we put them $w + k + 1$ positions apart, instead of just $w + 1$. This way, the windows do not share any characters (rather than not sharing any k-mers) and thus, the probability distributions X and X' of the position of the k-mers sampled from W_i and $W_{i+w+k+1}$ are independent again.

They again consider the positions $s_1 = i + X$ and $s_2 = i + w + k + 1 + X'$, and lower bound the expected number of sampled k-mers in this range. The length of the range is $w + k$, leading to the denominator, and the $1.5 + 1/(2w)$ term arises as before. The additional $\left\lfloor \frac{k-w}{w} \right\rfloor$ term arises from the fact that when k is large, just sampling one additional k-mer in between s_1 and s_2 is not sufficient to ensure a sample every w positions. ◀

7.3 Improving and extending Marçais et al.'s bound

It turns out that Marçais et al.'s bound is slightly inefficient. In the appendix of the mod-minimizer paper [GKP24], we improve it. Also note that the existing proof already works for any *local* scheme.

▶ **Theorem 7.4** (Improved lower bound [GKP24]). *The density of any* local *scheme f satisfies*

$$d(f) \geq \frac{1.5}{w + k - 0.5}.$$

Proof. We assume the input is an infinitely long random string S over Σ. The proof makes use of the setting illustrated in Figure 7.1, which is as follows. We partition the windows of S in consecutive groups of $2\ell + 1$ windows. Let one such group of windows start at position i, and consider windows W and W' starting at positions i and $i' := i + \ell$ respectively. Also let W'' be the window that is the exclusive end of the group, thus starting at position $i'' = i + 2\ell + 1 = i' + \ell + 1$. Note that there is no gap between the end of window W and the beginning of window W', whereas there is a gap of a single character between the end of W' and the beginning of W''. (shown as the gray shaded area in Figure 7.1). These three windows are disjoint and hence the random variables

X, X', and X'' indicating $f(W)$, $f(W')$, and $f(W'')$ respectively are independent and identically distributed. (But note that we do not assume they are uniformly distributed, as that depends on the choice of the sampling function f.) In Figure 7.1, we have $X = 1$ and $X' = X'' = 2$.

Since the three windows W, W', and W'' are disjoint, they sample k-mers at distinct positions. (indicated by the red boxes in Figure 7.1). The proof consists in computing a lower bound on the expected number of sampled k-mers in the range $[i + X, i'' + X'')$. Note that for non-forward schemes, it is possible that windows before W or after W'' sample a k-mer inside this range. For our lower bound, we will simply ignore such sampled k-mers.

When $X < X'$, the window starting at $i + X + 1$ ends at $i + X + \ell = i' + X < i' + X'$, thus at least one additional k-mer must be sampled in the windows between W and W'. Similarly, when $X' \leq X''$, the window starting at $i' + X' + 1$ ends at $i' + X' + \ell = i'' + X' - 1 < i'' + X''$, so that at least another k-mer must be sampled in the windows between W' and W''.

Thus, the number of sampled k-mers from position $i + X$ (inclusive) to $i'' + X''$ (exclusive) is at least $1 + \mathbb{P}[X < X'] + 1 + \mathbb{P}[X' \leq X'']$. Since X, X', and X'' are i.i.d., we have that $\mathbb{P}[X' \leq X''] = \mathbb{P}[X' \leq X] = 1 - \mathbb{P}[X < X']$, and hence

$$1 + \mathbb{P}[X < X'] + 1 + \mathbb{P}[X' \leq X''] = 3.$$

Since there are $2\ell + 1$ windows in each group, by linearity of expectation, we obtain density at least

$$\frac{3}{2\ell + 1} = \frac{1.5}{w + k - 0.5}.$$

◀

This new version does not include the $\max\left(0, \left\lfloor \frac{k-w}{w} \right\rfloor\right)$ term, because it turns out that when $k \geq w$, the full bound is anyway less than $1/w$.

In Figure 7.2 we can see that this new version indeed provides a small improvement over the previous lower bound when $k < (w+1)/2$. Nevertheless, a big gap remains between the lower bound and, say, the density of the random minimizer.

It is also clear that this proof is far from tight. It uses that an additional k-mer must be sampled when a full window of $w + k - 1$ characters fits between s_1 and s_2, while in practice an additional k-mer is already needed when the distance between them is larger than w. However, exploiting this turns out to be difficult: we can not assume that the sampled positions in overlapping windows are independent, nor is it easy to analyse a probability such as $\mathbb{P}[X \leq X'' - k]$.

7.4 A near-tight lower bound on the density of forward sampling schemes

Together with Kille et al., we prove a nearly tight lower bound on the density of *forward* schemes. We start with a slightly simplified version.

▶ **Theorem 7.5** (Near-tight lower bound (simple) [KGKM+24]). *Any forward scheme f has a density at least*

$$d(f) \geq \frac{\left\lceil \frac{w+k}{w} \right\rceil}{w + k}.$$

Proof. The density of a forward scheme can be computed as the probability that two consecutive windows in a random length $w + k$ context sample different k-mers [MPB+17, Lemma 4]. From this, it follows that we can also consider *cyclic strings* (cycles) of length $w + k$, and compute the expected number of sampled k-mers along the cycle. The density is then this count divided by $w + k$.

Because of the window guarantee, at least one out of every w k-mers along the length $w+k$ cycle must be sampled. Thus, at least $\lceil (w+k)/w \rceil$ k-mers must be sampled in each cycle. After dividing by the number of k-mers in the cycle, we get the result. ◀

The full and more precise version is as follows.

▶ **Theorem 7.6** (Near-tight lower bound (improved) [KGKM+24, Thm. 1]). *Let $M_\sigma(p)$ count the number of aperiodic necklaces of length p over an alphabet of size σ. Then, the density of any forward sampling scheme f is at least*

$$d(f) \geq g_\sigma(w,k) := \frac{1}{\sigma^{w+k}} \sum_{p \mid (w+k)} M_\sigma(p) \left\lceil \frac{p}{w} \right\rceil \geq \frac{\left\lceil \frac{w+k}{w} \right\rceil}{w+k} \geq \frac{1}{w},$$

where the middle inequality is strict when $w > 1$.

Proof (sketch). The core of this result is to refine the proof given above. While indeed we know that each cycle will have at least $\lceil (w+k)/w \rceil$ sampled k-mers, that lower bound may not be tight. For example, if the cycle consists of only zeros, each window samples position $i + f(000\ldots000)$, so that in the end every position is sampled.

We say that a cycle has *period* p when it consists of $(w+k)/p$ copies of some pattern P of length p, and p is the maximum number for which this holds. In this case, we can consider the cyclic string of P, on which we must sample at least $\lceil p/w \rceil$ k-mers. Thus, at least $\frac{w+k}{p} \left\lceil \frac{p}{w} \right\rceil$ k-mers are sampled in total, corresponding to a particular density along the cycle of at least $\frac{1}{p} \left\lceil \frac{p}{w} \right\rceil$.

Since p is maximal, the pattern P itself must be *aperiodic*. When $M_\sigma(p)$ counts the number of aperiodic cyclic strings of length p, the probability that a uniform random cycle has period p is $p \cdot M_\sigma(p)/\sigma^{w+k}$, where the multiplication by p accounts for the fact that each pattern P gives rise to p equivalent cycles that are simply rotations of each other. Thus, the overall density is simply the sum over all $p \mid (w+k)$:

$$d(f) \geq \sum_{p \mid (w+k)} \frac{p \cdot M_\sigma(p)}{\sigma^{w+k}} \cdot \frac{1}{p} \left\lceil \frac{p}{w} \right\rceil = \frac{1}{\sigma^{w+k}} \sum_{p \mid (w+k)} M_\sigma(p) \left\lceil \frac{p}{w} \right\rceil.$$

The remaining inequalities follow by simple arithmetic. ◀

As can be seen in Figure 7.2, this lower bound jumps up at values $k \equiv 1 \pmod{w}$. In practice, if some density d can be achieved for parameters (w,k), it can also be achieved for any larger $k' \geq k$, by simply ignoring the last $k'-k$ characters of each window. Thus, we can "smoothen" the plot via the following corollary.

▶ **Theorem 7.7** (Near-tight lower bound (monotone)). *Any forward scheme f has density at least*

$$d(f) \geq g'_\sigma(w,k) := \max\left(g_\sigma(w,k), g_\sigma(w,k')\right) \geq \max\left(\frac{1}{w+k}\left\lceil \frac{w+k}{w} \right\rceil, \frac{1}{w+k'}\left\lceil \frac{w+k'}{w} \right\rceil\right),$$

where k' is the smallest integer $\geq k$ such that $k' \equiv 1 \pmod{w}$.

At this point, one might assume that a smooth "continuation" of this bound (that exactly goes through the "peaks") also holds, but this turns out to not be the case, as for example for $\sigma = w = 2$, the optimal scheme exactly follows the lower bound.

Local schemes. The lower bounds discussed so far can also be extended to local schemes by replacing $c = w + k$ by $c = 2w + k - 2$. Sadly, this does not lead to a good bound. In practice, the best local schemes appear to be only marginally better than the best forward schemes, while the currently established theory requires us to increase the context size significantly, thereby making all inequalities much more loose. Specifically, the tightness of the bound is mostly due to the rounding up in $\frac{1}{c}\left\lceil \frac{c}{k} \right\rceil = \frac{1}{w+k}\left\lceil \frac{w+k}{k} \right\rceil$, and the more we increase c, the smaller the effect of the rounding will be.

Searching optimal schemes. For small parameters σ, w, and k, we can search for optimal schemes using an integer linear program (ILP) [KGKM+24]. In short, we define an integer variable $x_W = f(W) \in [w]$ for every window $W \in \sigma^{w+k-1}$, that indicates the position of the k-mer sampled from this window. For each context containing consecutive windows W and W', we add a boolean variable $y_{(W,W')}$ that indicates whether this context is charged. Additionally, we impose that $f(W') \geq f(W) - 1$ to ensure the scheme is forward. The objective is to minimize the number of charged edges, i.e., to minimize the number of y that is true. In practice, the ILP can be sped up by imposing constraints equivalent to our lower bound: for every pure cycle of length at most $w + k$, at least $\lceil (w + k)/w \rceil$ of the contexts must be charged. This helps especially when $k \equiv 1 \pmod{w}$, in which case it turns out that the ILP *always* finds a forward scheme matching the lower bound, and hence can finish quickly.

7.5 Discussion

In Figure 7.2 we compare the lower bounds to optimal schemes for small parameters. First, note that the bound of Marçais et al. (grey) is only better than $1/w$ for small $k < (w + 1)/2$. In this regime, the improved version (green) indeed gives a slight improvement. The simple version of the near-tight bound (blue) is significantly better, and closely approximates the best ILP solutions when at least one of the parameters is large enough. When all parameters are small, the improved version g_σ (purple) is somewhat better though. As discussed, the density can only decrease in k, and indeed the monotone version g'_σ (red) is much better.

We see that the bound exactly matches the best forward scheme when $k = 1$ and the ILP succeeded to find a solution (third column), and more generally when $k \equiv 1 \pmod{w}$. Furthermore, for $\sigma = w = 2$, the lower bound is also optimal for all even k. Thus, we have the following open problem.

▶ **Open Problem 7.8** (Tight lower bound). *Prove that the g'_σ lower bound on forward scheme density is tight when $k \equiv 1 \pmod{w}$, and additionally when $\sigma = w = 2$.*

For the remaining k, specifically $1 < k < w$, there is a gap between the lower bound and the optimal schemes.

▶ **Open Problem 7.9** (Improved lower bound). *Can we find a "clean" proof of a lower bound on forward scheme density that matches the optimal schemes for $1 < k < w$, or more generally when $k \not\equiv 1 \pmod{w}$?*

And lastly, a lot is still unknown about local schemes.

▶ **Open Problem 7.10** (Local scheme density). *In practice, local schemes are only slightly better than forward schemes, while the current best lower-bounds for local schemes are much worse. Can we prove a lower bound that is close to that of forward schemes? Or can we bound the improvement that local schemes can make over forward schemes?*

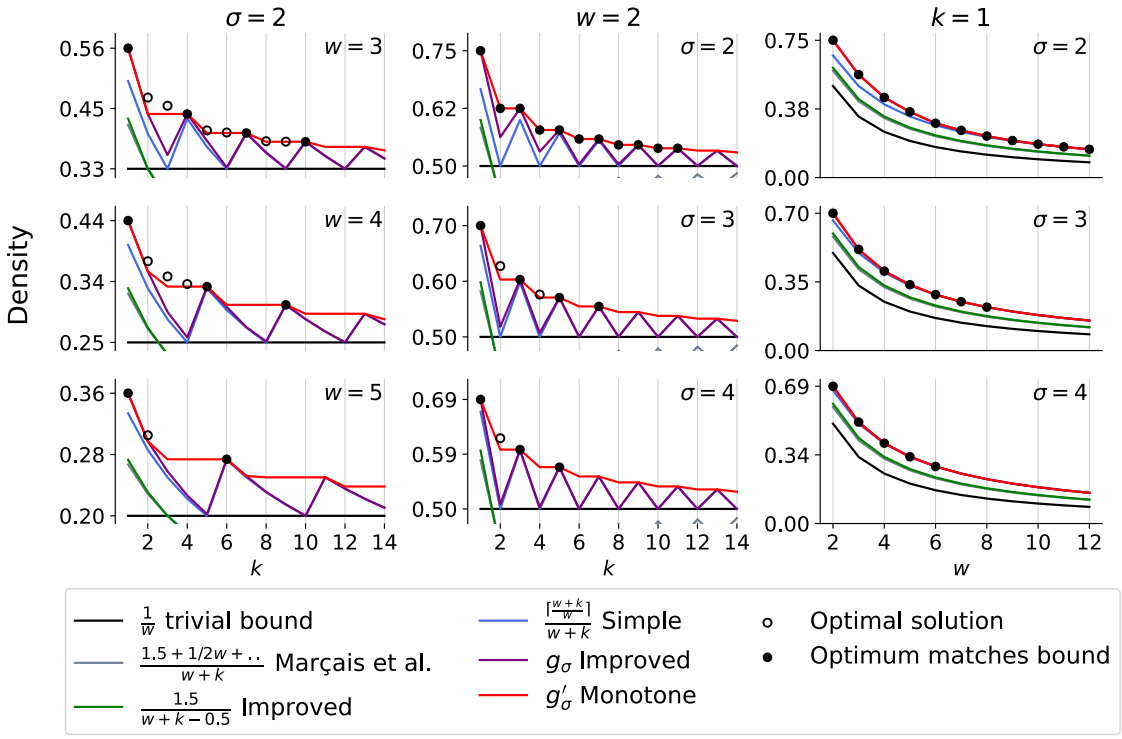


Figure 7.2 Comparison of forward scheme lower bounds and optimal densities for small w, k, and σ. Optimal densities were obtained via ILP and are shown as black circles that are solid when their density matches the lower bound g'_σ, and hollow otherwise. Each column corresponds to a parameter being fixed to the lowest non-trivial value, i.e., $\sigma = 2$ in the first column, $w = 2$ in the second column, and $k = 1$ in the third columns. Note that the x-axis in the third column corresponds to w, not k. This figure appeared before in [KGKM$^+$24] and was made in collaboration with Bryce Kille. The ILP implementation is also his.

Commentary. Bryce Kille and myself independently discovered the basis of the tight lower bound during the summer of 2024. In hindsight, I am very surprised it took this long (over 20 years!) for this theorem to be found. Minimizers were originally defined in 2003-2004, and only in 2018 the first improvement (or fix, rather) of Schleimer et al.'s original bound was found by Marçais et al. [MDK18]. Specifically, all ingredients for the proof have been around for quite some time already:

- The density of the random minimizer is $2/(w+1)$, which "clearly" states: out of every $w+1$ consecutive k-mers, at least 2 must be sampled. We just have to put those characters into a cycle.
- The density of any forward scheme can be computed using an order $w+k$ De Bruijn sequence, so again, it is only natural that looking at strings of length at least $w+k$ is necessary. Cyclic strings are a simple next step.
- And also, partitioning the De Bruijn graph into cycles is something that was done before by Mykkeltveit [Myk72].

8 Practical Sampling Schemes

Summary

In this chapter, we review existing minimizer schemes and more general sampling schemes. They fall into a few categories: they are variants of lexicographic minimizers, based on universal hitting sets with a greedy construction, or based on syncmers.

We then introduce the *open-closed minimizer* [GKLP25], which is a small variant of *miniception* that not only uses *closed syncmers*, but also *open syncmers*. Then, we introduce the *(extended) mod-minimizer* [GKP24], which is a practical minimizer scheme that reaches asymptotic optimal density $1/w$ as $k \to \infty$. For large alphabets, the mod-minimizer exactly matches the density of the lower bound when $k \equiv 1 \pmod{w}$. Together, these make the mod-minimizer the lowest density scheme when $k > w$.

Attribution

This chapter is based on two papers. "The mod-minimizer: A simple and efficient sampling algorithm for long k-mers" [GKP24] is coauthored with Giulio Ermanno Pibiri and introduces the mod minimizer. The followup paper "The open-closed mod-minimizer algorithm" [GKLP25] was written with Giulio Ermanno Pibiri and Daniel Liu and introduces the open-closed minimizer and the extended mod-minimizer.

The idea for the mod-minimizer is my own. The open-closed minimizer was found in close collaboration with Daniel Liu and Giulio Ermanno Pibiri. For both papers, the implementation, evaluation, and writing of the paper were done in equal parts by Giulio Ermanno Pibiri and myself.

We now turn our attention from lower bounds and towards low-density sampling schemes. We first consider various existing schemes, that we go over in three groups. In Section Section 8.1 we consider some simple variants of lexicographic minimizers. In Section Section 8.2, we consider some schemes that build on universal hitting sets, either by explicitly constructing one or by using related theory. We also include here the greedy minimizer, which is also explicitly constructed using a brute force search. Then, in Section Section 8.3, we cover some schemes based on syncmers.

We end with two new schemes. First, the *open-closed minimizer* [GKLP25] (Section Section 8.4), which extends the miniception by first preferring the smallest open syncmer, falling back to the smallest closed syncmer, and then falling back to the smallest k-mer overall. This simple scheme achieves near-best density for $k \leq w$.

Second, we introduce the *(extended) mod-minimizer* and the *open-closed mod-mini* [GKP24, GKLP25]. These schemes significantly improve over all other schemes when $k > w$ and converge to density $1/w$ as $k \to \infty$. Additionally, we show that they have optimal density when $k \equiv 1 \pmod{w}$ and the alphabet is large.

8.1 Variants of lexicographic minimizers

The lexicographic minimizer is known to have relatively bad density because it is prone to sampling multiple consecutive k-mers when there is a run of `A` characters. Nevertheless, they achieve density $O(1/w)$ as $k = \lfloor \log_\sigma(w/2) \rfloor - 2$ and $w \to \infty$ [ZKM20].

They can be improved by using an slightly modified order:

▶ **Definition 8.1** (Alternating order [RHH+04]). *The* alternating order *compares k-mers by by using lexicographic order for characters in* even *positions (starting at position* 0*), and* reverse *lexicographic order for all* odd *positions. Thus, the smallest string is be* `AZAZAZ...`.

This scheme typically avoids sampling long runs of equal characters, unless the entire window consists only of a single character.

A second variant is the *ABB* order.

▶ **Definition 8.2** (ABB order [FNK20]). *The* ABB order *compares the first character lexicographically, and then uses order* `B = C = ... = Z < A` *from the second position onward, so that the smallest string is* `ABBBB...`*, where the number of non-*`A` *characters following the first* `A` *is maximized.*

This scheme has the property that distinct occurrences of the `ABB...` pattern are necessarily disjoint, leading to good spacing of the minimizers. This was already observed before in the context of generating non-overlapping codes [Bla15].

A drawback of the ABB order is that it throws away some information: for example `AB` and `AC` are considered equal, which is usually not idea. Thus, we also consider a version with tiebreaking, *ABB+*:

▶ **Definition 8.3** (ABB+ order). *The* ABB+ *order first compares two k-mers via the ABB order, and in case of a tie, compares them via the plain lexicographic order.*

We also introduce a small variation on these schemes.

▶ **Definition 8.4** (Anti-lexicographic order). *The* anti-lexicographic order *sorts k-mers by comparing the first character lexicographically, and comparing all remaining characters reverse lexicographically.*

In this order, the smallest string is `AZZZZ...`.

8.1.1 Evaluation

In Figure 8.1, we compare the aforementioned variants of lexicographic minimizers. First, note that all schemes perform bad for $k \in \{1, 2\}$, since $k^\sigma \leq 2^4 = 16 < 24 = w$, and thus there will always be duplicate k-mers. As expected, the random minimizer (yellow) has density $0.08 = 2/(24 + 1)$. The lexicographic order (dim blue) is significantly worse at 0.89. The alternating order (orange, 0.78) is slightly better, and the anti-lex order (green, 0.76) is slightly better again.

The ABB order (red-purple, 0.69), and especially the ABB order with tiebreaking (blue) perform *much* better than the random minimizer. ABB+tiebreaking even performs nearly optimal for $3 \leq k \leq 5$. This is surprising, since the idea was already introduced as a sampling and minimizer scheme in 2020 [FNK20, Fig 4a] and appeared more generally before in 2015, but somehow never was compared against by other minimizer schemes. In particular, as we will see later, this scheme outperforms most other schemes for small k, and e.g. miniception (also 2020) is only slightly better for large k.

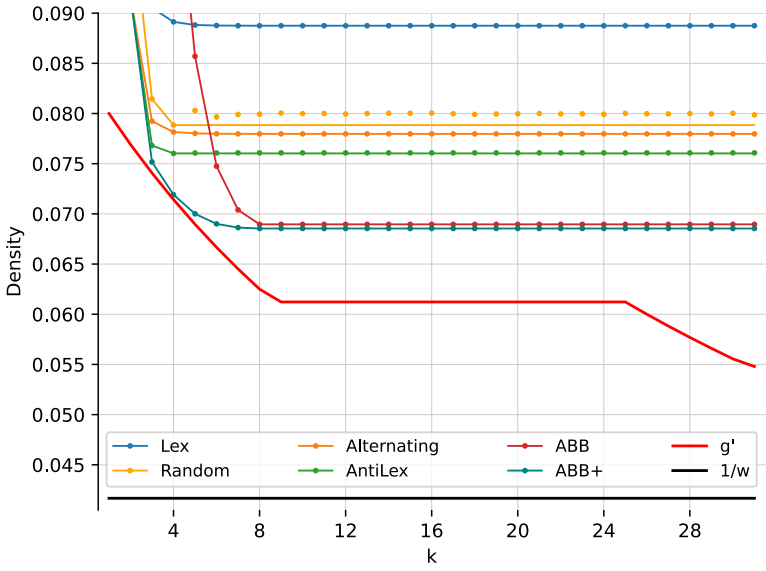


Figure 8.1 Comparison of the density of variants of lexicographic minimizers, for alphabet size $\sigma = 4$, $w = 24$, and varying k. The g' lower bound is shown in red and the trivial $1/w$ lower bound in shown in black. The solid lines indicate the best density up to k, which for the random minimizer happens to be best for $k = 4$ due to the selected random hash function. ABB+ indicates the ABB order with lexicographic tiebreaking.

8.2 UHS-inspired schemes

We first have a look at minimizer schemes that build on universal hitting sets. Of these schemes, all but the decycling minimizer use a bruteforce search to search small universal hitting sets, or minimizers schemes directly for GreedyMini. This means that those schemes are limited to cases where k and w are sufficiently small that this bruteforce search can finish in reasonable time. For DOCKS, **ReM$_u$val**, and PASHA, (double) decycling is better in practice, and thus we omit them from comparisons.

DOCKS. Orenstein et al. [OPM+16, OPM+17] introduce an algorithm to generate small universal hitting sets. It works in two steps. It starts by using Mykkeltveit's minimum decycling set [Myk72] such that every infinitely long string contains a k-mer from the decycling set. Then, it repeatedly adds the k-mer to the UHS that is contained in the largest number of length $\ell = w + k - 1$ windows that does not yet contain a k-mer in the UHS. In practice, the exponential runtime in k and w is a bottleneck. A first speedup is to consider the k-mer contained in the largest number of paths of *any* length. A second method for larger $k' > k$, called *naive extension*, is to simply ignore the last $k' - k$ characters of each k-mer and then use a UHS for k-mers. DOCKS can generate UHSes up to around $k = 13$, and for $k = 10$ and $w = 10$, it has density down to $1.737/(w + 1)$ [MPB+17], thereby being the first scheme that breaks the conjectured $2/(w + 1)$ lower bound.

ReM$_u$val [DGKM19] is a method that builds on DOCKS. Starting with some $(w, k - 1)$ UHS generated by DOCKS, first uses naive extension to get a (w, k) UHS U'. Then, it tries to reduce the size of this new UHS by removing some k-mers. In particular, if a k-mer only ever occurs in

windows together with another k-mer of U', then this k-mer may be removed from U'. Instead of greedily dropping k-mers, and ILP is built to determine an optimal set of k-mers to drop. This process is repeated until the target k is reached, which can be up to at least 200, as long as $w \leq 21$ is sufficiently small.

PASHA [EBO20] also builds on DOCKS and focuses on improving the construction speed. It does this by parallellizing the search for k-mers to add to the UHS, and by adding multiple k-mers at once (each with some probability) rather than only the one with the highest count of un-covered windows containing it. These optimizations enable PASHA to generate schemes up to $k = 16$, while having density comparable to DOCKS.

Decycling-based minimizer. An improvement to the brute force constructions of DOCKS, $\text{ReM}_\text{u}\text{val}$, and PASHA came with a minimizer scheme based directly on minimum decycling sets [PPE+23]: In any window, prefer choosing a k-mer in $\mathcal{D}_k$, if there is one, and break ties using a random order. They also introduce the *double decycling* scheme. This uses the *symmetric* MDS $\tilde{\mathcal{D}}_k$ consisting of those k-mers for which $-2\pi/k \leq \arg(x) < 0$. It then first prefers k-mers in $\mathcal{D}_k$, followed by k-mers in $\tilde{\mathcal{D}}_k$, followed by k-mers that are in neither.

It is easy to detect whether a k-mer is in the MDS or not without any memory, so that this method scales to large k. Surprisingly, not only is this scheme conceptually simpler, but it also has significantly lower density than DOCKS, PASHA, and miniception. Apparently, the simple greedy approach of preferring smaller k-mers works better than the earlier brute force searches for small universal hitting sets. Especially for k just below w, its density is much better than any other scheme.

GreedyMini. Unlike the previous UHS schemes, GreedyMini [GTK+25] directly constructs a low-density minimizer scheme using a brute force approach. As we saw, the density of a minimizer scheme equals the probability that the smallest k-mer in a $w + k$ long context is at the start or end. The GreedyMini builds a minimizer scheme by one-by-one selecting the next-smallest k-mer. It starts with the set of all $w + k$ contexts, and finds the k-mer for which the number of times it appears as the first or last k-mer in a context, as a fraction of its total number of appearances, is lowest. It then discards all contexts this k-mer appears in, and repeats the process until a minimizer has been determined for all contexts. To improve the final density, slightly submoptimal choices are also tried occasionally, and a local search and random restarts are used. To keep the running time manageable, the schemes are only built for a $\sigma = 2$ binary alphabet and up to $k \leq 20$. This is extended to larger k using naive extension and to larger alphabets by simply ignoring some of the input bits.

The resulting schemes achieve density very close to the lower bound, especially when k is around w. In these regions, the greedymini has lower density than all other schemes, and it is able to find optimal schemes for some small cases when $k \equiv 1 \pmod{w}$. This raises the question whether it is also optimal for other k, where the lower bound may not be tight yet. A drawback is that this scheme is not pure: it must explicitly store the chosen order of k-mers. In particular, our choice of $w = 24$ is much larger than the parameters for which precomputed schemes are provided, and so we omit it from the comparison in Figure 8.2.

8.3 Syncmer-based schemes

As we saw, universal hitting sets belong to a more general class of context-free schemes that only look at individual k-mers to decide whether or not to sample them. A well-known category of context-free schemes are *syncmers* [Edg21]. In general, syncmer variants consider the position of the smallest s-mer inside a k-mer, for some $1 \leq s \leq k$ and according to some order $\mathcal{O}_s$. Here we consider two well-known variants: *closed* and *open* syncmers.

▶ **Definition 8.5** (Closed syncmer [Edg21]). *A k-mer is a* closed syncmer *when the (leftmost) smallest contained s-mer according to some order $\mathcal{O}_s$, is either the first s-mer at position 0 or the last s-mer at position $k-s$.*

Closed syncmers satisfy a window guarantee of $k-s$, meaning that there is at least one closed syncmer in any window of $w \geq k-s$ consecutive k-mers. When the order $\mathcal{O}_s$ is random, closed syncmers have a density of $2/(k-s+1)$, which is the same as that of a random minimizer when $k > w$ and $s = k-w$. Indeed, syncmers were designed to improve the *conservation* metric rather than the density. See the paper by Edgar [Edg21] for details.

▶ **Definition 8.6** (Open syncmer [Edg21]). *A k-mer is an* open syncmer *whe the smallest contained s-mer (according to $\mathcal{O}_s$) is at a specific offset $v \in [k-s+1]$. In practice, we always use $v = \lfloor (k-s)/2 \rfloor$.*

The choice of v to be in the middle was shown to be optimal for conservation [SY21]. For this v, open syncmers satisfy a *distance guarantee* (unlike closed syncmers): two consecutive open syncmers are always at least $\lfloor (k-s)/2 \rfloor + 1$ positions apart.

Miniception is a minimizer scheme that builds on top of closed syncmers [ZKM20]. The name stands for "minimizer inception", in that it first uses an order $\mathcal{O}_s$ and then an order $\mathcal{O}_k$.

▶ **Definition 8.7** (Miniception [ZKM20]). *Let w, k, and s be given parameters and $\mathcal{O}_k$ and $\mathcal{O}_s$ be orders. Given a window W of w k-mers,* miniception *samples the smallest closed syncmer if there is one. Otherwise, it samples the smallest k-mer.*

Because of the window guarantee of closed syncmers, miniception *always* samples a closed syncmer when $w \geq k-s$. When k is sufficiently larger than w and $s = k-w+1$, it is shown that miniception has density bounded by $1.67/w + o(1/w)$. In practice, we usually use $s = k-w$ when k is large enough. Unlike UHS-based schemes, miniception does not require large memory, and it is the first such scheme that improves the $2/(w+1)$ density when $k \approx w$.

8.4 Open-closed minimizer

As we saw, Miniception samples the smallest k-mer that is a closed syncmer. The open-closed minimizer is a natural extension of this [GKLP25]:

▶ **Definition 8.8** (Open-closed minimizer [GKLP25]). *Given parameters w, k, and $1 \leq s \leq k$, and orders $\mathcal{O}_k$ and $\mathcal{O}_s$, the open-closed minimizer samples the smallest (by $\mathcal{O}_k$) k-mer in a window that is an open syncmer (by $\mathcal{O}_s$), if there is one. Otherwise, it samples the smallest k-mer that is a closed syncmer. If also no closed syncmer is present, the overall smallest k-mer is sampled.*

The rationale behind this method is that open syncmers have a distance *lower* bound [Edg21], i.e., any two open syncmers are at least $\lfloor (k-s)/2 \rfloor + 1$ positions apart. This is in contrast to closed syncmers, that do not obey a similar guarantee (but instead have an *upper* bound on the

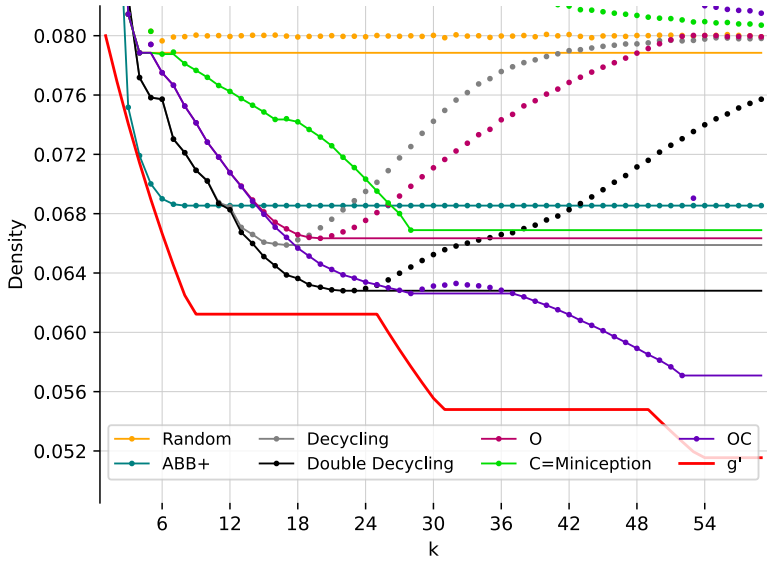


Figure 8.2 Comparison of the density of pure minimizer schemes, for alphabet size $\sigma = 4$, $w = 24$, and varying k. The solid lines indicate the best density up to k. The open-closed minimizer has the OC label, and the O and C labels correspond to preferring open or closed syncmers before falling back to a random order. We use $s = 4$ for the syncmer-based schemes.

distance between them). As it turns out, by looking at Figure 8.2, the distance lower bound of open syncmers (O, red-purple) gives rise to lower densities than the upper bound of closed syncmers (C=miniception, green).

In the full paper [GKLP25], we give a polynomial algorithm to compute the exact density of the open-closed minimizer scheme, assuming that no duplicate k-mers occur. At a high level, this works by considering the position of the smallest s-mer in the window, and then recursing on the prefix or suffix before/after it, until an s-mer is found that is sufficiently far from the boundaries to induce an open syncmer.

8.4.1 Evaluation

In Figure 8.2, we compare all pure schemes seen so far. We see that miniception (green) performs slightly better than the ABB+tiebreak order when k is sufficiently large. The decycling-set based orders (grey and black) significantly outperform the miniception, especially for k just below w. Surprisingly, just changing the closed syncmers of miniception for open syncmers (O, red-purple) yields a better scheme, although not as good as decycling. The combination (OC, purple) does reach the same density as double decycling, and improves over it for $w \leq k \leq 2w$. Interestingly, the O and OC curves look similar to the decycling and double decycling curves, but slightly shifted to the right. The right shift is caused by looking at syncmers where the inner minimizer has length $s = 4$. If we were to use a large alphabet with $s = 1$, this difference mostly disappears.

8.5 Mod-minimizer

So far, all the schemes we have seen in this section work well up to around $k \approx w$, but then fail to further decrease in density as k grows to infinity. The rot-minimizer [MDK18] *does* converge to density $1/w$, but in its original form it only does so very slowly.

Here we present the *mod-minimizer* [GKP24, GKLP25], which turns out to converge to $1/w$ nearly as fast as the lower bound we showed before in Chapter 7.

We start with a slightly more general definition.

▶ **Definition 8.9** (Mod-sampling). *Let W be a window of $w+k-1$ characters, let $1 \le t \le k$ be a parameter, and let $\mathcal{O}_t$ be a total order on t-mers. Let x be the position of the smallest t-mer in the window according to $\mathcal{O}_t$. Then,* mod-sampling *samples the k-mer at position $x \bmod w$.*

As it turns out, this scheme is only forward for some choices of t [GKP24, Lemma 8].

▶ **Theorem 8.10** (Forward). *Mod-sampling is forward if and only if $t \equiv k \pmod w$ or $t \equiv k+1 \pmod w$.*

Proof. Consider two consecutive windows W and W'. Let x be the position of the smallest t-mer in window W and x' that of the smallest t-mer in W'. mod-sampling is forward when $(x \bmod w) - 1 \le (x' \bmod w)$ holds for all x and x'. Given that the two windows are consecutive, this trivially holds when $x = 0$ and when $x' = x - 1$. Thus, the only position x' that could violate the forwardness condition is when W' introduces a new smallest t-mer at position $x' = w + k - t - 1$. In this case, we have $x' \bmod w = (w + k - t - 1) \bmod w = (k - t - 1) \bmod w$. The rightmost possible position of the sampled k-mer in W is $x \bmod w = w - 1$. Hence, if the scheme is forward, then it must hold that $(w-1) - 1 = w - 2 \le (k - t - 1) \bmod w$. Vice versa, if $w - 2 \le (k - t - 1) \bmod w$ always holds true, then the scheme is forward. % since $x \bmod w \le w - 1$.

Now, note that $(k - t - 1) \bmod w \ge w - 2 \iff qw - 2 \le k - t - 1 < qw \iff k - qw \le t \le k - qw + 1$ for some $0 \le q \le \lfloor k/w \rfloor$. In conclusion, the scheme is forward if and only if $t = k - qw$ or $t = k - qw + 1$, i.e., when $t \equiv k \pmod w$ or $t \equiv k+1 \pmod w$. ◀

In Figure 8.3, it can be seen that mod-sampling has local minima in the density when $t \equiv k \pmod w$ [GKP24, Lemma 12], thus, we restrict our attention to this case only.

Furthermore, we can show that for $t \equiv k \pmod w$, mod-sampling is not only forward, but also a minimizer scheme:

▶ **Theorem 8.11** (Minimizer [GKP24, Lem. 13]). *Mod-sampling is a minimizer scheme when $t \equiv k \pmod w$.*

Proof. Our proof strategy explicitly defines an order $\mathcal{O}_k$ and shows that mod-sampling with $t \equiv k \pmod w$ corresponds to a minimizer scheme using $\mathcal{O}_k$, i.e., the k-mer sampled by mod-sampling is the leftmost smallest k-mer according to $\mathcal{O}_k$.

Let $\mathcal{O}_t$ be the order on t-mers used by mod-sampling Define the order $\mathcal{O}_k(K)$ of the k-mer K as the order of its smallest t-mer, chosen among the t-mers occurring in positions that are a multiple of w:

$$\mathcal{O}_k(K) = \min_{i \in \{0, w, 2w, \ldots, k-t\}} \mathcal{O}_t(K[i..i+t))$$

where $k - t$ is indeed a multiple of w since $t \equiv k \pmod w$. Now consider a window W of consecutive k-mers $K_0, \ldots, K_{w-1}$. Since each k-mer starts at a different position in W, $\mathcal{O}_k(K_i)$ considers different sets of positions relative to W than $\mathcal{O}_k(K_j)$ for all $i \neq j$. However, t-mers starting at

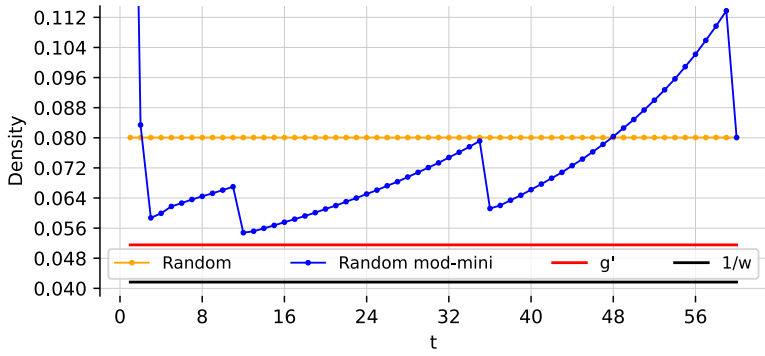


Figure 8.3 The density of the random minimizer and mod-sampling for varying t, for $w = 24$ and $k = 60$. The random mod-minimizer has local minima in the density at $t = 12$ and $t = 36$, where $t \equiv k \equiv 12 \pmod{w}$. There is also a local minimum at $t = 3$, which is the first t that is large enough to avoid duplicate k-mers. Based on this, we choose t to be the least number at least some lower bound r that satisfies $t \equiv k \pmod{w}$. This figure is based on Figure 4 of [GKP24] which was made by Giulio Ermanno Pibiri.

different positions in W could be identical, i.e., the smallest t-mer of K_i could be identical to that of K_j. In case of ties, $\mathcal{O}_k$ considers the k-mer containing the leftmost occurrence of the t-mer to be smaller.

Suppose the leftmost smallest t-mer is at position $x \in [w + k - t]$. Then mod-sampling samples the k-mer K_p at position $p = x \bmod w$. We want to show that K_p is the leftmost smallest k-mer according to $\mathcal{O}_k$. If $\mathcal{O}_t(W[x..x + t)) = o$, then

$$\begin{aligned}\mathcal{O}_k(K_p) = \mathcal{O}_k(W[p..p + k)) &= \min_{j \in \{0,w,2w,\ldots,k-t\}} \mathcal{O}_t(W[p + j..p + j + t)) \\ &= \min_{j \in \{x-p\}} \mathcal{O}_t(W[p + j..p + j + t)) = o.\end{aligned}$$

Since o is minimal, any other k-mer K_j must have order $\geq o$. Also, since o is the order of the leftmost occurrence of the smallest t-mer, K_p is the leftmost smallest k-mer according to $\mathcal{O}_k$. ◀

This now allows us to define the mod-minimizer.

▶ **Definition 8.12** (Random mod-minimizer). *Let $r = O(\log_\sigma(w))$ be a small integer lower bound on t. For any $k \geq r$, choosing $t = r + ((k - r) \bmod w)$ in combination with a uniform random order $\mathcal{O}_t$ gives* the mod-minimizer.

It turns out this definition can be extended to wrap *any* sampling scheme, rather than just random minimizers.

▶ **Definition 8.13** (Extended mod-minimizer [GKLP25]). *Let w, k, and $t \equiv k \pmod{w}$ be given parameters, and let $f : \Sigma^{w+k-1} \to [w + k - t]$ be any sampling scheme with parameters $(w', k') = (w + k - t, t)$. Then, given a window W of length $w + k - 1$, the* extended mod-minimizer *of f samples position $f(W) \bmod w$.*

Naturally, the extended mod-minimizer can be applied to the open-closed minimizer, to obtain the oc-mod-mini.

8.5.1 Theoretical density

When we restrict f to be a *minimizer* scheme specifically, we can compute the density of the extended mod-minimizer.

▶ **Theorem 8.14** (Extended mod-minimizer density [GKLP25, Thm. 1]). *Let w, k, and $t \equiv k \pmod w$ be given parameters, and let f be a* minimizer *scheme on t-mers with order $\mathcal{O}_t$. Then, the density of the extended mod-minimizer is given by the probability that, in a context of length $w+k$, the smallest t-mer is at a position $0 \pmod w$.*

Proof. Consider two consecutive windows W and W' of length $w+k-1$ of a uniform random string. Let x and x' be the position of the smallest t-mer in W and W' respectively, and let $p = x \bmod w$ and $p' = x' \bmod w$ be the positions of the sampled k-mers Let $y \in \{x, x'+1\}$ be the absolute position of the smallest t-mer in the two windows.

Since A is a forward scheme, we can compute its density as the probability that a different k-mer is sampled from W and W'. First note that the two consecutive windows contain a total of $w+k-t+1$ t-mers, and thus, $0 \le y \le w+k-t$, where $w+k-t$ is divisible by w since $t \equiv k \pmod w$.

When $y \not\equiv 0 \pmod w$, this implies $0 < y < w+k-t$, and thus, the two windows share their smallest t-mer. Thus, $p = x \bmod w = y \bmod w$ and $p'+1 = x' \bmod w + 1 = (y-1) \bmod w + 1$. Since $y \not\equiv 0 \pmod w$, this gives $p'+1 = y \bmod w$, and thus, the two windows sample the same k-mer.

When $y \equiv 0 \pmod w$, there are two cases. When $y = x$ (and thus $y < w+k-t$), we have $p = x \bmod w = y \bmod w = 0$, and since the k-mer starting at position 0 is not part of W', the second window must necessarily sample a new k-mer. Otherwise, we must have $y = (x'+1) \equiv 0 \pmod w$, which implies $p' = x' \bmod w = (y-1) \bmod w = w-1$, and since the k-mer starting at position $w-1$ in W' is not part of W, again the second window must necessarily sample a new k-mer.

To conclude, the two windows sample distinct k-mer if and only if the smallest t-mer occurs in a position $y \equiv 0 \pmod w$. ◀

Before we compute the density of the mod-minimizer, we first re-state Lemma 9 of [GKP24], which is a slightly modified version of Lemma 9 of [ZKM20] that we saw earlier in Section 6.7. The proof is nearly identical.

▶ **Theorem 8.15** (Duplicate (k)-mers [GKP24, Lem. 9]). *For any $\varepsilon > 0$, if $t > (3+\varepsilon)\log_\sigma(\ell)$, the probability that a random window of $\ell - t + 1$ t-mers contains two identical t-mers is $o(1/\ell)$. Given that $\ell = w+k-1$, $o(1/\ell) \to 0$ for $k \to \infty$.*

From the above two results, we obtain the density of the random mod-minimizer.

▶ **Theorem 8.16** (Random mod-minimizer density [GKP24, Cor. 17] [GKLP25, Thm. 2]). *If $t \equiv k \pmod w$ satisfies $t > (3+\varepsilon)\log_\sigma(\ell)$ for some $\varepsilon > 0$, the density of the random mod-minimizer is*

$$\frac{2 + \frac{k-t}{w}}{w+k-t+1} + o(1/(w+k-1)).$$

When w is fixed and $k \to \infty$, this density tends to $1/w$.

Proof. Given the bound on t, the probability that a context of $w+k$ characters contains duplicate t-mers is $o(1/\ell) = o(1/(w+k-1))$. Otherwise, the context contains $w+k-t+1$ t-mers, of which the ones at positions $\{0, w, 2w, \ldots, w+k-t\}$ cause the context to be charged, which is a fraction of $\frac{2+\frac{k-t}{w}}{w+k-t+1}$ of all t-mers. ◀

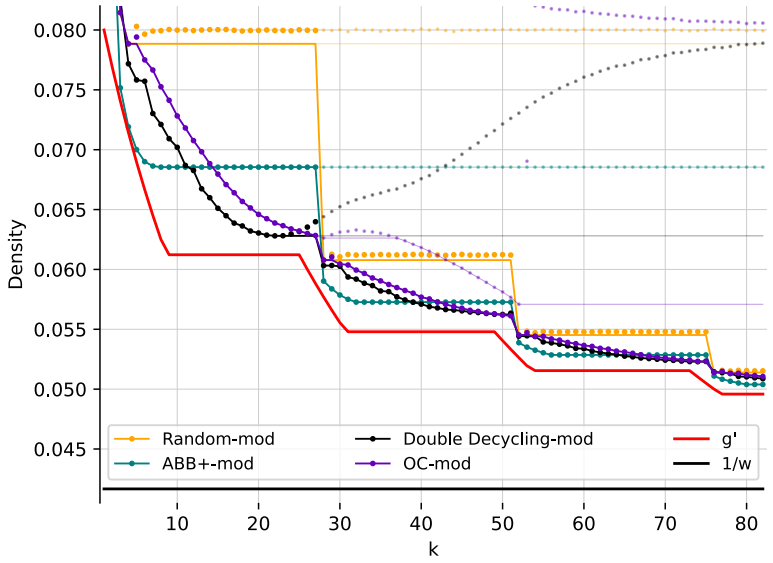


Figure 8.4 Comparison of the density of extended mod-minimizer schemes, for alphabet size $\sigma = 4$, $w = 24$, $r = 4$, and varying k. The solid lines indicate the best density up to k. Versions without mod-mini are shown dimmed.

▶ **Theorem 8.17** (Optimality of the mod-minimizer). *The random mod-minimizer has optimal density when w is fixed, $r = t = 1$, $k \equiv 1 \pmod{w}$, and $\sigma \to \infty$.*

Proof. First note that the probability of duplicate k-mers in a window goes to 0 as $\sigma \to \infty$, and hence the error term in the density computed above disappears. Substituting variables, we get

$$\frac{2 + \lfloor \frac{k-1}{w} \rfloor}{w + \lfloor \frac{k-1}{w} \rfloor w + 1} = \frac{2 + \frac{k-1}{w}}{w + \frac{k-1}{w} w + 1} = \frac{\frac{k+2w-1}{w}}{k + w} = \frac{\lceil \frac{k+w}{w} \rceil}{k + w}.$$

◀

8.5.2 Evaluation

In Figure 8.4, we compare the mod-minimizer version of the best schemes so far against their normal density. We use $r = 4$ as the lower bound on t, so that repeated t-mers are rare in practice. We clearly see that the mod-minimizer schemes roughly follow the red lower bound, and indeed have density that converges to $1/w$ as k increases. Between $k = w + 1$ and $k = 2w + 1$, the graphs look roughly similar to the area between $k = 1$ and $k = w + 1$: ABB with tiebreaking is best when k is small (up to around $w/2$), and hence also when t is small (again up to around $w/2$). For larger k and t, we see that double decycling has minimal density.

Like the OC-minimizer, the mod-minimizer requires $t \geq 4$ to avoid duplicate k-mers. This causes the graphs to jump down at $k = w + 4$, rather than at $k = w + 1$. When the alphabet is large, $t = 1$ suffices, and the graphs of the mod-minimizer touch lower bound when $k \equiv 1 \pmod{w}$.

8.6 Discussion

Looking at Figure 8.4, there are still some regions where we do not yet have near-optimal sampling schemes. First, when $k \leq 2$, all schemes have poor density, as minimizer schemes generally can not do well when $k \leq \log_\sigma(w)$ (Theorem 6.24). In the next section, we will investigate forward schemes that *do* achieve good density for such small k.

Secondly, when $\sigma = 4$ and $k \leq w$ or slightly smaller, the current schemes do not achieve near-optimal density, while we do expect such schemes to exist based on the ILP results for small parameters. Indeed, the greedy minimizer appears to be near-optimal for k close to w. Nevertheless, for roughly $w/6 < k < \frac{2}{3}w$, no schemes are close to the lower bound, and specifically at $k = w/3$, the "deepest" point of the lower bound, the gap is large.

▶ **Open Problem 8.18** (Pure optimal schemes). *Is there a pure forward sampling scheme with density close or equal to the lower bound g'_σ for $k \approx w$ or $k \approx w/3$? Is it possible when $\sigma \to \infty$? Or when $w \to \infty$?*

9 Towards Optimal Selection Schemes

Summary

We end this part on minimizers with a look into *selection* schemes, which are sampling schemes with $k = 1$. These are interesting, because the lower bound appears to be tight: for small parameters, the brute force search finds exactly optimal schemes. Thus, we ask ourselves whether we can construct "pure" optimal schemes that are simple to understand and analyze.

We first review *bidirectional anchors* (bd-anchors) [LP21]. Then, we newly introduce *smallest-unique-substring* anchors, or *sus-anchors*. Initially these improve only slightly over bd-anchors, and by changing from lexicographic order to *anti-lexicographic order*, they become nearly optimal.

We end with a discussion on the remaining gap between our scheme and the true lower bound, and how this could be closed. More generally, we also recap the results on minimizer density we obtained, and state some open problems related to this.

Attribution

This chapter is based on unpublished notes that are my own work. Some of the open problems in the discussion are based on previously written discussions [GKP24, GKLP25, KGKM+24].

As we saw for the sampling schemes so far, they do not do well for very small k. Indeed, they are all minimizer schemes, and these can not achieve density below $1/\sigma^k$. More generally, when $w \to \infty$, *any* fixed finite k at some point becomes less than $\log_\sigma(w)$ and thus too small to result in a good minimizer scheme. At that point, we instead have to consider more general *sampling* schemes to still be able to achieve density close to the lower bound. Here, we take this to the limit and explore the $k = 1$ case, which are also called *selection* schemes. In this case, the window size is $\ell = w + k - 1 = w$, and thus, the task is to select any of the w characters in a window of length w.

We first introduce bidirectional anchors [LP21] in Section 9.1, and then we improve these into smallest-unique-substring anchors in Section 9.2, which is unpublished work of myself.

9.1 Bidirectional anchors

Bidirectional anchors (bd-anchors) are a variant on minimizers that take the minimal lexicographic *rotation* instead of the minimal k-mer substring :

▶ **Definition 9.1** (Bidirectional anchor [LP21]). *Given a window W of length w, the* bidirectional anchor *is the lexicographical minimal rotation of the window, starting at position $i \in [w]$, so that $W[i, w)W[0, i) \leq W[j, w)W[0, j)$ for all j. In case of ties, the leftmost minimal rotation is chosen.*

It turns out that bd-anchors are somewhat brittle because the are not forward: for example, take the window `ZABAAC`, that has minimal rotation starting with `AAC...`. If we then shift the window by one, we may get `ABAACA`, of which the smallest rotation `AAB...` starts at the end. Shifting again to e.g. `BAACAY`, the smallest rotation again starts with `AAC...`. Thus, in the middle step, the final `A` was only sampled because the string happened to start with an `A` as well at that single step. Reduced bd-anchors solve this, although they are still not forward.

▶ **Definition 9.2** (Reduced bidirectional anchor [LPS23, Dfn. 2]). *Given a parameter* $0 \leq r < w$, *the* reduced bidirectional anchor *of a window* W *of length* w *is the lexicographic smallest rotation starting at a position* $i \in [w - r]$.

Now, the following theorem can be proven.

▶ **Theorem 9.3** (Reduced bd-anchor density [LPS23, Lem. 6]). *When* $r = \lceil 4 \log_\sigma w \rceil$, *the reduced bd-anchor has density at most* $2/(w + 1 - r) + o(1/w)$.

Proof (sketch). Because of the choice of r, the probability that the smallest r-mer is not unique is small, and thus, we can simply find the smallest r-mer. Then, the probability that two consecutive windows have a different smallest r-mer is $2/(w + 1 - r)$, similar to the random minimizer. ◀

This parameter r is slightly unfortunate: while plain bd-anchors are parameter-free, the r acts very similar to $k - 1$ for minimizer schemes. And since r is $4 \log_\sigma(w)$, in practice, the reduced bd-anchors usually just sample a length-r lexicographic minimizer. This still has the drawbacks of lexicographic sorting, while not providing the benefit of an actual parameter free scheme.

9.2 Sus-anchors

To avoid the instability of bd-anchors caused by comparing rotations, we can simply only look for the smallest *suffix* instead. A plain version of that would still be unstable, since in e.g. `XABA`, the trailing `A` would compare less than `ABA`, even though it might be followed by a `Z` once the window slides along. To fix this, we prefer longer suffixes over shorter suffixes in case one is a prefix of the other. This leads to the following definition.

▶ **Definition 9.4** (Smallest-unique-substring). *Given a window* W *of length* w, *the* smallest unique substring *anchor (sus-anchor) samples the start position* $i \in [w]$ *of the lexicographic smallest suffix* $W[i, w)$ *that does not occur elsewhere in* W.

The smallest unique substring *is the prefix of the smallest unique* suffix *that does not repeat in* W.

With this new definition and input window `XABA`, we would sample the suffix `ABA`, since the trailing `A` suffix occurs elsewhere. The smallest unique substring is then `AB`, since it only occurs in `XABA` once and is smaller than `XA` and `BA`.

Anti-lexicographic order. One drawback of taking the lexicographic smallest substring is that suffixes of small substrings are also small. In particular, when a window starts with `AAABB...`, both the smallest rotation and the smallest unique suffix are the entire string `AAABB...`, at position 0. After shifting the window one position, we get `AABB...`, and still the bd-anchor and sus-anchor are `AABB...`. If no other `A` occurs at all, after shifting again to `ABB...` again both anchors sample position 0. Thus, neither of these anchors solves the issues associated with lexicographic sorting.

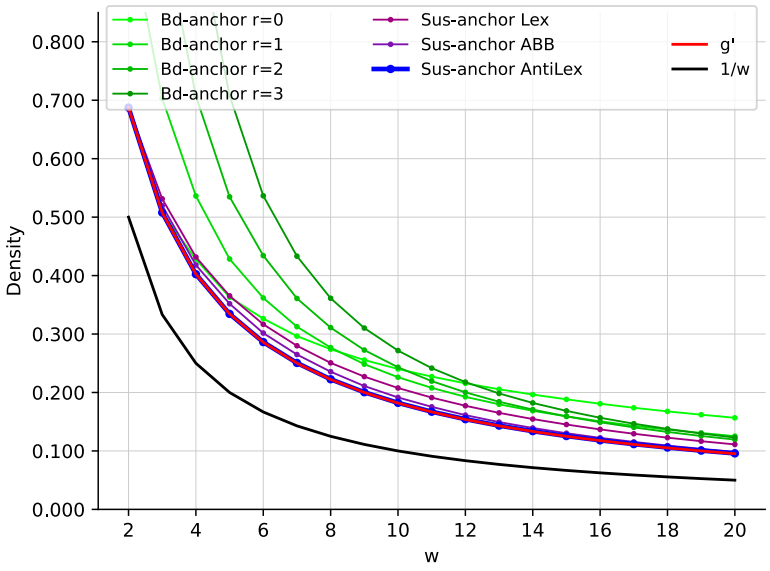


Figure 9.1 Comparison of the density of selection schemes, for alphabet size $\sigma = 4$, $k = 1$, and varying w. Bd-anchors are shown for various r, and the sus-anchor is shown with various underlying orders. The sus-anchor with anti lexicographic order is best, and nearly as good as the lower bound.

To fix this, we consider two variants of sus-anchors: with anti-lexicographic sorting (where `AZZZZ...` is minimal) and with `ABB` sorting, where the smallest string is an `A` followed by any non-`A` characters. Since we only compare suffixes of different lengths, that are thus already distinct, there is no need to use the `ABB+` sorting that includes tie-breaking.

Both these orders avoid the issue of sampling consecutive positions in runs of ~A~s, specifically by preferring a *transition* from `A` to some other character. Further, patterns of `A` followed by non-`A` characters are pairwise disjoint, so that consecutively sampled minimizers are pushed apart from each other.

One of the reasons that this scheme can perform so well for $k = 1$ is that it is not a *minimizer scheme*, and thus not bound by the $1/\sigma^k$ density lower bound. sus-anchors *are* always forward though, regardless of the sort order for each character.

▶ **Theorem 9.5** (Sus-anchors forward). *Sus-anchors are forward when the sort-order is character-by-character.*

We end with the following conjecture.

▶ **Conjecture 9.6** (Sus-anchor density). *Sus-anchors with anti-lexicographic sort order have density* $(2 + o(1))/(w + 1)$.

9.2.1 Evaluation

We compare bd-anchors with various r and sus-anchors with various underlying orderings in Figure 9.1. We see that as w grows, so does the optimal value of r. Still, even with the best r, bd-anchors do not get too close to the density lower bound for $w \leq 20$.

Lexicographic sus-anchors (red-purple) perform around the same or slightly better than bd-anchors, but still do not hit the lower bound. We saw that the ABB order works great as a

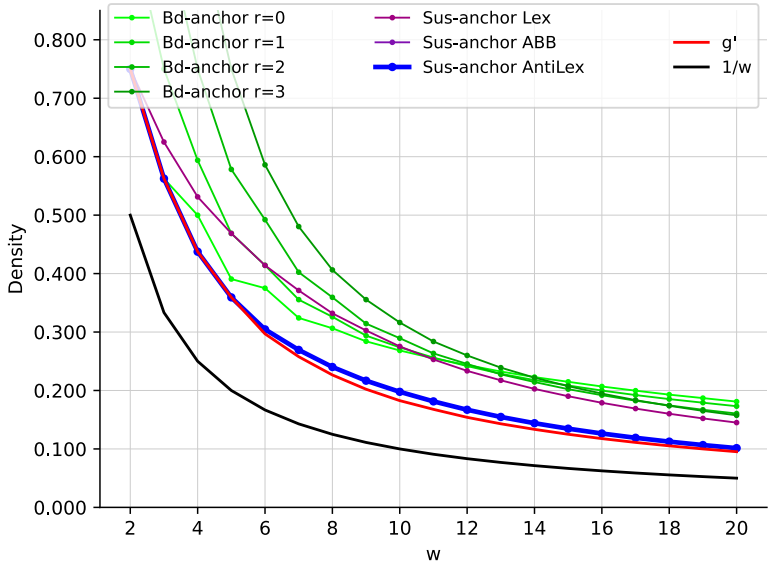


Figure 9.2 Comparison of the density of selection schemes for alphabet size $\sigma = 2$. In this case, ABB and anti-lexicographic order perform the same (ABB is hidden behind the thick blue line), but neither reaches the lower bound for $w \approx 10$.

low-density minimizer scheme, and also here this improves the density. While anti-lexicographic was worse than ABB for minimizers, in this case it turns out to be better, and surprisingly, it reaches density indistinguishable from (but not quite equal to) the lower bound!

If we look at the same plot for alphabet size $\sigma = 2$, Figure 9.2, we see that indeed there is still some room for improvement. In this case, ABB and anti-lexicographic perform exactly the same, since there are only two characters, and hence, sorting by non-A and sorting in reverse are the same.

9.3 Discussion

Selection schemes. As mentioned earlier, ILP is always able to find exactly optimal schemes for small parameters when $k = 1$. The sus-anchor with anti-lexicographic sorting gets very close to optimal for $\sigma = 4$, but still has a little gap when $\sigma = 2$. Thus, the questions remains:

▶ **Open Problem 9.7** (Optimal selection schemes). *Are there pure selection schemes with optimal density* exactly *matching the lower bound g'?*

Sampling schemes. To end, in Figure 9.3 we once more compare all sampling schemes so far. We also include a $k > 1$ variant of the sus-anchor, that simply picks the smallest-unique-substring that has length at least k. Since the sus-anchor works well for small k, we can use $r = 1$ for the extended mod-mini version of it. As can be seen, this is nearly optimal exactly in the small-k region where the ABB+ order was lacking. And the mod-version also works well for $k = w + 1$ and just above. The ABB+ scheme then picks up around $\log_\sigma(w)$, and remains near-optimal until around $w/6$. From then on, double decycling is best up to $k = w$, after which the mod-minimizer versions work best. Looking at the plot, the biggest headroom is currently for intermediate values of k. It is

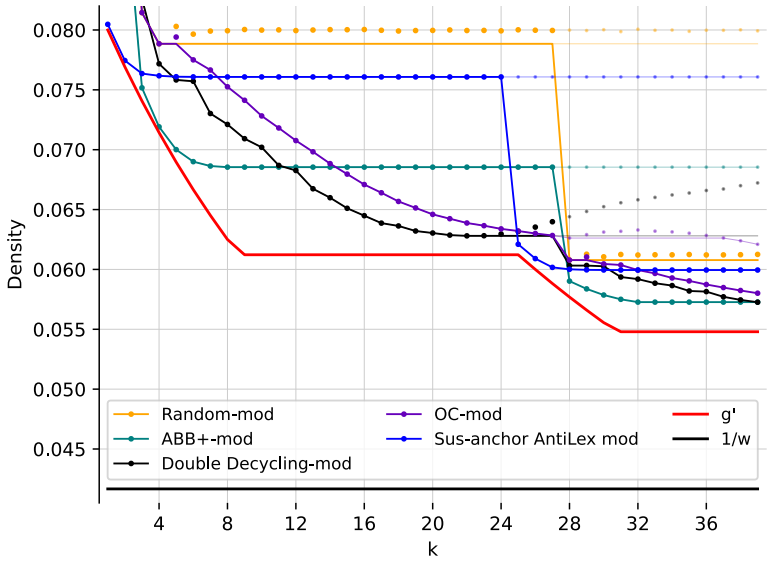


Figure 9.3 Comparison of the density of sampling schemes for alphabet size $\sigma = 4$ and $w = 24$, including sus-anchors. Sus-anchors use $r = 1$ for the mod-minimizer.

unclear whether this is due to the lower bound not being tight, or because better sampling schemes just have not been found yet.

We repeat the open problems that came up so far:

- Can we improve the g' lower bound for $1 < k < w$? And generally for $k \not\equiv 1 \pmod{w}$.
- Can we design sampling schemes that get close to the lower bound for $k \approx w/3$? If not in general, then maybe if $\sigma \to \infty$ or $w \to \infty$?
- Can we prove that the g' lower bound tight for $k \equiv 1 \pmod{w}$? And for $\sigma = w = 2$?
- Can we design an optimal pure selection scheme, having density exactly equal to the g' lower bound.
- How much can local schemes improve over forward schemes? Can we find a near-tight lower bound on their density? Already for $k = 1$, the current bound is only $1.5/(w+1)$, rather than the more probably $2/(w+1)$.

Additionally, there is the question of how much better general forward schemes are compared to minimizer schemes. Of all the schemes in Figure 9.3, only the sus-achor is not a minimizer scheme. So this raises the question: are minimizer schemes as good as forward schemes when k is sufficiently large?

Part III

High Throughput Bioinformatics

10 Optimizing Throughput

Summary

In this chapter, the goal will be *fast code*. Specifically, the goal is to solve some given problem as fast as possible by fully exhausting the hardware. In fact, the goal is not just fast code in itself, but *provably* fast code: ideally we can prove that, given some assumptions, the code is within some percentage of the fastest the given hardware could be.

We start by introducing a few common techniques to achieve speedups. For compute-bound problems, these include SIMD, avoiding unpredictable branches, and instruction-level parallellism, For memory-bound problems, the core principle is to optimize for *throughput* rather than *latency*. Some solutions are interleaving multiple queries using batching or streaming, prefetching, and doing cache line-aware optimization of the memory layout.

We also include some general background on benchmarking and optimizing code.

We show these techniques by using the SSHash k-mer index as an example application. This data structure takes as input a text (without duplicate k-mers), and builds a dictionary on its k-mers. It can then efficiently answer queries whether given k-mers occur in the input text.

The first step is to compute the minimizers of the text. As the number of minimizers is typically much lower than the number of characters in the text, we want this to be fast so that this "compression" step is not the bottleneck.

Then, when answering queries, SSHash uses a *minimal perfect hash function* (MPHF) to find the index of each minimizer in its internal data structure. With PtrHash, we develop an MPHF that only needs a single memory access for most queries, and can answer queries nearly as fast as the memory can support random access reads.

Attribution

In this part, we summarize high level techniques for writing high throughput code. We give examples of how these optimizations are applied in the two upcoming chapters, where we optimize the computation of random minimizers [GKM25] and build a fast *minimal perfect hash function* data structure [GK25].

All text is my own, and at most loosely based on the mentioned papers.

10.1 Introduction

The amount of sequenced biological data is growing exponentially. For a long time, the performance of computer hardware was also growing exponentially, but over the last years this has slowed down. As a result, waiting for faster hardware is not an solution if one wants to process all current data. Thus, there is a need for faster algorithms, and indeed, many papers in bioinformatics use this remark as their motivation.

Complexity. Unfortunately, the performance of a piece of code does not *just* depend on the theoretical runtime complexity of the algorithm used [Med23a]. For example, the *hidden constant* can have a large influence on the actual running time. For example, summing a vector of n integers is $\Theta(n)$, and can be done very efficiently using SIMD instructions at a speed of 0.1 ns per value (or 4 values per CPU cycle). On the other hand, reading n values from random memory addresses is also considered $O(n)$, but this can be as slow as 100 ns per read, which is 1000× slower!

I/O complexity. Thus, the performance of an algorithm also depends on the type of operations it does. One way to capture this is by using the IO model, where the number of memory (or disk) accesses is explicitly tracked. This also somewhat captures the intuitive simplicity of an algorithm: it is well known that dynamic programming algorithms that simply do linear loops over an array are much faster than IO-bound data structures. More generally, this is an instance of the Von Neumann bottleneck [Bac78]: all reading and writing of memory is limited by the memory bus, while simple operations are orders of magnitude faster.

Implementation. Lastly, we come to the most practical part, which is the implementation of the algorithm. While Moore's law [Moo06] may not quite apply anymore, modern CPUs are vastly complex machines, with increasingly more optimizations and heuristics to squeeze everything out of the code they execute. We, the programmer, should be aware of these things, and write our code in such a way that indeed the CPU can execute it efficiently. This also means that we should explicitly use the dedicated instructions that the CPU provides, such as SIMD (*single-instruction multiple-data*) instructions to operate on up to 8 64-bit integers at a time, or the BMI2 (*bit manipulation*) instruction set for efficient bit-wise instructions.

Algorithm design. Taking this one step further, we should not just aim to implement our algorithm of choice efficiently, but we may want to choose and design our algorithm by explicitly keeping in mind the capabilities of the hardware it will run on. Even more, we may want to also change the input format to better suit the algorithm.

10.1.1 Overview

In this chapter, we will go over some techniques to optimize code for modern CPUs. Many works have been written on this topic, such as Ulrich Drepper's book on CPU memory [Dre07], Denis Bakhvalov's book on optimizing performance on modern CPUs [Bak24], and Agner Fog's resources on instruction latency and more [Fog24]. Thus, in this chapter we simply highlight some techniques that are particularly relevant for the following chapters. We group them into two categories.

The first category are optimizations for compute-bound problems. In Section 10.2 (and Chapter 11), we have a look at `simd-minimizers`, which is an efficient implementation of an algorithm to compute random minimizers. We have also already seen a number of compute-bound optimizations being applied to A*PA2 in Chapter 4.

Then, in Section 10.3 (and Chapter 12), we consider a memory-bound application, where the CPU is mostly waiting for data to be read from RAM (or disk): minimal perfect hashing. Here, the problem is to build a data structure that, given a fixed input set S of n elements, efficiently maps each element to a unique integer in $\{0, \ldots, n-1\}$.

Text indexing. We note that both these applications have uses in text indexing. First, minimizers are used in many different data structures. One way in which they are used is to *sketch* the input text into a smaller representation. Then, one can build a much smaller and faster data structure

only on this sketched representation [AFGK+25, GR17, EBC21]. This sketching step can also be seen as a way to compress the data. This means that the compression algorithm itself (the computation of the minimizers) is the only part of the pipeline that sees the full input data, while all subsequent steps only work on the sketched representation. This means that as the compression factor increases (for example, because genomic reads become more accurate), the proportion of time spent on the compression increases, and indeed, this can take a significant portion of the time. Thus, we design an optimized implementation to compute random minimizers.

A second application of minimizers is to cluster the k-mers of a text, where k-mers that share the same minimizer are mapped to the same bucket. This is used, for example, by the GGCAT De Bruijn graph construction algorithm [CT23] in order to build disjoint pieces of the graph in parallel, and a similar technique is used by k-mer counting methods such as KMC2 [DKGDG15].

The same technique is also used by SSHash [Pib22], which is an efficient representation of a static set of k-mers. Again, each k-mer is first mapped to its minimizer. It then efficiently stores buckets of k-mers that share the same minimizer via super-k-mers, which are longer strings containing multiple adjacent k-mers as substrings. Once the data structure is built, a critical step is to efficiently retrieve the bucket that corresponds to a minimizer, which is done by building a *minimal perfect hash function*. Since a data structure implementing such a hash naturally takes quite some space, queries usually hit the main memory, and thus this is a memory bound problem.

SSHash originally uses PTHash [PT21], and in Chapter 12, we build on this to develop PtrHash by applying the techniques from this chapter to optimize its throughput.

Throughput, not latency. We end here with one more remark. Many memory-bound applications are in fact bound by the memory *latency*. For example, this means that a piece of data is requested from RAM, and then the CPU has to wait for this data to become available before further progress can be made. This means that for (up to) the entire duration of the request, which can take 80 ns, the CPU is waiting for one bit of data. At the same time, the memory can handle many more reads than only one every 80 ns, and thus, the memory bandwidth is also not exhausted.

We argue that in many bioinformatics applications, sequences are processed in a relatively homogeneous way, where for example the same function is applied to every k-mer. This means that multiple k-mers are processed independently. If every k-mer requires read from memory, we can then process those in parallel.

Currently, not many applications are written in this way, and thus, there is a lot of room for improvement.

10.2 Optimizing Compute Bound Code: Random Minimizers

We start with an overview of techniques that can be used to optimize compute-bound code.

As an example application, we apply these techniques to the problem of efficiently computing the minimizers of a sequence. There are many indices and tools using minimizers, such as SSHash [Pib22] and minimizer-space De Bruijn graphs [EBC21]. In some cases, minimizers are also specifically used as a sketch of the text [GR17, AFGK+25]. Specifically there, this "compression" step of computing the minimizers can easily become a bottleneck, since all subsequent operations only have to operate on the much smaller sketched space. Thus, this is a classic compute bound problem, where the input is a DNA sequence, and the output is the set of minimizer positions or kmers.

A*PA2. Most of the techniques mentioned below are also already used in the A*PA2 pairwise aligner (Chapter 4), which is also compute bound. It processes parts of the DP matrix in large

blocks, so that the execution is very predictable and branch misses are avoided. It also uses SIMD (on top of bitpacking) to compute even more states in parallel, and exploits instruction level parallellism by independently processing two SIMD vectors at a time. It also uses a bit-packed input format to reduce the memory pressure.

10.2.1 Avoiding Branch Misses

Modern CPUs have execution *pipelines* that are hundreds of instructions long Thus, if one instruction is waiting for some data (from memory), the CPU will already start execution upcoming instructions. When a branch occurs, the CPU has to predict which of the two paths will be taken in order to proceed this *speculative execution*, since waiting for the condition to be resolved would remove most of the benefits of pipelining.

Thus, the CPU has a *branch predictor* that fulfils this task. Very much simplified, it tracks for each branch instruction whether it is usually taken or not, and makes a prediction based on this. Modern branch predictors can perfectly recognize patterns like taking a branch every 10th iteration.

When a *branch misprediction* happens, the CPU has to unwind the speculative computations that depended on the wrong assumption, and then start over with the correct sequence of instructions. In practice, this can cause a delay of 10 to 20 clock cycles, and can easily become the bottleneck for performance. Thus, we should aim to design algorithms without *data-dependent branches*, so the branches that remain are all predictable and quick to compute.

Application. For the problem of computing minimizers, we apply this technique by replacing the classic queue based algorithm for minimizers by an efficient version of the *two-stacks* method, that only uses a single branch every w iterations.

10.2.2 SIMD: Processing In Parallel

A common technique to speed up computations on modern hardware is by using SIMD, or single-instruction-multiple-data, instructions. The are for example 256 bit registers that contain four 64 bit integers at once, or eight 32 bit integers. The processor can then do arithmetic on all *lanes* in parallel, providing up to 4× or 8× speedup over scalar arithmetic.

In order to use SIMD instructions, we have to make sure that the input data is sufficiently homogeneous: we need to fill the lanes with integers that require exactly the same computation. And since these computations happen in parallel, they can not depend on each other.

Application. Unfortunately, the problem of computing minimizers is (locally) very sequential, since it requires taking a rolling minimum. To circumvent this, we can split each input sequence into 8 *chunks* that are independent and can be processed in parallel via 256 bit AVX2 SIMD instructions on 8 32 bit lanes.

Because we use a data-independent method to compute the minimizers, the data-flow and executed instructions in each of the 8 chunks are exactly the same. This is the perfect case for SIMD, since there is no *divergence* between the lanes.

10.2.3 Instruction Level Parallelism

Modern CPUs can not only execute many instructions ahead, but they also execute many instructions in parallel. For example, typical Intel CPUs can execute up to 4 instructions each clock cycle. In particular in very simple for loops, e.g., that sum the values of an array, there is a *loop carried*

dependency, and each iteration depends on the previous one. Thus only one addition can be executed at a time, so that the CPU is not fully utilized.

One way to increase the amount of parallelism available in the code is by solving two instances in parallel. For example, to sum the integers in a vector, we can split it in two halves (or even four quarters!) and sum them at the same time.

Application. We tried to apply this to the computation of minimizers by splitting the input into 16 chunks, and then running two instances of the 8-lane algorithm interleaved. In this case, the gains were marginal. Probably the additional instructions increase the load on the hardware registers too much.

10.2.4 Input Format

Lastly, also the input format and more generally memory IO can have a big impact on performance, since highly optimized code usually processes a lot of data.

Specifically, the SIMD `scatter` instruction, that reads 8 arbitrary addresses, and `gather` instruction, that writes to 8 arbitrary addresses, are often slow. More generally, any kind of shuffling data, either by writing spread out over memory or by reading from random parts of memory, tends to be much slower than simply sequentially iterating over some input.

Application. The input for the SIMD version of our minimizer algorithm is 8 streams of text, that are initially encoded as plain 8 bit ASCII characters. Thus, while we could read one character from each stream at a time, it is much more efficient to `gather` 8 32 bit integers at once, each containing 4 characters. In practice, it is better to read a full 64 bit integer at a time, rather than splitting this into 2 32 bit reads.

Still that is not maximally efficient. For DNA, each ASCII character can only really be one of four values, ACGT. Thus, each 8 bit character has 6 wasted bits. We can avoid this by first *packing* the input in a separate linear pass. Then, the algorithm itself can read 64 bits at a time from each lane, containing 32 characters.

10.3 Optimizing Memory Bound Code: Minimal Perfect Hashing

We now consider techniques for optimizing memory bound code.

As an application, we consider the *minimal perfect hash function* in SSHash. SSHash first collects all minimizers, and then builds a hash table on these minimizers as a part of its data structure. Building a classic hash table that stores the values of the keys is possible, but this would take a lot of space, since it has to store all the keys. Instead, we can use the fact that the data structure is *static*: the set of m minimizers is fixed. Thus, we can build a *minimal perfect hash function* (MPHF) that takes this set, and bijectively maps them to the range $\{0, \ldots, m-1\}$. Then, queries can use this function to find the right slot in an array storing additional data for each minimizer.

We focus on designing an MPHF that can answer queries quickly. Specifically, we optimize for throughput, i.e., to answer as many independent queries per second as possible. When the number of keys (minimizers) is large, say 10^9, the MPHF data structure will not fit in L3 cache, and hence, most of the queries will need to access main memory. Thus, like most data structures, this problem is memory-bound.

We note that code can be memory bound in two ways: by memory *latency*, where it is usually waiting for one read to come through, or by memory *throughput*, where the entire bandwidth is saturated. We should avoid being bound by latency, and instead aim to get as much work done as possible given the available throughput.

10.3.1 Using Less Memory

A first way to reduce a memory latency or throughput bottleneck is by simply using less memory. CPUs have a hierarchy of caches, typically with L1, L2, and L3 cache, with L1 being the closest to the CPU and hence fastest, but also the smallest. This means that if the data fits in L1, random accesses to it will be significantly faster (a few cycles) than for data that only fits in L2 (around 10 cycles), L3 (around 40 cycles), or main memory (up to 200 cycles). Thus, smaller data fits in a smaller cache, and hence will have faster accesses. Even when the data is much larger than L3, reducing its size can still help, because then, a larger fraction of it can be cached in L3.

One way to apply this is by reducing the size of integers from 64 bits to 32 bits, when this is still sufficiently large to hold the data.

10.3.2 Reducing Memory Accesses

A first step to reduce the memory bottleneck is by avoiding memory access as much as possible. Completely removing a dependency on some data is usually not possible, but instead, it is often possible to organize data more efficiently.

In particular, RAM works in units of *cache lines*, which (usually) consist of 64 bytes. Thus, whenever an integer is read from main memory, the entire corresponding cache line must be fetched into the L1 cache. This means that it may be more efficient to store a single *array of structs* rather than a *struct of arrays* if elements of the struct are usually accessed together.

Additionally, one should avoid sequential memory accesses, where the result of memory read determines the location of a second access to memory, since these can not be executed in parallel.

Application. A common application of this technique is in B-trees, which are balanced search trees holding a set of sorted elements. Classic binary search trees have an indirection at every level of the tree. B-trees on the other hand store B values in each node. This reduces the height of the tree from $\log_2(n)$ to $\log_{B+1}(n)$, and efficiently uses a cacheline by reading B values from it at once, rather than just a single value.

Our MPHF, PtrHash, internally uses Elias-Fano (EF) coding [Eli74, Fan71] to compactly encode sequences of integers. We introduce a CacheLineEF version, that overall uses a bit more space, but stores the information to retrieve each value in a single cache line. That way, we can still compress the data, while not paying with more memory accesses.

10.3.3 Interleaving Memory Accesses

As already discussed, CPU pipelines can execute many instructions at the same time. This means that the CPU will already fetch memory for upcoming instructions whenever it can. For example, in a for-loop where each iteration reads a single independent memory address, the CPU can fetch memory a number of iterations ahead.

More precisely, each core in the CPU has a number (12, in case of the hardware used for the experiments in this thesis) of *line fill buffers*. Each time the core requests a new cache line to be read from memory, it reserves one of these buffers so that the result can be stored there when it is

available. Thus, the latency of each individual access can be hidden by doing around 10 reads in parallel. The result is then 10 times higher memory throughput.

One way to achieve this is by clustering independent memory accesses, so that they are automatically executed in parallel. More generally, it can help to have as little code as possible in between consecutive reads, so that the CPU can look relatively more iterations ahead.

10.3.4 Batching, Streaming, and Prefetching

One way to make the interleaving of memory accesses more explicit is by using *batching*. If we have to process n independent iterations of a for loop, and each requires a read to memory, we can group (chunk) them into *batches* of size B, say of size $B = 16$ or $B = 32$. Then, we can first make B reads to memory, and then process the results.

To make this slightly more efficient, *prefetching* can be used, where instead of directly reading the B values into a register, we first ask the CPU to read them into L1 cache using a dedicated prefetch instruction. Then we process the elements in the batch as usual, and all the data should already be present.

A slight variant of this is *streaming*, where instead of processing chunks of size B, we prefetch the data required for the iteration B ahead of the current one.

Application. We apply both batching and streaming in PtrHash, and achieve up to 2× speedup compare to plain for-loops. In particular, using these techniques, each iteration only takes just over 8 ns on average, which on my CPU, is very close to the maximum random memory throughput each core can have.

11 SimdMinimizers: Computing Random Minimizers, *Fast*

Summary

Many tools in bioinformatics use the minimizers of a genomic sequences as a more compact representation. When only few minimizers are sampled and the compression ratio is high, this means that subsequent analysis will be relatively fast. Thus, it is beneficial to develop a fast algorithm to compute the minimizers, since this might be the only part of a method that works on the uncompressed input data. We use this application as an example of optimizing compute-bound code.

The `simd-minimizers` library implements an algorithm to compute random minimizers using SIMD instructions. Its main novelty is two-fold. First, it splits the input into 8 chunks that are streamed over in parallel through all steps of the algorithm. This is enabled by using the completely deterministic (data-independent) *two-stacks* sliding window minimum algorithm, which seems not to have been used before for finding minimizers.

The result is that `simd-minimizers` is up to 6.8× faster than a scalar implementation of the *rescan* method when $w = 5$ is small, and 3.4× faster for larger $w = 19$. Computing *canonical* minimizers is only around 50% slower than computing forward minimizers, and around 15× faster than the existing implementation in the `minimizer-iter` crate. Our library finds all (canonical) minimizers of a 3.2Gbp human genome in 5.2 (resp. 6.7) seconds.

Attribution

This chapter is based on "SimdMinimizers: Computing random minimizers, *fast*" [GKM25], which is co-authored with Igor Martayan. Large parts of this chapter are copied verbatim or with minor changes from that publication.

The SimdMinimizers code was largely developed by myself, with contributions from Igor Martayan to support the NEON architecture. The paper itself has equal contribution from both Igor Martayan and myself.

11.1 Introduction

Minimizers were simultaneously introduced by [SWA03] and [RHH+04] as a method to sample short strings of fixed length k, called k-mers or k-grams, for the purpose of fingerprinting and comparing large textual documents such as genomic sequences. This sampling method plays a central role in bioinformatics for the high-throughput analysis of DNA sequencing data and is a fundamental building block for many related tasks such as indexing [Pib22, MKL21], counting [DKGDG15, MRSL25], aligning [Li18, JRH+22], or assembling [EBC21, BRJ+24] genomic sequences.

Minimizers are defined as follows: given a window W of w consecutive k-mers, the minimizer of W is the smallest k-mer according to some order. In practice, this order is often pseudo-random by hashing the k-mers, leading to the *random minimizer*. The *density* of a minimizer scheme is the expected fraction of sampled k-mers on a sufficiently long random string. When k is not too small, random minimizers have an expected density close to $2/(w+1)$, which is around twice as much as the lower bound of $1/w$. In recent years, there have been a number of papers on methods with lower density than random minimizers [ZKM20, PPE+23, GKP24, GKLP25, GTK+25]. While these contributions have narrowed the gap to an optimal-density sampling scheme [KGKM+24], none of them focused on improving the computation time of minimizers.

In bioinformatics applications, the k-mer length is usually at most $k \leq 31$, so that each k-mer can be represented by a single `u64` machine word, and the window length can be as low as 5 (where around a third of the k-mers is sampled), but is typically between 10 and 30. Longer windows of size up to 100 are also possible to index highly conserved texts.

Problem statement. We aim to solve the following problem as fast as possible: given a bitpacked representation of a sequence of ACGT DNA characters, compute the positions of all (canonical) random minimizers.

Contributions. This work introduces a carefully optimized algorithm to compute the minimizers of a genomic sequence. Conceptually, it consists of two parts. First, we introduce a scalar algorithm (Section 11.3) that uses $O(n)$ time and $O(w)$ space. Most of this can be trivially parallelized to $L = 8$ independent lanes with AVX2 or NEON SIMD instructions, and in Section 11.4 we specifically handle the input and output. The parts we discuss are:

1. We apply ntHash [MCVB16, KWN+22], a pseudo-random rolling hash function for k-mers (Section 11.3.2).
2. We compute the sliding-window minima of the hashes using the *two-stacks* method [HST17, TKPP18] (Section 11.3.3).
3. We compute canonical minimizers based on *refined minimizers* [PR24] to decide the *strand* of each window (Section 11.3.4).
4. We extend the scalar algorithm to SIMD by using it on L chunks of the (bitpacked) input sequence in parallel (Section 11.4.1).
5. Lastly, we collect and deduplicate the L parallel streams into unique minimizer positions (Section 11.4.2).

Results. Our method is 3.4× (for large $w = 19$) to 6.8× (for small $w = 5$) times faster than the fastest non-SIMD algorithm for computing forward minimizers. For canonical minimizers, we only compare against a simple implementation and find over 15× speedup. As a result, we can compute the minimizers of a human genome in 5.2 seconds, and the canonical minimizers in 6.7 seconds. We also adapt our method to support generic plain-text ASCII input ($|\Sigma| = 256$), which is slightly (30%) slower due to the larger input characters.

Software. A Rust implementation of our method is publicly available at `https://github.com/rust-seq/simd-minimizers`. The packed sequence representation, the splitting into chunks, and the parallel iteration over their bases is extracted to a separate library, `packed-seq`, available at `https://github.com/rust-seq/packed-seq`.

11.2 Preliminaries

Bitpacking. In this work, we assume that the input sequence is over the DNA alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{T}, \mathsf{G}\}$ and that each letter is encoded using two bits: $\mathsf{A} = \texttt{00}, \mathsf{C} = \texttt{01}, \mathsf{T} = \texttt{10}, \mathsf{G} = \texttt{11}$. This encoding can easily be obtained from the ASCII representation by applying a mask: `(c >> 1) & 3`. Additionally, we assume that the whole sequence is bitpacked using this 2-bit encoding, which can be done as a preprocessing step on the input if necessary. Non-ACGT characters have to be handled during this preprocessing as well, and could be skipped, converted to A, or the input sequence could be split at these points. We assume that the hardware is little-endian, and that the integer value of a sequence $x_0x_1x_2\dots x_{k-1}$ is given by $\sum_{i=0}^{k-1} x_i \cdot 4^i$.

Minimizers. Given parameters w and k, a *window* W of length $\ell = w+k-1$ contains w consecutive k-mers. The *minimizer* of the window is the smallest k-mer in the window. For *random* minimizers, k-mers are ordered by a pseudo-random order, usually given by comparing hashes of the k-mers. In case of ties, the leftmost smallest k-mer is chosen.

Our goal is to compute the absolute position of the minimizer of every window W in the input text. Since adjacent windows often have the same k-mer as minimizer, we only want each position to be listed once in the output.

Canonical minimizers. Because DNA is double-stranded and most sequencing technologies do not distinguish these two strands, genomic sequences have an additional constraint: a sequence and its *reverse-complement* (the reversed sequence of complementary bases $\mathsf{A} \leftrightarrow \mathsf{T}$ and $\mathsf{C} \leftrightarrow \mathsf{G}$) should be considered identical. To satisfy this constraint, *canonical minimizers* should return the same set of k-mers regardless of the strandedness of the input. Specifically, if the canonical minimizer of a window W is at position p, then the canonical minimizer of the reverse-complement $\overline{W}^r$ of W should be at position $|W|-k-p = w-1-p$. In practice, canonical minimizers are often overlooked as an implementation detail and most existing methods simply compare canonical k-mers, computed as $x^c = \min(x, \overline{x}^r)$, which gives a weaker guarantee [MEK24].

11.3 A predictable scalar algorithm

Overview. Computing the minimizers of a sequence generally involves a few steps, as shown in Figure 11.1. First, the k-mers are hashed. We do this using ntHash, a rolling hash (Section 11.3.2). Next, in each window of w k-mer hashes, we must find the position of the leftmost k-mer with the smallest hash. For this, we use a method based on the two-stacks algorithm (Section 11.3.3). Lastly, many adjacent windows will have the same minimizer, and thus these positions must be deduplicated.

In this section, we present a scalar algorithm (that will be trivial to parallelize later by using SIMD instructions Section 11.4), and we also introduce a variant to compute canonical minimizers in Section 11.3.4.

The rolling hash takes constant time per character, and our sliding window algorithm will also take amortized constant time per character, so that the entire method runs in $O(n)$. It needs $O(w)$ space to store the hashes of the current window. With the SIMD optimizations, this improves to $O(n/L)$ time and $O(Lw)$ space, where L is the number of SIMD lanes.

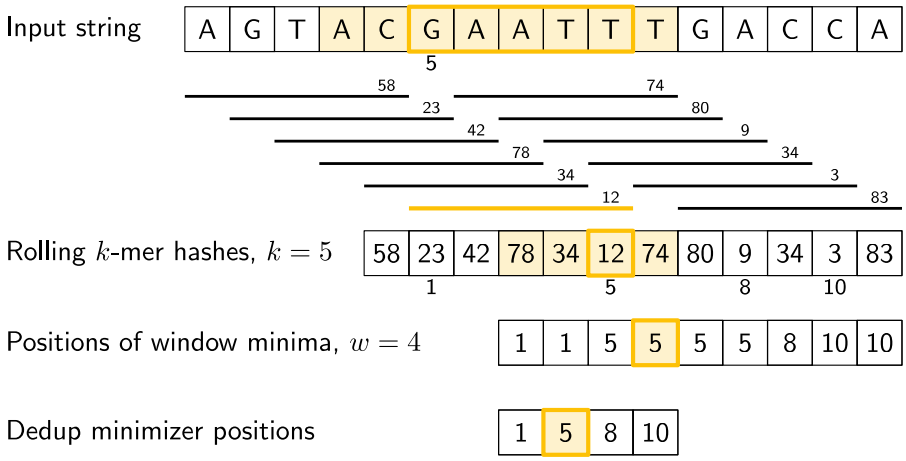


Figure 11.1 Overview of computing minimizers, for $k = 5$ and $w = 4$. First, all k-mers of the input string (shown as black horizontal lines) are hashed (the small numbers above them). Then, for each window of $w = 4$ consecutive k-mers, we find the absolute position of its smallest k-mer. For example, for the first window, the k-mer at position 1 (zero-based) has the smallest hash (23). Similarly, the fourth window (highlighted, spanning $\ell = w + k - 1 = 8$ bases) has minimal hash 12 by the k-mer at position 5. Lastly, these minimizer positions are deduplicated.

11.3.1 Iterating Packed Input

As input to our algorithm, we use a 2-bit packed sequence representation. Our representation is little-endian, in that the two least significant bits of each 8-bit or 32-bit value correspond to the leftmost encoded base.

Extracting bases. To iterate the bases of the input string, we process them one `u32` of 16 bases at a time. For each of those, we do 16 iterations where we extract the 2 least significant bits to return, and then shift the remainder down by 2 bits.

For 8-bit ASCII input, instead, the low 8 bits can be extracted and then shifted away.

Delayed iterator. In order to support a rolling hash, we simply iterate the characters twice in parallel, with an offset: once for the character *entering* each k-mer and once delayed by $k - 1$ iterations for the character *leaving* each k-mer. Similarly, for canonical minimizers we will also need the character leaving the *window*, for which we do a third iteration delayed by $\ell - 1$ steps. To avoid additional reads from memory, we store `u32` values read from memory in a small ring buffer.

11.3.2 Rolling Hash

The first step of the algorithm is the computation of a hash for each k-mer. We adapt ntHash [MCVB16, KWN+22], a popular rolling hash function for DNA sequences that is based on cyclic polynomials [Coh97] as an improvement over Karp-Rabin hashing [KR87]. This rolling hash function can be summarized as follows: each of the 4 bases is associated to a fixed 32-bit random value, denoted $f(x)$, and the hash of a k-mer $u = x_0 \dots x_{k-1}$ is computed as $h(u) = \bigoplus_{i=0}^{k-1} \mathrm{rot}^{k-1-i}(f(x_i))$, where rot^i denotes a cyclic rotation by i bits to the left and $\oplus$ denotes xor. In practice, each x_i is a value in $\{0, 1, 2, 3\}$ and $f(x_i)$ is a simple table lookup, as shown in Algorithm 4.

Algorithm 3 Pseudocode for iterating the packed input sequence, and for iterating over the last (*in*) and first (*out*) bases of all k-mers. The code samples are written in terms of "iterator adapters", where the input is a stream over something, and the output is a transformed stream of *yielded* elements.

```
1:  function IterBp(seq)                          ▷ The bases in seq are bitpacked (4 bases in each byte).
2:      for block: u32 in seq do                  ▷ The text is split into 32-bit blocks. Padding at end not shown.
3:          for i in {0, ..., 15} do
4:              yield block & 3
5:              block ← block » 2

6:  function IterBpDelayed(seq, k)                ▷ Return two streams, one delayed by k − 1 steps.
7:      in_bps ← IterBp(seq)
8:      out_bps ← IterBp(seq)
9:      for in_bp in in_bps[0 : k − 1] do         ▷ [i : j] notation is right-exclusive.
10:         yield (in_bp, _)                      ▷ The first k − 1 iterations, there is out base.
11:     for (in_bp, out_bp) in zip(in_bps[k − 1 :], out_bps) do
12:         yield (in_bp, out_bp)
```

Algorithm 4 NtHash rolling hash. The input is an iterator over pairs of characters coming in and out of each k-mer as returned by IterBpDelayed (Algorithm 3). The output is an iterator over the hashes of all k-mers. The algorithm first processes the first $k-1$ characters to initialize the rolling hash, and then repeatedly adds and removes one character at a time. Each operation can be vectorized to work on L lanes in parallel.

```
1:  function ntHash(k, in_out)                    ▷ in_out iterates pairs (last bp, first bp) of each k-mer.
2:      T_in ← [f(A), f(C), f(T), f(G)]           ▷ Static lookup tables.
3:      T_out ← [rot^{k−1}(f(A)), rot^{k−1}(f(C)), rot^{k−1}(f(T)), rot^{k−1}(f(G))]
4:      h ← 0                                     ▷ 32-bit rolling hash value of (k − 1) bp overlaps.
5:      for (in_bp, _) in in_out[0 : k − 1] do    ▷ The first k − 1 iterations, there is no out_bp.
6:          h ← rot^1(h)                          ▷ In SIMD, implemented as h ← (h ≪ 1 | h ≫ 31).
7:          h ← h ⊕ table_lookup(T_in, in_bp)     ▷ In SIMD, implemented with permute instructions.
8:      for (in_bp, out_bp) in in_out[k − 1 :] do ▷ 2-bit bp coming in and out of each k-mer.
9:          h ← rot^1(h)
10:         h ← h ⊕ table_lookup(T_in, in_bp)
11:         yield h
12:         h ← h ⊕ table_lookup(T_out, out_bp)
```

Using a rolling hash has a twofold advantage in our use case: first, we only need the first and last bases of each k-mer to update its hash based on the previous one, so that we do not have to store the k-mer itself, and second, we are not limited to k-mers that fit in a constant number of words.

Most of the pseudocode in Algorithm 4 can be trivially adapted to work on `u32x8` SIMD registers containing 8 32-bit values. The `table_lookup` function computes the values of $f(x)$ of the L nucleotides stored in `in_bp` or `out_bp` by looking up their value in `table_in` or `table_out`. It can be implemented using `_mm256_permutevar_ps` in AVX2 or `vqtbl1q_u8` in NEON.

MulHash. A drawback of ntHash is that the efficient SIMD table lookup only works because it uses an alphabet of size 4. This means it does not work for general ASCII input. We introduce an alternative that we call *mulHash*, which replaces the table lookup of ntHash by $f(x) = C \cdot x$, where C is a fixed random constant and the multiplication is wrapping over 32-bit integers. Although slightly slower, this can be easily computed for any input value x.

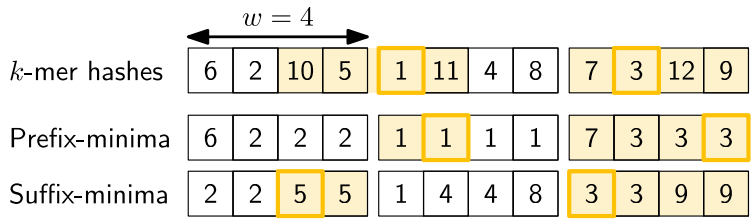


Figure 11.2 An example showing how sliding-window minima can be computed for windows of size $w = 4$. The input k-mer hashes are split into blocks of size w, and for each block, prefix and suffix minima are computed. Then, each window that overlaps two blocks can be split into a suffix and prefix, and the two corresponding minima (highlighted, 5 and 1) can be looked up. The minimum of these two is the minimum of the window. Windows that coincide with a block (as shown on the right) simply use the entire block itself as both prefix and suffix. In practice, we track the position of each minimum alongside its value.

11.3.3 Sliding Window Minimum

The second step of our algorithm computes the position of the minimum k-mer hash in each sliding window of w hashes. A number of different approaches can be used for this, and pseudocode for each of the methods discussed can be found in Algorithm 5.

Naive. The simplest approach is to simply loop over the w values in each window independently. This takes $O(wn)$ time, but can still be quite efficient when w is small by using vectorized instructions.

Monotone queue. An approach with better complexity is to use a *monotone queue*, which stores a non-decreasing subsequence of the w hashes, alongside their positions. Every time the window slides one to the right and we are about to push a new k-mer hash onto the right of the queue, we first remove any values larger than it, as they are "shadowed" by the new hash and can never be minimal anymore. The minimum of the window is then always the leftmost queue element. This data structure guarantees an amortized constant time update, but has many unpredictable branches due to removing between 0 and w values, which makes it costly in practice.

Rescan. Another approach used in bioinformatics is to only keep track of the minimum value and rescan the entire window of w values when the current minimum goes out of scope [Li18, Liu23]. While this algorithm does not guarantee a worst-case constant time update, it only branches when the minimum goes out of scope and hence is more predictable. This makes it more efficient in practice, especially since minimizers typically have a density of $O(1/w)$ so that the $O(w)$ rescan step takes amortized constant time per element.

To the best of our knowledge, most existing methods in bioinformatics use either a monotone queue or a rescan approach [HM20, Li18, Liu23].

Two-stacks. Since our goal here is to compute L minima at the same time using vectorized instructions, we want to avoid any kind of data-dependent branches to ensure that the code path is the same for each chunk. A method for *online* sliding minima where elements may be added and removed at varying rates is the *two-stacks* method [HST17, TKPP18], that is well-known in the competitive programming community[1]. Here, we only discuss a version where the number of

[1] `https://codeforces.com/blog/entry/71687`

Algorithm 5 Scalar implementations of the naive, queue and rescan sliding minimum methods, that return the position of the leftmost minimum value in each window of size w.

```
1: function Naive(w, vals)
2:     for i in {0, ..., |vals| − w} do
3:         yield arg min{vals[i], ..., vals[i + w − 1]}

4: function Queue(w, vals)
5:     queue ← empty DoubleEndedQueue
6:     for i in {0, ..., |vals| − 1} do
7:         if queue.front().pos + w ≤ i then
8:             queue.pop_front()
9:         while queue.back.val > vals[i] do
10:            queue.pop_back()
11:        queue.push_back(val ← vals[i], pos ← i)
12:        if i ≥ w − 1 then
13:            yield queue.front().pos

14: function Rescan(w, vals)
15:    min_val ← +∞
16:    min_pos ← 0
17:    for i in {0, ..., |vals| − 1} do
18:        if vals[i] < min_val then
19:            min_val ← vals[i]
20:            min_pos ← i
21:        if min_pos + w ≤ i then
22:            min_pos ← arg min{vals[i − w + 1], ..., vals[i]}
23:            min_val ← vals[min_pos]
24:        if i ≥ w − 1 then
25:            yield min_pos
```

elements remains constant at w.

Conceptually, we first split the sequence of input k-mer hashes into blocks of size w, as shown in Figure 11.2. Then, we can compute both prefix minima and suffix minima of each block in $O(w)$ per block, or $O(1)$ amortized per input hash. Now, any window of size w can be split into a suffix of the previous block and a prefix of the current block (as highlighted in Figure 11.2), and we can return the minimum of the two corresponding suffix/prefix minima. When the window exactly coincides with a block (as shown on the right), the suffix and prefix minimum are equal.

In the implementation, Algorithm 6, the prefix minima are simply computed incrementally, while the suffix-minima are computed in batches after every block of w hashes has been filled. This way, only a single buffer of size w is needed, and the two cases above are unified.

The only branch in the algorithm triggers exactly every w iterations, and is thus completely data-independent. Thus, branches are both highly predictable, and the same across all L lanes.

16-bit hashes. To easily return the leftmost position of the minimum, instead of the minimum itself, we only use the upper 16 bits of each hash value, and store the position in the lower 16 bits. After taking the minimum, we mask out the high bits to obtain its position. Strings longer than 2^{16} characters can be processed in chunks of 2^{16}. We note here that while using a 16-bit hash is usually not sufficient for k-mer indexing purposes, this *is* sufficiently good for selecting the minimum of a window when $w \ll 2^{16}$, which is indeed the case in bioinformatics applications, where usually

Algorithm 6 Sliding minimum computation using a method based on the *two-stacks* algorithm. The input is an iterator over 32-bit hash values of the k-mers in each of the chunks as returned by ntHash (Algorithm 4), and the output is an iterator over the position of the leftmost minimum hash in each window of w hashes. Each operation can be vectorized to work on L lanes in parallel, each identified by a `LANE_IDX` $\in [0, L-1]$.

```
1:  function SlidingMin(w, hash_it)          ▷ hash_it iterates over 32-bit hashes of each k-mer.
2:      val_mask ← 0xffff0000                ▷ Use high 16 bits of each value.
3:      pos_mask ← 0x0000ffff                ▷ Low 16 bits store positions.
4:      prefix_min ← 0xffffffff              ▷ Rolling prefix min.
5:      suffix_min ← Buffer of size w filled with 0xffffffff
6:      n ← |hash_it| − w − 1                ▷ The first w − 1 hashes do not complete a window.
7:      i ← LANE_IDX × n                     ▷ Current position of the lane.
8:      j ← 0                                ▷ j := (i − LANE_IDX × n) mod w, index in buffer.
9:      for h in hash_it do
10:         val ← (h & val_mask) | i
11:         prefix_min ← min(prefix_min, val)
12:         suffix_min[j] ← val              ▷ Write the current value into the buffer.
13:         j ← j + 1
14:         if j = w then                    ▷ When the buffer is full, recompute suffix minima.
15:             j ← 0
16:             prefix_min ← 0xffffffff      ▷ Reset prefix min.
17:             for k in {w − 2, w − 3, ..., 0} do
18:                 suffix_min[k] ← min(suffix_min[k], suffix_min[k + 1])
19:         if i ≥ LANE_IDX × n + w − 1 then ▷ Skip the first w − 1 incomplete windows.
20:             yield min(prefix_min, suffix_min[j]) & pos_mask   ▷ Position of the min.
21:         i ← i + 1
```

$w \leq 100$. Our library intentionally does not expose this hash to the user. Instead, a second, larger, hash can be computed afterwards for indexing purposes if needed.

11.3.4 Canonical Minimizers

One problem that arises when using minimizers in practice is that the DNA *strand* is often unknown. Thus, we do not know whether we are reading the *forward* (sense) or *reverse-complement* (antisense) strand, where the sequence is reversed and complementary bases A ↔ T and C ↔ G are used. When given a sequence, we would like to select the same minimizers in a strand-agnostic manner.

Canonical strand. Following [PR24], we define the *canonical* strand for each window of length ℓ as the strand where the count of GT bases (encoded values 3 and 2) is the highest, as shown in Algorithm 7. (In fact, any pair of bases can be chosen, as long as they are not complementary.) This count is in $[0, \ell]$ and when ℓ is odd there can be no tie between the two strands. In code, we instead compute the more symmetric $\#\mathsf{GT} - \#\mathsf{AC} = 2\#\mathsf{GT} - \ell$, which is in $[-\ell, \ell]$, so that count > 0 defines the canonical strand. Then, we select the leftmost forward minimizer when the input window is canonical, and the rightmost reverse-complement minimizer when the input window is not canonical.

The benefit of this method over, say, determining the strand via the middle character (assuming again that ℓ is odd), is that the GT count is more stable across consecutive windows, since it varies by ±1. This way, the strandedness and thus the chosen minimizer is less likely to flip between adjacent windows.

Algorithm 7 Counting #GT − #AC. When ℓ is odd, this can never be 0, and a window is canonical when the value is positive. The input is a stream over bases entering and leaving each *window*, where one is delayed by $\ell - 1 = w + k - 2$ steps. Each operation can be vectorized to work on L lanes in parallel.

```
1: function CANONICAL(k, w, in_out)        ▷ in_out iterates pairs (last bp, first bp) of each window.
2:     ℓ ← k − w + 1                       ▷ Window length, it must be odd to guarantee canonicity.
3:     c ← −ℓ          ▷ Count #GT − #AC, starting at −ℓ and adding/subtracting 2 for each G or T.
4:     for (in_bp, _) in in_out[0 : ℓ − 1] do      ▷ The first ℓ − 1 iterations, there is no out_bp.
5:         c ← c + (in_bp & 2)
6:     for (in_bp, out_bp) in in_out[ℓ − 1 :] do   ▷ 2-bit bp coming in and out of each window.
7:         c ← c + (in_bp & 2)
8:         yield c > 0                     ▷ A window is canonical if #GT > #AC.
9:         c ← c − (out_bp & 2)
```

Canonical ntHash. NtHash can be easily modified to compute both the forward and reverse-complement hash of each k-mer at the same time. Then, we can duplicate the sliding minimum algorithm to find the minimum of $(h_{\text{fwd}}(kmer[i]), i)$ and the *maximum* of $(-h_{\text{rc}}(kmer[i]), i)$ over each window. This way, ties in the forward direction are broken towards small i and ties in the reverse-complement direction are broken towards large i. To avoid selecting a new minimizer whenever the strand changes, we use a strand-independent *canonical* version of ntHash for both strands, defined as $h_{\text{c}} = h_{\text{fwd}} + h_{\text{rc}}$ [KWN+22], and we do not distinguish which strand a minimizer k-mer was chosen from.

While this process does not assume k to be odd, this may be a useful additional assumption for further processing of the minimizers, so that the canonical representation of each k-mer itself can be indexed. Alternatively, k-mers could all be processed in the direction of the strand of the window they minimize, but this requires additional bookkeeping to store this direction, and would require duplicating k-mers when they minimize both forward and reverse-complement strands.

Canonical minimizers are not forward. One small drawback of this scheme is that it is not *forward*. Suppose a long window has many occurrences of the smallest minimizer, and that shifting the window one position changes its canonical strand. Then, the position of the sampled minimizer could jump backwards: from sampling the rightmost minimizer to sampling the leftmost minimizer. In practice, this does not seem to be a major limitation, both because it is rare and because downstream methods usually work fine on non-forward schemes anyway.

11.4 A SIMD algorithm

SIMD. So far, our algorithms for iterating the input bases (Algorithm 3), hashing k-mers (Algorithm 4), and computing sliding window minima (Algorithm 6) are completely scalar. Now, we would like to use SIMD instructions to speed them up. With 256-bit AVX2 instructions, for example, we can process $L = 8$ *lanes* of 32-bit values at a time. A first approach could be to use this to compute the hash of L consecutive minimizer values at the same time, and then to compute the minimizer of the L new windows all at once. Unfortunately, this is tricky due to the sequential nature of rolling hashing and sliding window minima.

Chunks. Instead of processing *consecutive* k-mers in parallel, we choose to split the input sequence into L equally long *chunks* that we process in parallel. This way, we compute one hash of each chunk in parallel, and then compute one minimizer position of a window of each chunk in parallel

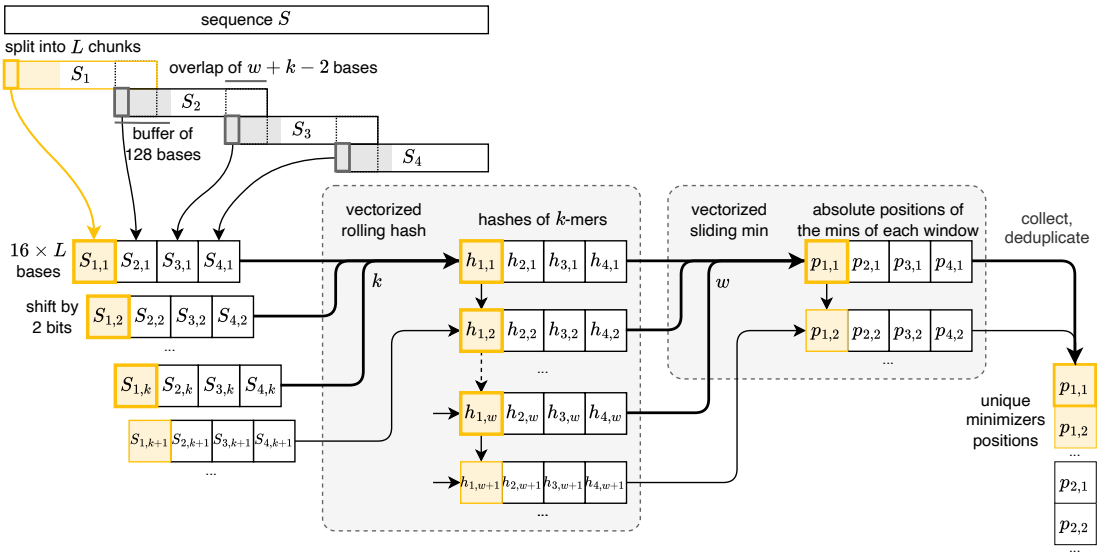


Figure 11.3 High-level view of the vectorized computation of minimizers using the building blocks presented in Section 11.3. The sequence is first split into L chunks (in this example $L = 4$) that are processed in parallel. As described in Section 11.4.1, we load 128 bases at a time for each chunk, from which we extract 16 bases $S_{i,j}$ which are gathered in a SIMD register (one lane for each chunk, 32 bits per lane). This SIMD register is used to iterate over each lane by shifting and masking 2 bits at a time, and is passed as input to a vectorized rolling hash function that computes a hash $h_{i,j}$ for the k-mers in each lane. The absolute position $p_{i,j}$ of the minimum is then computed over a sliding window of w vectors of hashes for each lane. The positions are finally reordered to match the order of the original sequence and deduplicated to keep a unique occurrence of each position. At every step of the computation, the lane corresponding to the first chunk is highlighted in yellow. A bold outline indicates the bases in the first k-mer and the hashes corresponding to the first window.

as well. This works, because the computations on the chunks are completely independent of each other. We let adjacent chunks overlap by $\ell - 1$ characters, so that each window is fully contained in exactly one chunk. The total number of windows may not be divisible by L. In that case, we round up the chunk length and return the number of elements that was added as padding, so that these can be removed later. For simplicity, we omit that case from the code snippets.

11.4.1 Gathering the input

The parts that need special attention for SIMD instructions are the parallel reading of the input sequence, and the parallel deduplicating of the output minimizer positions (Section 11.4.2).

The first step is to read the actual bases. Since memory access instructions are relatively slow[2] compared to the SIMD operations we do on them, we cannot afford to read from each chunk one base or one byte at a time. We could read 32 bits at a time from each chunk using SIMD `gather` instructions, but these are relatively slow. Instead[3], we read a full SIMD register of 256 bits worth

[2] `https://uops.info/html-instr/VPGATHERDD_YMM_VSIB_YMM_YMM.html`
[3] As suggested by one of the anonymous reviewers, for which we thank them.

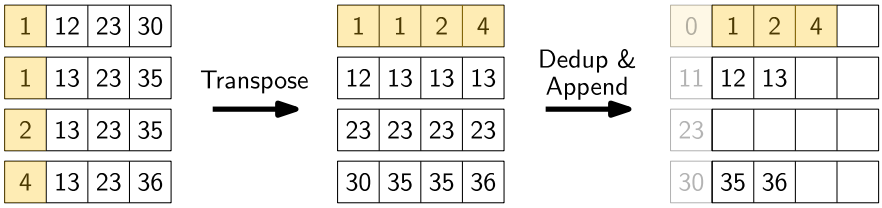


Figure 11.4 The SIMD version of the sliding window minimum algorithm produces minimizer positions for L chunks at a time. For example, the top row on the left indicates a SIMD vector containing the minimizer position of a window in each of the $L = 4$ chunks. In the end, we want to return a single "flat" vector. Thus, we have to "deinterleave" the L lanes. First, we collect L minimizer positions of every chunk (the matrix on the left). Then, we transpose this matrix, so that the resulting SIMD vectors each correspond to a single chunk, with values corresponding to the first chunk highlighted. These are then compared with their preceding element (including the previous minimizer position, as shown in grey), and distinct elements are shuffled to the front. These positions are accumulated in a separate buffer for each chunk, and these buffers are finally concatenated into a single flat vector.

Algorithm 8 Pseudocode for splitting the input sequence into L chunks and iterating each chunk in parallel. The input sequence is assumed to be bitpacked, and indexed by bytes. Yields an iterator over L lanes of `u32` values.

```
1:  function GATHER(k, seq)                          ▷ Chunks overlap by k − 1 bases.
2:      n = |seq| − (k − 1)                          ▷ Number of k-mers. The sequence length is in bases.
3:      b = ⌈n/(4L)⌉                                 ▷ Byte offset between chunks.
4:      B = ⌈(4b + (k − 1))/16⌉                       ▷ Number of u32 covering each chunk.
5:      M ← []                                       ▷ L × L matrix of u32, represented as L u32xL registers.
6:      for i in {0, ..., B − 1} do
7:          if (i mod L) = 0 then
8:              for j in {0, ..., L − 1} do          ▷ Read a full SIMD-vector of data from each chunk.
9:                  M[j] ← seq[jb + 4i : jb + 4i + 4L]   ▷ seq is indexed by bytes.
10:             M ← transpose(M)                     ▷ Transpose the L × L matrix.
11:         yield M[i mod L]                         ▷ Yield one u32 per lane.
```

of data at a time from each chunk, as shown in Algorithm 8. Then, we *transpose*[45] this matrix so that we obtain L SIMD registers, where the first contains the upcoming `u32` for each chunk, the second contains the next `u32` for each lane, and so on. Then, we use this transposed matrix as a buffer, and use it for the next 128 bases. After each 16 bases, we shift to the next `u32x8` from it.

11.4.2 Collecting and Deduplicating Positions

Transpose. The last step of the minimizer algorithm is to deduplicate the results, since adjacent windows often share the same minimizer position. In practice, deduplicating works best when the data to be duplicated is linear in memory. But the output of the vectorized sliding-window-minimum gives a `u32x8` containing the position of one minimizer of each chunk. Thus, every L iterations we reuse the matrix transpose to obtain a `u32x8` for each chunk, containing L consecutive minimizer

[4] `https://stackoverflow.com/questions/25622745/transpose-an-8x8-float-using-avx-avx2`
[5] `https://github.com/rust-seq/simd-minimizers/blob/master/simd-minimizers/src/intrinsics/transpose.rs`

Table 11.1 Total time per base taken when incrementally including more steps of the implementation, for $(w, k) = (11, 21)$.

Part	ns/bp
Iterate the bases	0.15
+ collect to vector	0.30
+ iterate the delayed bases	0.30
+ ntHash	0.32
+ sliding window min	0.90
+ collect	1.48
+ dedup	**1.61**
+ canonical nthash	1.04
+ canonical strand	1.53
+ collect	2.03
+ dedup	**2.20**

Table 11.2 Comparison of our `simd-minimizers` implementation against `minimizer-iter` [Mar] and a rescan implementation based on [Liu23]. Times in ns/bp are shown for both forward and canonical minimizers (where supported), and for various (w, k) tuples. For our library, we test both ntHash and mulHash with multiple encodings of the input DNA: 1) 2-bit packed, 2) ASCII-ACGT that is packed on-the-fly, and 3) plain ASCII (for mulHash only).

(w, k):	$(5, 31)$		$(11, 21)$		$(19, 19)$	
Method	fwd.	cano.	fwd.	cano.	fwd.	cano.
`minimizer-iter`	25.30	32.84	26.96	33.93	26.81	34.04
Rescan ntHash	11.65	-	7.41	-	5.61	-
`simd-minimizers` ntHash						
- packed input	1.69	2.28	1.61	2.20	1.64	2.16
- on-the-fly packing	1.92	2.50	1.84	2.42	1.91	2.42
Rescan mulHash	11.37	-	6.79	-	5.76	-
`simd-minimizers` mulHash						
- packed input	1.85	2.49	1.74	2.40	1.78	2.42
- on-the-fly packing	2.12	2.70	2.05	2.62	2.05	2.65
- ASCII input	2.11	2.71	2.06	2.63	2.01	2.66

positions[6].

Dedup. We deduplicate each lane using the technique of [Lem17]. This compares each element to the previous one, and compares the first element to the last minimizer of the window before. The distinct elements are then *shuffled* to the front of the SIMD vector using a lookup table and appended to a buffer for each lane (Figure 11.4). We end by concatenating all the per-lane buffers into a single vector of minimizer positions, and make sure to avoid duplicates between the end and start of adjacent lanes.

Super-*k*-mers. The deduplication can be amended to also find super-k-mers, which are sequences of consecutive windows sharing the same minimizer position. After comparing adjacent minimizer positions, we obtain a mask that determines the shuffle instruction to apply. Normally we shuffle the 32-bit minimizer positions directly. Instead, we can mask out the upper 16 bits and store there the index of its window. If we then shuffle those values, we obtain for each minimizer its position in the input text, and the position of the first window where this k-mer became a minimizer. This information is sufficient to recover all super-k-mers, and sequences longer than 2^{16} bp can be either split-up, or else it is easy to detect manually when the values wrapped.

11.5 Experimental Evaluation

Our code is available in the `simd-minimizers` crate that can be found at `https://github.com/rust-seq/simd-minimizers`. Part of the code was extracted into a separate library, `packed-seq`, for easy reuse in other projects, which is available at `https://github.com/rust-seq/packed-seq`.

[6] `https://github.com/rust-seq/simd-minimizers/blob/master/simd-minimizers/src/collect.rs`

Both libraries support both AVX2 and NEON instruction sets, and we will now look at their performance. The code to reproduce the experiments is available in the `simd-minimizers-bench` subdirectory.

The experiments were run on an Intel Core i7-10750H with 6 cores with AVX2 running at a fixed frequency of 2.6GHz with hyperthreading disabled and cache sizes of 32 KiB (L1), 256 KiB (L2), and 12 MiB shared L3. Code is compiled with `rustc 1.88.0-nightly`[7]. As input, we use a fixed random string of 10^8 bases, depending on the method encoded either as ASCII or as packed representation. Reported timings are the median of five runs and shown in nanoseconds per base.

In our experiments, we use parameter values for w and k as used by Kraken2 [WLL19] $(5, 31)$, SSHash [Pib22] $(11, 21)$, and Minimap2 [Li18] $(19, 19)$.

Tables 11.3 and 11.4 in Section 11.A show equivalent results for the NEON architecture.

Incremental time usage. Table 11.1 shows the time usage for various incremental subsets of our method. To start, iterating the 8 chunks of the input and summing all bases takes 0.15 ns/bp. Appending all `u32x8` SIMD vectors containing the bases to a vector takes 0.30 ns/bp, indicating that writing to memory induces some overhead. Collecting the second $k-1$-delayed stream of characters that leave the k-mer (by adding them to the non-delayed stream) has no additional overhead. Computing ntHash only takes 0.02ns/bp extra. The sliding window computation nearly triples the total time. Collecting the minimizer positions to a linear vector (i.e., transposing matrices and writing output for each of the 8 chunks) again incurs 50% overhead, ad deduplicating them actually again adds some time.

Going back a step, using canonical ntHash instead of forward ntHash takes 0.14 ns/bp extra, and determining the canonical strand (via a third ℓ-delayed stream and counting `GT` bases) takes another 0.49 ns/bp. As before, collecting and deduplicating are slow and add around 0.70ns/bp.

In conclusion, we see that iterating the chunks of the input and determining minimizers is quite fast, but that a lot of time must then be spent to "deinterleave" the output into a linear stream. As can be expected, canonical minimizers are slower to compute than forward minimizers, but the overhead is less than 50%, which seems quite low given that the ntHash and sliding window minimum computation are duplicated and a canonical-strand computation is added.

Full comparison. We compare against the `minimizer-iter` crate (v1.2.1) [Mar], which implements a queue-based sliding window minimum using `wyhash` [YBR] and also supports canonical minimizers. For an additional comparison, we optimized an implementation of the remap method with ntHash based on a code snippet by Daniel Liu [Liu23].

Results are in Table 12.2 and Figure 11.5. `minimizer-iter` takes around 26ns/bp for forward and 33ns/bp for canonical minimizers, and its runtime does not depend much on w and k, because the popping from the queue is unpredictable regardless of w. Rescan starts out at 11.7 ns/bp for $w = 5$ and gets significantly faster as w increases, converging to around 5 ns/bp for $w \gg 100$. This is explained by the fact that rescan has a branch miss every time the current minimizer falls out of the window, which happens for roughly half the minimizers at a rate of $1/(w+1)$. Thus, as w increases, the method becomes more predictable and branch misses go down. Our method, `simd-minimizers`, runs around 1.61 ns/bp for forward and 2.20 ns/bp for canonical minimizers when given packed input, and therefore is 3.4× to 6.8× faster than the rescan method.

[7] We have made sure that all layers of iterators are inlined into single function. This is usually needed for optimal performance. Even then, small changes to the generated code (by changing the source, or compiler version) can impact performance by as much as 20%.

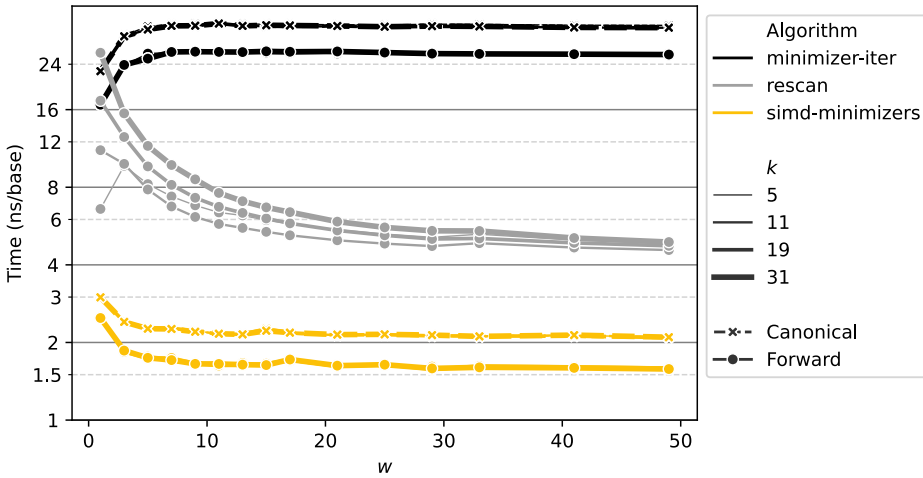


Figure 11.5 Running time (logarithmic) of `minimizer-iter`, rescan, and `simd-minimizers` for different values of w and k. For `minimizer-iter` and `simd-minimizers`, runtime is nearly independent of k, while both are slower when computing canonical minimizers (indicated with crosses).

In Figure 11.5, we see that `simd-minimizers`' performance is mostly independent of k and w since it is mostly data-independent. Only for small $w \leq 5$ it is slightly slower due to the larger number of minimizers and hence larger size of the output.

As we use SIMD with 8 lanes, we could in theory expect up to 8× speedup. In practice this is hard to reach because of constant overhead and because the overhead to work well with SIMD in the first place. In particular for large w, rescan benefits from very predictable and simple code and only outputs unique minimizer positions, making it very efficient. In SIMD, on the other hand, we use a data-independent algorithm, and output the minimizer position for every single window, which then has to be deduplicated. Thus, it is nice to see that even for large w, our method is over 3× faster, despite this overhead.

ASCII input and mulHash. Apart from taking bit-packed input, `simd-minimizers` also works on ASCII-encoded DNA sequences of ACTG characters directly, which are then packed into values $\{0, 1, 2, 3\}$ for ntHash (in that order) during iteration. This is around 0.25ns/bp slower, mostly because of the larger size of the unpacked input.

The mulHash variant is around 0.20ns/bp slower again, but works for any ASCII input. Performance on 100 MB of the Pizza&Chili corpus [FN07] English[8] and Sources[9] datasets is nearly identical to performance on the random DNA shown in Table 12.2.

Human genome. We also run `simd-minimizers` on the chromosomes of a human genome (T2T-CHM13v2.0[10] [NKR+22]), of total size 3.2 Gbp. Here, computing forward minimizers takes 5.19 seconds, and canonical minimizers takes 6.71 seconds for $(w, k) = (11, 21)$, which corresponds 1.67 and 2.15 ns/bp, which is within a few percent of Table 12.2.

[8] `https://pizzachili.dcc.uchile.cl/texts/nlang/`
[9] `https://pizzachili.dcc.uchile.cl/texts/code/`
[10] Available at `https://github.com/marbl/CHM13`

Density. For $(w, k) = (11, 21)$, we get a density of 0.173 for forward minimizers, and 0.167 for canonical minimizers, which are both close to the expected density of $2/(w+1) = 1/6 \approx 0.167$. Changing to $(w, k) = (19, 19)$, we get density 0.098 for forward minimizers and 0.100 for canonical minimizers, both close to the expected density of $2/(19+1) = 1/10 = 0.1$. Thus, we see that in practice, ntHash is a sufficiently random hash function to approach the expected density of random minimizers with a perfectly uniform random hash function.

Multithreading. We test throughput in a multithreaded setting, by using 6 threads to process the 25 chromosomes (22, X, Y, and mitochondrion) in parallel. This way, processing takes 0.97 s (forward) and 1.27 s (canonical) when the input data is already loaded into memory, showing slightly above 5× speedup. This is just below 6× speedup as the chromosomes don't perfectly partition the data into 6 equal parts, so that some threads finish before others.

For applications, we recommend to simply call our library in parallel from multiple threads as needed.

11.6 Conclusions and Future Work

Our library `simd-minimizers` computes minimizer positions 3.4× to 6.8× faster than other methods. Using the library, only a single function call is needed to obtain the list of (canonical) minimizer positions, taking as input the (packed) DNA sequence and the parameters k and w. General ASCII input are also supported, allowing use cases such as sketching protein sequences. We hope that the community will adopt `simd-minimizers` as the standard library to compute random minimizers.

Future work. We chose to use the data-independent two-stacks method as the core of our algorithm, that returns the minimizer position of every window and then requires deduplication. Given the promising performance of rescan for large w in Table 12.2, an interesting alternative could be to go the opposite way and speed up the rescan step. This is particularly relevant when minimizers are sparse, as the number of branches may be small enough for linear scans to benefit from SIMD. Accelerating linear scans could also be useful for smaller inputs such as short reads, where splitting into 8 chunks may not be very efficient.

Another approach would be to use 512-bit AVX512 instructions and process 16 lanes in parallel. In theory that could be another 2× faster, but in practice the collecting and deduplicating of values may become an even larger bottleneck. In simple experiments, without further profiling and optimizing the code, it is in fact 20% *slower*.

Additionally, implementing low-density schemes like the open-closed mod-minimizer [GKLP25, GKP24] would be a valuable extension.

11.A Results for NEON Architecture

The NEON experiments are run on a performance core on an Apple M1 chip with 4 efficiency and 4 performance cores. The performance core runs at 3.2GHz and has 128KiB of L1 cache, 12MiB of shared L2 cache (for the performance cores) and 8MiB of shared L3 cache (for the whole system).

Table 11.3 Time per base taken when adding steps of the implementation, for $(w, k) = (11, 21)$ on NEON architecture.

Part	ns/bp
Gather and sum all bases	0.10
+ collect to vector	0.18
+ collect the delayed bases	0.19
+ ntHash	0.33
+ sliding window min	0.82
+ collect	0.95
+ dedup	**1.38**
+ canonical nthash	0.85
+ canonical strand	1.16
+ collect	1.37
+ dedup	**1.82**

Table 11.4 Comparison of our `simd-minimizers` implementation against `minimizer-iter` [Mar] and a rescan implementation based on [Liu23]. Times in ns/bp are shown for both forward and canonical minimizers (where supported), and for various (w, k) tuples, on NEON architecture.

(w, k):	$(5, 31)$		$(11, 21)$		$(19, 19)$	
Method	fwd.	cano.	fwd.	cano.	fwd.	cano.
`minimizer-iter`	12.89	16.60	14.26	18.08	14.20	18.04
Rescan ntHash	5.95	-	3.92	-	2.82	-
`simd-minimizers` ntHash						
- packed input	1.42	1.85	1.38	1.82	1.38	1.83
- on-the-fly packing	1.66	2.07	1.58	1.99	1.59	2.02
Rescan mulHash	6.94	-	3.93	-	3.42	-
`simd-minimizers` mulHash						
- packed input	1.97	4.34	1.91	4.27	1.93	4.26
- on-the-fly packing	2.11	4.37	2.04	4.41	2.03	4.30
- ASCII input	2.12	4.39	2.06	4.44	2.04	4.29

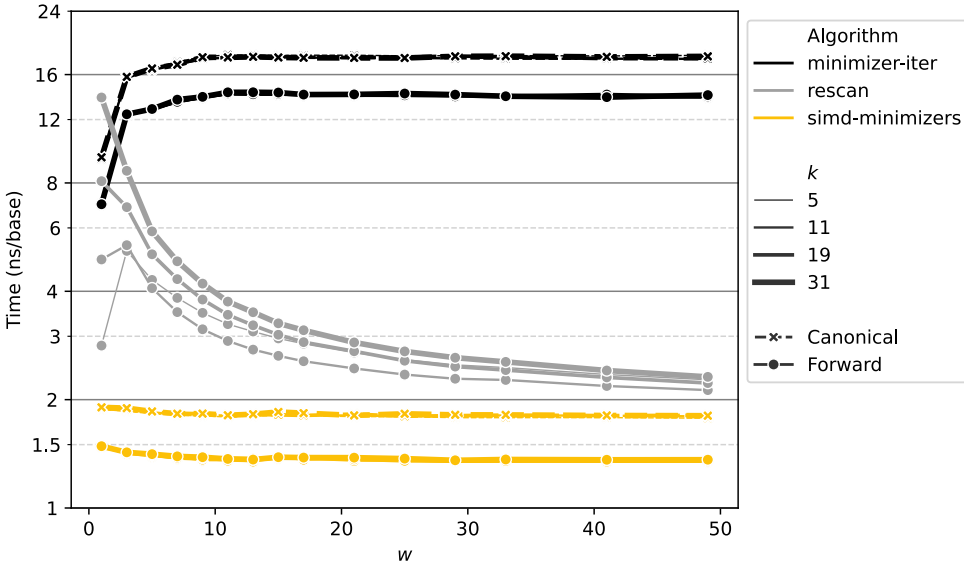


Figure 11.6 Running time of `minimizer-iter`, rescan and `simd-minimizers` on NEON architecture for different values of w and k.

12 PtrHash: Minimal Perfect Hashing at RAM Throughput

Summary

Given a set K of n keys, a minimal perfect hash function (MPHF) is a collision-free bijective map $\mathsf{H}_{\mathsf{mphf}}$ from K to $\{0, \dots, n-1\}$. These functions have uses in databases, search engines, and are used in multiple bioinformatics indexing tools. For example, the PTHash MPHF is used in SSHash, a data structure on k-mers that supports membership queries. In this case, PTHash only takes around 5% of the total space of SSHash, and thus, trading slightly more space for faster queries is beneficial.

In this chapter, we take PTHash (and the more recent PHOBIC) as a starting point, and optimize them for query throughput, at the cost of using slightly more space. At the core, we achieve this by simplifying the data structure as much as possible to make queries as simple and fast as possible. Then, we apply the *batching* and *streaming* techniques to achieve query throughput that is within 10% of the maximum possible throughput of the CPU, at 8 ns/query.

Attribution

This chapter is based on "PtrHash: minimal perfect hashing at RAM throughput" [GK25]. Large parts of this chapter are copied verbatim or with minor changes from that publication. Both PtrHash software and paper are fully my own work.

12.1 Introduction

Given a set of n keys $\{k_0, \dots, k_{n-1}\}$, a *hash function* maps them to some co-domain $[m] := \{0, \dots, m-1\}$. When $m \geq n$ and the hash is injective (collision-free), it is also called *perfect*. When additionally $m = n$ and it is surjective onto $[n]$, it is *minimal*. Thus, a *minimal perfect hash function* (MPHF) bijectively maps a set of n keys onto $[n]$.

Metrics. Various aspects of MPHF data structures can be optimized. First, one could minimize its space usage and try to approach the $\log_2(e) = 1.4427$ bits/key lower bound [Meh82]. Indeed, there are many recent works in this direction, such as Bipartite ShockHash-RS, which uses under 1.5 bits/key [LSW24, LSW23a, Leh24], and Consensus-RecSplit [LSWZ25], which goes as low as 1.444 bits/key.

In this paper, we focus primarily on optimizing for query throughput and secondarily on construction speed, while relaxing space usage up to 3 bits/key. This continues the line of work of FCH [FCH92], PTHash [PT21, PT24], and PHOBIC [HLP+24a], that all provide relatively fast queries.

Problem statement. Construct a *minimal perfect hash function* data structure $\mathsf{H}_{\mathsf{mphf}}$ that is fast to query, ideally using one memory access per lookup, and fast to construct, while staying below 3 bits/key of space.

Motivation. Our main motivating application is to optimize the use of PTHash in SSHash [Pib22], a data structure to index a set of k-mers (sequences of k DNA bases). There, the MPHF only takes around 5% of the total space. Thus, a slightly increased space usage of the MPHF has little effect on the total space, while faster lookups could significantly improve the overall query speed. In this application, k-mers are typically encoded as 64-bit integers, and thus we will focus our attention on integer keys.

Further applications can be found in domains such as networking [LPB06], databases [Cha05], and full-text indexing [BN14], where one could imagine hashing IP addresses, URLs, or (compact) suffix-trie edge labels.

Contributions. We introduce PtrHash, a minimal perfect hash function that is primarily optimized for query throughput and construction speed, at the cost of slightly more memory usage. It builds on the same principles als PTHash(-HEM) and Phobic: first, keys are partitioned into *parts*. Then, the keys in each part are further split into *buckets*, and each bucket is assigned a *pilot* that controls the values (*slots*) that the keys in the bucket hash to.

Compared to PTHash and PHOBIC, the main novelties of PtrHash are:

1. a fixed number of buckets *and slots* per part, removing the need for a part-offset lookup;
2. the use of fixed-width 8-bit *pilots*, so that no compact encoding is needed;
3. a pilot search based on Cuckoo hashing;
4. remapping using a *single* remap table, again simplifying lookups;
5. a remap table based on a per-cacheline Elias-Fano encoding [Eli74, Fan71], CacheLineEF;
6. the use of *prefetching* to *stream* multiple queries in parallel.

Results. When using 300 million string keys, PtrHash with default parameters takes 2.4 bits/key and is nearly as fast to construct as the fastest other methods, while being much faster to query. Compared to the next-fastest method to query, PtrHash provides 2.1× faster queries when looping naively, or 3.3× faster when streaming.

When using 10^9 integer keys instead, PtrHash can achieve an inverse throughput[1] as low as 12 ns/key when looping over queries, or even 8 ns/key when streaming.

The hardware used for benchmarking has a maximum single-threaded memory bandwidth of 7.4 ns per cache line. Thus, under the assumption[2] that almost every query requires reading at least one new cache line from main memory, our method is close to the maximum possible query throughput. Likewise, in a multi-threaded setting, PtrHash can fully saturate the DDR4 memory bandwidth while answering around 1 query per fetched cache line.

---

[1] For interpretability and consistency with latency numbers, we report the inverse throughput in nanoseconds per key, rather than keys per second. We will still refer to this as *throughput*, rather than *inverse throughput*, following the intel instruction manual.

[2] This is a strong assumption, and indeed, PtrHash with the cubic bucket assignment function already slightly breaks this assumed lower bound.

12.2 Related work

There is a vast amount of literature on (minimal) perfect hashing, going back to e.g. [Maj96]. Here we only give a highlight of recent approaches. We refer the reader to Section 2 of [PT24] and Sections 4 and 8 of the thesis of Hans-Peter Lehmann [Leh24], which contains a nice overview of different approches taken by various tools.

Space lower bound. There is a lower bound of $n \log_2(e)$ bits to store a minimal perfect hash function on n random keys [Meh82]. To get some feeling for this bound, consider any hash function. Intuitively the probability that this is an MPHF is $n!/n^n$. From this, it follows that at most, around $\log_2(n^n/n!) \approx n \log_2(e)$ bits of information are needed to "steer" the hash function in the right direction. Now, a naive approach is to use a seeded hash function, and try $O(e^n)$ seeds until a perfect hash function is found. However, that is not feasible in practice. The method that currently gets closest to the lower bound is Consensus-RecSplit [LSWZ25], which goes as low as 1.444 bits/key.

Bucket placement. PtrHash builds on methods that first group the keys into buckets of a few keys. Then, keys in the buckets are assigned their hash value one bucket at a time, such that newly assigned values do not collide with previously taken values. All methods iterate different possible key assignments for each bucket until a collision-free one is found, but differ in the way hash values are determined. To speed up this search, large buckets are assigned a hash before small buckets, since smaller buckets are easier to place when many slots are already taken.

FCH [FCH92] uses a fixed number of bits to encode the seed for each bucket and uses a *skew* distribution of bucket sizes. The seed stored in each bucket determines how far the keys are *displaced* (rotated) to the right from their initially hashed positions. A fallback hash can be used if needed, and construction can fail if that also does not work. CHD [BBD09] uses uniform bucket sizes, but uses a variable-width encoding for the seeds. PTHash [PT21] combines these two ideas and introduces a number of compression schemes for the seed values, that are called *pilots*. Instead of directly generating an MPHF, it first generates a PHF to $[n']$ for $n' = n/\alpha \approx n/0.99$, and values mapping to positions $\geq n$ are *remapped* to the skipped values in $[n]$. PTHash-HEM [PT24] first partitions the keys, and uses this to build multiple parts in parallel. This also enables external-memory construction. Lastly, PHOBIC [HLP+24a] improves from the simple *skew* distribution of FCH to an *optimal bucket assignment function*, which speeds up construction and enables smaller space usage. Secondly, it partitions the input into parts of expected size 2500 and uses the same number of buckets for each part. Then, it uses that the pilot values of the i'th bucket of each part follow the same distribution, and encodes them together. Together, this saves 0.17 bits/key over PTHash. Lastly, some of the ideas in PtrHash (fixed 8-bit pilots and cuckoo hashing) have been independently proposed in [Her23].

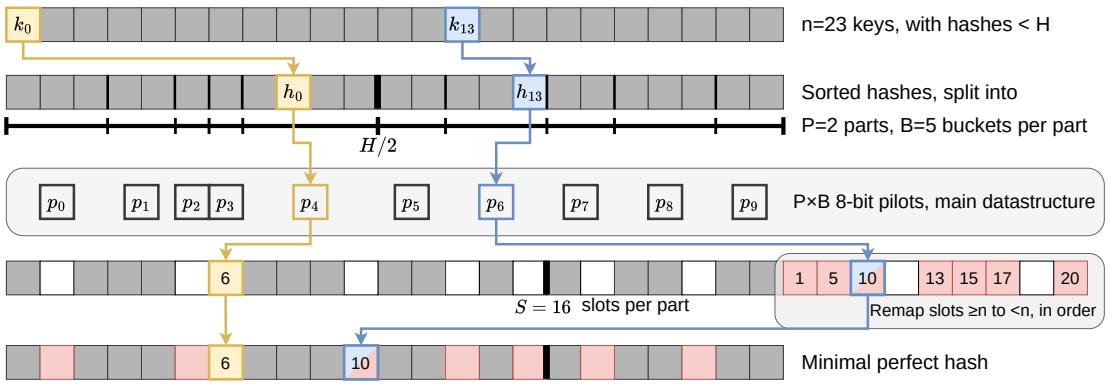


Figure 12.1 Overview of PtrHash on $n = 23$ keys. The keys are hashed into $[H] = [2^{64}]$ and this range is split into $P = 2$ parts and $B = 5$ buckets per part. The key highlighted in yellow has a the 9'th smallest hash, and ends up in *bucket* 4 (starting at index 0). The corresponding *pilot* p_4 hashes the key to *slot* 6. The array of pilots (grey background) is the main component of the PtrHash data structure, and ensures that all keys hash to different slots. The blue key has a hash in the second part (upper half) of hashes, in bucket 6. It gets hashed to slot 25, which is larger than the number of keys $n = 23$. Thus, it is *remapped* (along with the other red cells) into an empty slot $< n$ via a (compressed) list of free slots, which is the second main component of the data structure.

12.3 PtrHash

The core design goal of PtrHash[3] is to simplify PTHash to speed up both query speed and construction time, at the cost of possibly using slightly more memory.

12.3.1 Overview

Before going into details, we first briefly explain the fully constructed PtrHash data structure and how to query it, see Figure 12.1. We also highlight differences to PTHash [PT21] and PHOBIC [HLP+24a].

Parts and buckets. The input is a set of n *keys* $\{k_0, \dots, k_{n-1}\}$ that we want to hash to n *slots* $[n] := \{0, \dots, n-1\}$. We first hash the keys using a 64-bit hash function h into $\{\mathsf{h}(k_0), \dots, \mathsf{h}(k_{n-1})\}$. The total space of hashes $[2^{64}]$ is equally partitioned into P *parts*, and the part of a key is easily found as $\lfloor P \cdot \mathsf{h}(k_i)/2^{64} \rfloor = \mathsf{hi}(P \cdot \mathsf{h}(k_i))$ [Lem19], where $\mathsf{hi}(a \cdot b)$ returns the high 64 bits of the product of two 64-bit integers, and likewise, $\mathsf{lo}(a \cdot b)$ returns the low 64 bits. Then, the expected n/P keys in each part are further split into exactly B non-uniform *buckets*: each key has a *relative position* x inside the part, and this is passed through a *bucket assignment function* $\gamma : [0, 1) \mapsto [0, 1)$ such as $\gamma(x) = x^2$ that controls the distribution of expected bucket sizes [HLP+24a], as explained in detail

[3] The PT in PTHash stand for *Pilot Table*. The author of the present paper mistakenly understood it to stand for Pibiri and Trani, the authors of the PTHash paper. Due to the current author's unconventional last name, and PTGK not sounding great, the first initial (R) was appended instead, doubling as a hint that PtrHash is written in Rust. As things go, nothing is as permanent as a temporary name. Furthermore, we follow the Google style guide and avoid a long run of uppercase letters, and write PtrHash instead of PTRHash.

in Section 12.3.3. The result is then scaled to a bucket index in $[B]$:

$$\begin{aligned} \mathsf{part}(k_i) &:= \mathsf{hi}(P \cdot \mathsf{h}(k_i)), \\ x &:= \mathsf{lo}(P \cdot \mathsf{h}(k_i))/2^{64}, \\ \mathsf{bucket}(k_i) &:= \mathsf{hi}(B \cdot (2^{64} \cdot \gamma(x))). \end{aligned} \tag{12.1}$$

Slots and pilots. Now, the goal and core of the data structure is to map the n/P expected keys in each part to $S \approx (n/P)/\alpha$ *slots*, where $\alpha \approx 0.99$ gives us $\approx 1\%$ extra slots to play with. The pilot for each bucket controls to which slots its keys map. PtrHash uses fixed-width 8-bit *pilots* [Her23] $\{p_0, \ldots, p_{P \cdot B - 1}\}$, one for each bucket. Specifically, key k_i in bucket $\mathsf{bucket}(k_i)$ with pilot $p_{\mathsf{bucket}(k_i)}$ maps to slot

$$\mathsf{slot}(k_i) := \mathsf{part}(k_i) \cdot S + \mathsf{reduce}(\mathsf{h}(k_i) \oplus \mathsf{h_p}(p_{\mathsf{bucket}(k_i)}), S), \tag{12.2}$$

where $\mathsf{reduce}(\cdot, S)$ maps the random 64-bit integer into $[S]$ as explained below, and $\oplus$ denotes xor.

Compared to PHOBIC and PTHash(-HEM) [PT24], there are two differences here. First, while we still split the input into parts, we assign each part not only the same number of buckets, but also the *same* number of slots, instead of scaling the number of slots with the *actual* size of each part. This removes the need store a prefix sum of part sizes, and avoids one memory access at query time to look up the offset of the key's part. This idea was recently independently introduced as *ε-cost sharding* [Vig25]. Second, previous methods search for arbitrary large pilot values that require some form of compression to store efficiently. Our 8-bit pilots can simply be stored in an array so that lookups are simple.

We now go over some specific details.

Hash functions. The 8-bit pilots p_b are hashed into pseudo-random 64-bit integers by using FxHash [Bre] for $\mathsf{h_p}$, which simply multiplies the pilot with a *mixing constant* C after xoring by a global seed:

$$\mathsf{h_p}(p) := \mathsf{C} \cdot (p \oplus \text{seed}). \tag{12.3}$$

When the keys are 64-bit integers, we use this same FxHash algorithm to hash them ($\mathsf{h}(k) := \mathsf{C} \cdot k$), since multiplication by an odd constant is invertible modulo 2^{64} (since $\gcd(C, 2^{64}) = 1$) and hence collision-free. For other types of keys, the hash function depends on the number of elements. When the number of elements is not too far above 10^9, the probability of hash collisions with a 64-bit hash function is sufficiently small, and, following PHast [BS25], we use the 64-bit variant of GxHash [Gin23], a hash function based on AES hardware instructions. When the number of keys goes beyond $2^{32} \approx 4 \cdot 10^9$, the probability of 64-bit hash collisions increases. In this case, we use the 128-bit variant of GxHash. The high 64-bits determine the part and bucket in Equation (12.1), and the low 64-bits are used in Equation (12.2) to determine the slot.

The reduce function. To obtain the slot inside the current part, we must reduce the hash based on the key and its pilot to a number in $\{0, \ldots, S-1\}$. One way of doing this is to use "fast mod" [LKK19], which uses two multiplications when the modulus (the number of slots per part S) is less than 2^{32}.

When S is a power of two, we can instead use $\mathsf{reduce}(x, S) = \mathsf{hi}(C \cdot x) \bmod S$, which only needs a single multiplication and a bitmask. The multiplication by the mixing constant C ensures that all bits of x are used. In practice, this is the method we use.

When the number of parts is small, a drawback of limiting S to powers of two is that this could cause up to 50% empty slots. In this case, fast mod can be used for reliability. Then, that S must

not a power of two, so that $x \bmod S$ depends on all[4] bits of x. Additionally, we can only use a single part, simplifying queries.

Remapping. Since each part has slightly ($\approx 1\%$) more slots than keys, some keys will map to an index $\geq n$, leading to a *non-minimal* perfect hash function. To fix this, those are *remapped* back into the "gaps" left behind in slots $< n$ using a (possibly compressed) lookup table. This is explained in detail in Section 12.3.4.

Whereas PTHash-HEM uses a separate remap *per part*, PtrHash only has a single "global" remap table.

Construction. The main difficulty of PtrHash is during construction (Section 12.3.2), where we must find values of the pilots p_j such that all keys indeed map to different slots. Like other methods, PtrHash processes multiple parts in parallel. Within each part, it sorts the buckets from large to small and "greedily" assigns them the smallest pilot value that maps the keys in the bucket to slots that are still free. Unlike other methods though, PtrHash only allows pilots up to 255. When no suitable pilot is found, we use a method similar to (blocked) Cuckoo hashing [PR01, FPSS04]: a pilot with a minimal number of collisions is chosen, and the colliding buckets are "evicted" and will have to search for a new pilot.

Parameter values. In practice, we usually use $\alpha = 0.99$. Similar to PHOBIC, the number of buckets per part is set to $B = \lceil (\alpha \cdot S)/\lambda \rceil$, where λ is the expected size of each bucket and is around 3 to 4. The number of parts is $P = \lceil n/(\alpha S) \rceil$. Smaller parts fit better in cache and hence are faster to construct, while too small parts have too much variance in their size, causing some parts to possibly have more than S keys in them. Thus, we would like to choose S as the smallest size for which the probability that any part is over-subscribed is sufficiently small. Vigna [Vig25, eq. 3] shows that in practice, the following formula works well:

$$P \approx n/(\alpha S) \leq \frac{n\varepsilon^2/2}{\ln(n\varepsilon^2/2)},$$

where we use $\varepsilon = (1-\alpha)/2$ to ensure that all parts have at last half of the average number of free slots. For $\alpha = 0.99$, this reduces to

$$\alpha S \geq 80\,000 \cdot \ln(n/80\,000), \tag{12.4}$$

and so this is the number of key per part αS we choose, with a minimum of 80 000 for when $n \leq 80\,000$.

Streaming queries. PtrHash supports *streaming* queries, where multiple queries are processed in parallel. This allows prefetching pilots from memory, and thus increases throughput and better uses the available memory bandwidth. This is explained and evaluated in Section 12.A.

Sharding. When the number of keys is so large that their hashes do not fit into memory, one of three sharding strategies can be used: in-memory, on-disk, or hybrid. These are explained and evaluated in Section 12.B.

---

[4] Only depending on the $\lg_2 S$ low bits is not good enough, since the part and bucket functions only depend on the high $\lg_2(P \cdot B)$ bits, leaving some bits in the middle ususued.

12.3.2 Construction

Both PTHash-HEM and PHOBIC first partition the keys into parts, and then build an MPHF part-by-part, optionally in parallel on multiple threads. Within each part, the keys are randomly split into *buckets* of average size λ (Figure 12.1). Since $\lambda \leq 4$ in practice, the variance on bucket sizes is quite large. Thus, the buckets are sorted from large to small, and one-by-one *greedily* assigned a *pilot*, such that the keys in the bucket map to *slots* not yet covered by earlier buckets.

As more buckets are placed, there are fewer remaining empty slots, and searching for pilots becomes harder. Hence, PTHash uses $n/\alpha > n$ slots to ensure there sufficiently many empty slots for the last pilots. This speeds up the search and reduces the values of the pilots. PHOBIC, on the other hand, uses relatively small parts of expected size 2500, so that the search for the last empty slot usually should not take much more than 2500 attempts. Nevertheless, a drawback of the greedy approach is that pilots values have an uneven distribution, making it somewhat harder to compress them while still allowing fast access (e.g., requiring the interleaved coding of PHOBIC).

Hash-evict. In PtrHash, we instead use *fixed width*, single byte pilots. To achieve this, we use a technique resembling Cuckoo hashing [PR01] that was also independently found in [Her23, Section 4.5]. As before, buckets are greedily *inserted* from large to small. For some buckets, there may be no pilot in $[2^8]$ such that all its keys map to empty slots. When this happens, a pilot is found with the lowest weighted number of *collisions*. The weight of a collision with an element of a bucket of size s is s^2, to prevent *evicting*[5] large buckets, as those are harder to place. The colliding buckets are evicted by emptying the slots they map to and pushing them back onto the priority queue of remaining buckets. Then, the new bucket is inserted, and the next largest remaining or evicted bucket is processed.

In order to efficiently search for pilot vectors, we use a bitvector of taken slots. Additionally, we avoid infinite loops of evicted buckets by storing the 16 most recently placed buckets, and never displacing those.

12.3.3 Bucket Assignment Functions

During construction, slots fill up as more buckets are placed. Because of this, the first buckets are much easier to place than the later ones, when only few empty slots are left. To compensate for this, we can introduce an uneven distribution of bucket sizes, so that the first buckets are much larger and the last buckets are smaller. FCH [FCH92] accomplishes this by a *skew* mapping that assigns 60% of the elements to 30% of the buckets, so that those 30% are *large* buckets while the remaining 70% is *small* (Figure 12.2). This is also the scheme used by PTHash.

The optimal bucket function. PHOBIC [HLP+24a] provides a more thorough analysis and uses the optimal function $\gamma_p(x) = x + (1-x)\ln(1-x)$ when the target load factor is $\alpha = 1$. A small modification is optimal for $\alpha < 1$ [HLP+24b, Appendix B], but for simplicity we only consider the original γ_p. This function has derivative 0 at $x = 0$, so that many x values map close to 0. In practice, this causes the largest buckets to have size much larger than $\sqrt{S}$. Such buckets are hard to place, because by the birthday paradox they are likely to have multiple elements hashing to the same slot. To fix this, PHOBIC ensures the slope of γ is at least $\varepsilon = 1/\left(5\sqrt{S}\right)$ by using

[5] We would have preferred to call this method hash-displace, as *displace* is the term used instead of *evict* in e.g. the cuckoo filter [FAKM14]. Unfortunately, *hash and displace* is already taken by hash-and-displace [Pag99, BBD09].

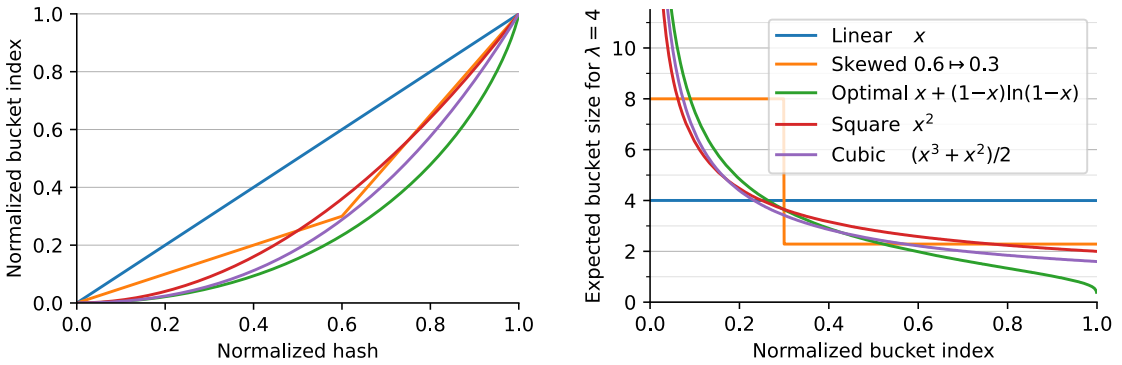


Figure 12.2 The left shows various bucket assignment functions γ, such as the piecewise linear function (skewed) used by FCH and PTHash, and the optimal function introduced by PHOBIC. Flatter slopes at $x = 0$ create larger buckets, while steeper slopes at $x = 1$ create more small buckets, as shown on the right, as the distribution of expected bucket sizes given by $(\gamma^{-1})'$ when the expected bucket size is $\lambda = 4$.

$\gamma_{p,\varepsilon(x)} = x + (1-\varepsilon)(1-x)\ln(1-x)$ instead. For simplicity in the implementation, we fix $\varepsilon = 1/2^8$, which works well in practice.

Approximations. For PtrHash, we aim for high query throughput, and thus we would like to only use simple computations and avoid additional lookups as much as possible. To this end, we replace the $\ln(1-x)$ by its first order Taylor approximation at $x = 0$, $\ln(1-x) \approx -x$, giving the quadratic $\gamma_2(x) := x^2$. Using the second order approximation $\ln(1-x) \approx -x - x^2/2$ results in the cubic $\gamma(x) = (x^2 + x^3)/2$. This version again suffers from too large buckets, so in practice we use $\gamma_3(x) = \frac{2^8-1}{2^8} \cdot (x^2 + x^3)/2 + \frac{1}{2^8} \cdot x$. We also test the trivial $\gamma_1(x) := x$.

These values can all be computed efficiently by using that the input and output of γ are 64-bit unsigned integers representing a fraction of 2^{64}, so that e.g. x^2 can simply be computed as $\mathsf{hi}(x \cdot x)$.

12.3.4 Remapping using CacheLineEF

Like PTHash, PtrHash uses a parameter $0 < \alpha \leq 1$ to use a total of $n' = n/\alpha$ slots, introducing $n' - n$ additional free slots. As a result of the additional slots, some, say R, of the keys will map to positions $n \leq q_0 < \cdots < q_{R-1} < n'$, causing the perfect hash function to not be *minimal*.

Remapping. Since there are a total of n keys, this means there are exactly R empty slots ("gaps") left behind in $[n]$, say at positions L_0 to L_{R-1}. We *remap* the keys that map to positions $\geq n$ to the empty slots at positions $< n$ to obtain a *minimal* perfect hash function.

A simple way to store the remap is as a plain array F, such that $F[q_i - n] = L_i$. PTHash encodes this array using Elias-Fano coding [Eli74, Fan71], after setting undefined positions of F equal to their predecessor. The benefit of a plain F array is fast and cache-local lookups, whereas Elias-Fano coding provides a more compact encoding that typically requires multiple lookups to memory.

CacheLineEF. We would like to answer each query by reading only a single cache line from memory. To do this, we use a method based on *interleaving* data. First, the list of non-decreasing F positions is split into chunks of $C = 44$ values $\{v_0, \ldots, v_{43}\}$, with the last chunk possibly containing fewer values. We assume that values are at most 40 bits, and that the average difference between

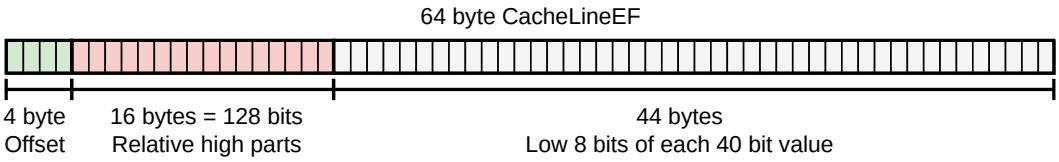


Figure 12.3 Overview of the CacheLineEF data structure.

adjacent values in each chunk is not more than 500. Then, each chunk is encoded into 64 bytes that can be stored as single cache line, as shown in Figure 12.3.

We first split all values into their 8 *low* bits ($v_i \bmod 2^8$) and 32 *high* bits ($\lfloor v_i/2^8 \rfloor$). Further, the high part is split into an *offset* (the high part of v_0) and the *relative* high part:

$$v_i = 2^8 \cdot \underbrace{\lfloor v_0/2^8 \rfloor}_{\text{Offset}} + 2^8 \cdot \underbrace{\left(\lfloor v_i/2^8 \rfloor - \lfloor v_0/2^8 \rfloor\right)}_{\text{Relative high part}} + \underbrace{(v_i \bmod 2^8)}_{\text{Low bits}}. \tag{12.5}$$

This is stored as follows.

- First, the 32 bit offset $\lfloor v_0/2^8 \rfloor$ is stored.
- Then, the relative high parts are encoded into 128 bits. For each $i \in [44]$, bit $i + \lfloor v_i/2^8 \rfloor - \lfloor v_0/2^8 \rfloor$ is set to 1. Since the v_i are increasing, each i sets a distinct bit, for a total of 44 set bits.
- Lastly, the low 8 bits of each v_i are directly written to the 44 trailing bytes.

Lookup. The value at position i is found by summing the terms of Equation (12.5). The offset and low bits can be read directly. This relative high part can be found as $2^8 \cdot (\mathsf{select}(i) - i)$, where $\mathsf{select}(i)$ gives the position of the i'th 1 bit in the 128-bit-encode relative high parts. In practice, this can be implemented efficiently using a `popcount` to go into the high or low half, followed by the `PDEP` instruction[6] provided by the BMI2 bit manipulation instruction set [PBJ17].

Limitations. CacheLineEF uses $64/44 \cdot 8 = 11.6$ bits per value, which is more than the usual Elias-Fano, which for example takes $8 + 2 = 10$ bits per value for data with an average *stride* (gap between consecutive integers) of 2^8. Furthermore, values are limited to 40 bits, covering 10^{12} items. The range could be increased to 48 bit numbers by storing 5 bytes of the offset, but this has not been necessary so far. Lastly, each CacheLineEF can only span a range of around $(128 - 44) \cdot 2^8 = 21\,504$, or an average stride of 500. This means that for PtrHash, we only use CacheLineEF when $\alpha \leq 0.99$, so that the average distance between empty slots is 100 and the average stride of 500 is not exceeded in practice. When $\alpha > 0.99$, a simple plain array can be used without much overhead.

12.4 Results

We now evaluate PtrHash construction and query throughput for different parameters, and compare PtrHash to other minimal perfect hash functions. All experiments are run on an Intel Core i7-10750H CPU with 6 cores and hyper-threading disabled. The frequency is pinned to 2.6 GHz.

[6] Unfortunately, while AMD Zen 2 does support this instruction, it is very slow in practice.

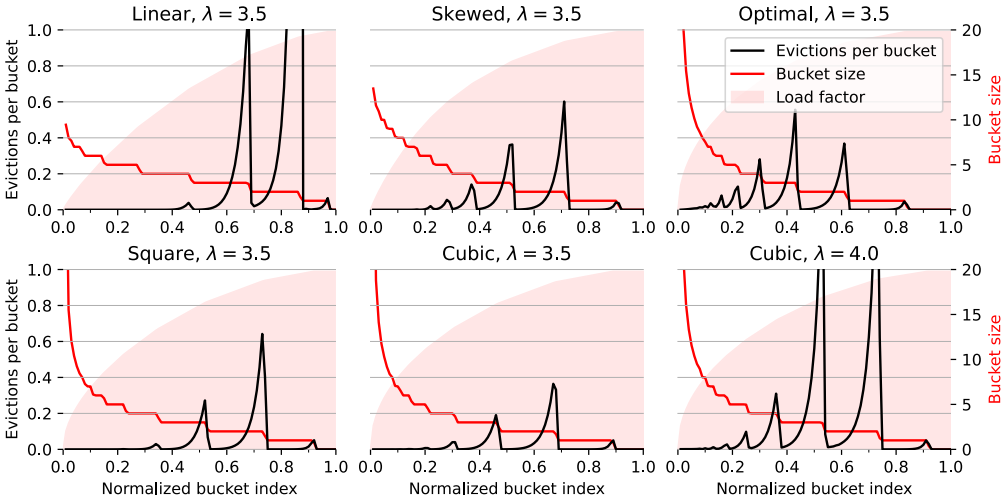


Figure 12.4 Bucket size distribution (red) and average number of evictions (black) per additionally placed bucket during construction of the pilot table, for different bucket assignment functions. Parameters are $n = 10^9$ keys, $S = 2^{18}$ slots per part, and $\alpha = 0.99$, and the red shaded load factor ranges from 0 to α. In the first five plots $\lambda = 3.5$ so that the pilots take 2.29 bits/key. For $\lambda = 4.0$ (bottom-right), the linear, skewed, and optimal bucket assignment functions cause endless evictions, and construction fails. The cubic function does work, resulting in 2.0 bits/key for the pilots.

Cache sizes are 32 KiB L1 and 256 KiB L2 per core, and 12 MiB shared L3 cache. Main memory is 64 GiB DDR4 at 3200 MHz, split over two 32 GiB banks.

In Section 12.4.1, we compare the effect of various parameters and configurations on the size, construction speed, and query speed of PtrHash. In Section 12.4.2, we compare PtrHash to other methods.

Further, Section 12.A.2 evaluates the effect of prefetching with batching and streaming queries. We select streaming with prefetching 32 iterations ahead as the default. We also show that in a multi-threaded setting, this can fully exhaust the available memory bandwidth.

Lastly, in Section 12.B.1 we state the results of constructing PtrHash on 50 billion keys using various sharding strategies.

12.4.1 Construction

The construction experiments use 10^9 random 64-bit integer keys, for which the data structure takes around 300 MB and thus is much larger than L3 cache. Unless otherwise mentioned, construction is in parallel using 6 cores. For the query throughput experiments, we also test on 20 million keys, for which the data structure take around 6 MB and easily fit in L3 cache. To avoid the time needed for hashing keys, and since our motivating application is indexing k-mers that fit in 64 bits, we always use random 64-bit integer keys, and hash them using FxHash.

Without using the external memory construction, memory usage during construction is dominated by the size of the input keys and their hashes, which are typically much larger than the few bits per key needed for the construction itself.

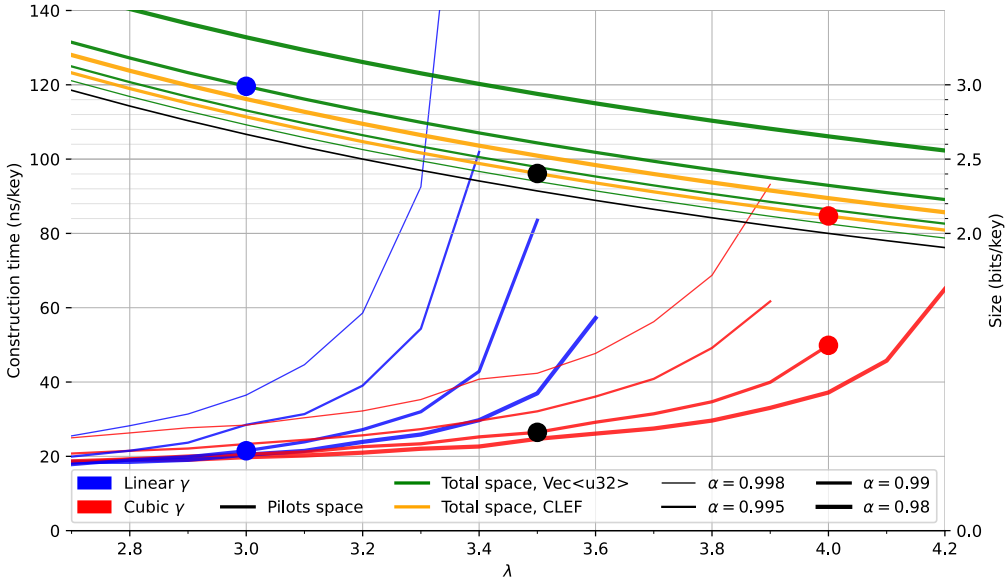


Figure 12.5 This plot shows the construction time (blue and red, left axis) and data structure size (black, green, and yellow, right axis) as a function of λ for $n = 10^9$ keys. Parallel construction time on 6 threads is shown for both the linear and cubic γ, and for various values of α (thickness). The curves stop because construction times out when λ is too large. For each λ, the black line shows the space taken by the array of pilots. For larger λ there are fewer buckets, and hence the pilots take less space. The total size including the remap table is shown in green (plain vector) and yellow (CacheLineEF) for various α. The blue (fast), black (default), and red (compact) dots highlight the chosen parameter configurations.

12.4.1.1 Bucket Functions

In Figure 12.4, we compare the performance of different bucket assignment functions γ in terms of the bucket size distribution and the number of evictions for each additionally placed bucket. We see that the linear $\gamma_1(x) = x$ has a lot of evictions for the last buckets of size 3 and 2, but like all methods it is fast for the last buckets of size 1 due to the load factor $\alpha < 1$. The optimal distribution of PHOBIC performs only slightly better than the skewed one of FCH and PTHash, and can be seen to create more large buckets since the load factor increases fast for the first buckets. The cubic γ_3 is clearly much better than all other functions, and is also tested with larger buckets of average size $\lambda = 4$, where all other functions fail.

In the remainder, we will test the linear γ_1 for simplicity and lookup speed, and the cubic γ_3 for space efficiency.

12.4.1.2 Tuning Parameters for Construction

In Figure 12.5 we compare the multi-threaded construction time and space usage of PtrHash on $n = 10^9$ keys for various parameters $\gamma \in \{\gamma_1, \gamma_3\}$, $2.7 \leq \lambda \leq 4.2$, $\alpha \in \{0.98, 0.99, 0.995, 0.998\}$, and plain remapping or CacheLineEF. We see that for fixed γ and α, the construction time appears to increase exponentially as λ increases. At too large λ, some parts fail to build after a total of $10S$ evictions, which is a hard limit we impose to avoid running into eviction cycles. Load factors α closer to 1 (thinner lines) achieve smaller overall data structure size, but take longer to construct

Table 12.1 Comparison of space usage (bits/key) and query throughput (ns/query) of PtrHash when using the recommended parameters with different remap structures. Query throughput is shown both for perfect hashing (without remap), and for minimal perfect hashing (with remap). Additionally, query throughput is shown both for a for-loop and for streaming.

Configuration	Pilots	Query PHF		Remap		Query MPHF	
	Space	Loop	Stream	Type	Space	Loop	Stream
	2.67	11.5	8.6	Vec<u32>	0.33	12.5	8.8
Fast				CacheLineEF	0.12	12.9	8.8
$\alpha = 0.99$, $\lambda = 3.0$, linear γ_1				EF	0.09	14.2	9.7
	2.29	17.6	7.9	Vec<u32>	0.33	20.0	8.6
Default				CacheLineEF	0.12	21.0	8.7
$\alpha = 0.99$, $\lambda = 3.5$, cubic γ_3				EF	0.09	21.2	9.6
	2.00	17.7	8.0	Vec<u32>	0.33	20.3	8.6
Compact				CacheLineEF	0.12	20.9	8.6
$\alpha = 0.99$, $\lambda = 4.0$, cubic γ_3				EF	0.09	21.7	9.7

and time out at smaller λ. The cubic γ_3 is faster to construct than the identity γ_1 for small $\lambda \leq 3.5$. Unlike γ_1, it also scales to much larger λ up to 4, and thereby achieves significantly smaller overall size.

We note that for small λ, construction time does converge to around 19!ns/key. A rough time breakdown is that for each key, 1 ns is spent on hashing, 5 ns on sorting all the keys, 12 ns to search for pilots, and lastly 1 ns on remapping to empty slots.

Recommended parameters. Based on these results, we choose three sets of parameters for further evaluation, as indicated with blue, black, and red dots in Figure 12.5:

- **Fast** (blue), aiming for query speed: using the linear γ_1, $\lambda = 3.0$, $\alpha = 0.99$, and a plain vector for remapping. Construction takes only just over 20 ns/key, close to the apparent lower bound, and space usage is 3 bits/key. This can be used when n is small, or more generally when memory usage is not a bottleneck.
- **Default** (black), a trade-off between fast construction and small space: using cubic γ_3, $\lambda = 3.5$, and $\alpha = 0.99$, with CacheLineEF remapping.
- **Compact** (red), aiming for small space: using the cubic γ_3, $\lambda = 4.0$, $\alpha = 0.99$, and CacheLineEF remapping. Construction now takes around 50 ns/key, but the data structure only uses 2.12 bits/key. In practice, this configuration sometimes ends up in endless eviction cycles, and $\lambda = 3.9$ may be better.

12.4.1.3 Remap

In Table 12.1, we compare the space usage and query throughput of the different remap data structures for both the fast and compact parameters, for $n = 10^9$ keys. We observe that the overhead of CacheLineEF is 2.75× smaller than a plain vector, and only 40% larger than Elias-Fano encoding as implemented in the sux library [Vig24].

The speed of non-minimal (PHF) queries that do not remap does not depend on the remap structure used.

For *minimal* (MPHF) queries with the for loop, with fast parameters, EF is significantly slower (14.2 ns) with the fast parameters than the plain vector (12.5 ns), while CacheLineEF (12.9 ns)

Table 12.2 Performance comparison of MPHF methods on 300 million random string keys of uniform length between 10 and 50. Construction time is shown for 6 threads. A * indicates single-threaded timings (optimistic 6-fold speedup in parentheses). Near-optimal values in each column are bolded.

	Approach		Configuration	Space bits/key	Construction 6t, ns/key	Query ns/query
Bruteforce	SIMDRecSplit		n=5, b=5	2.96	**26**	310
			n=8, b=100	**1.81**	66	258
	Bip. ShockHash-Flat		n=64	**1.62**	2140* (357)	201
	Consensus-RecSplit		k = 256, ε = 0.10	**1.58**	521* (87)	565
			k = 512, ε = 0.03	**1.49**	1199* (200)	528
Fingerprinting	FMPH		γ=2.0	3.40	44	168
			γ=1.0	2.80	69	236
	FMPHGO		s=4, b=16, γ=2.0	2.86	298	160
			s=4, b=16, γ=1.0	2.21	423	212
	FiPS		γ=2.0	3.52	93* (**16**)	109
			γ=1.5	3.12	109* (**18**)	124
Graph	SicHash		p_1=0.21, p_2=0.78, α=0.90	2.41	48	149
			p_1=0.45, p_2=0.31, α=0.97	2.08	63	141
Bucket placement	CHD		λ=3.0	2.27	1059* (177)	542
	PTHash		λ=4.0, α=0.99, C-C	3.19	403	77
			+ HEM		173	
			λ=5.0, α=0.99, EF	2.17	765	156
			+ HEM		323	
	PHOBIC		λ=3.9, α=1.0, IC-C	4.14	62	116
			λ=4.5, α=1.0, IC-R	2.34	80	179
			λ=6.5, α=1.0, IC-C	2.44	220	108
			λ=7.0, α=1.0, IC-R	**1.86**	446	157
	PtrHash	Fast	λ=3.0, α=0.99, γ_1, Vec	2.99	**27**	**33**
			+ streaming			**16**
		Default	λ=3.5, α=0.99, γ_3, CLEF	2.40	**32**	**37**
			+ streaming			**23**
		Compact	λ=4.0, α=0.99, γ_3, CLEF	2.12	63	**35**
			+ streaming			**23**

is only slightly slower. The difference is much smaller with the compact parameters, because the additional computations for the cubic γ_3 reduce the number of iterations the processor can work ahead. When streaming queries, for both parameter choices CacheLineEF is less than 0.1 ns slower than the plain vector, while EF is 1 ns slower.

In the end, we choose CacheLineEF when using compact parameters, but prefer the simpler and slightly faster plain vector for fast parameters. Since $\alpha = 0.99$ is close to 1, the remap structure is not accessed much, and the performance improvement of CacheLineEF over plain EliasFano coding is not too large.

12.4.2 Comparison to Other Methods

In Table 12.2 we compare the performance of PtrHash against other methods on short, random strings. In particular, we compare against methods and configurations that are reasonably fast

to construct: SIMDRecSplit [EGV20, BKLS23], Bipartite ShockHash-Flat [LSW24, LSW23a], Consensus-RecSplit [LSWZ25], FMPH and FMPHGO [Bel23], FiPS [Leh24], SicHash [LSW23b], CHD [BBD09], PTHash [PT21, PT24], and PHOBIC [HLP+24a]. The specific parameters are based on Table 1 of [HLP+24a], Table 8.1 of [Leh24], and Table 3 of [Bel23]. These results were obtained using the excellent MPHF-Experiments library [Leh25] by Hans-Peter Lehmann. Construction is done on 6 threads in parallel when supported. By default, the framework queries one key at a time. For PtrHash with streaming queries, we modified this to query all keys at once.

Input. The input is 300 million random strings of random length between 10 and 50 characters. This input size is such that the MPHF data structures take around 75 MB, which is much larger than the 12 MB L3 cache.

PtrHash. As expected, the space usage of PtrHash matches the numbers of Table 12.1. In general, PtrHash can be slightly larger due to rounding in the number of parts and slots per part, but for large inputs like here this effect is small. Construction times per key are slightly slower than as predicted by Figure 12.5, while we might expect slightly faster construction due to the lower number of keys. Likely, the slowdown is caused by hashing the input strings. The hashing of input strings has a much worse effect on query throughput. In Table 12.1, we obtained query throughput of 12 ns and 18 ns for the fast and compact configurations when looping over integer keys, and as low as 8 ns when streaming queries. With string inputs, these numbers increase to 33 ns resp. 35 ns when looping, and 16 ns (resp. 23 ns) when streaming. A similar effect can be seen when comparing Tables 3 and 4 of [Bel23].

Speed. We observe that PtrHash with fast parameters is the fastest to construct alongside SIMDRecSplit (27 ns/key and 26 ns/key) and FiPS (16 ns/key, assuming optimal scaling to 6 threads), resulting in around 3 bits/key for all three methods. However, query throughput of PtrHash is 9× (SIMDRecSplit) resp. 3.3× (FiPS) faster, going up to 19× resp. 6.8× faster when streaming all queries at once. Compared to the next-fastest method to query, PTHash-CC (HEM), PtrHash is twice faster to query (or nearly 5× when streaming), is 6.5× faster to build, and even slightly smaller.

With default parameters, PtrHash is 2.1× faster to query than the fastest configuration of PTHash, and 3.3× faster when using streaming, while being over 5× faster to construct. Indeed, the speedup in query speed is explained by the fact that only a single memory access is needed for most queries (compared to ≥ 2 for PtrHash-HEM and PHOBIC), and generally by the fact that the code for querying is short.

Space. PtrHash with the fast parameters is larger (2.99 bits/key) than some other methods, but compensates by being significantly faster to construct and/or query. When space is of importance, the compact version can be used (2.12 bits/key). This takes 2.4× longer to build at 63 ns/key, and has only slightly slower queries. Compared to methods that are smaller, PtrHash is over 3× faster to build than PHOBIC. Consensus, SIMDRecSplit, and SicHash achieve smaller space of 1.58, 1.81, and 2.08 bits/key in comparable time (63-87 ns/key), but again are at least 3× slower to query, or over 6× compared to streaming queries.

12.5 Conclusions and Future Work

We have introduced PtrHash, a minimal perfect hash function that builds on PTHash and PHOBIC. Its main novelty is the used of fixed-width 8-bit pilots that simplify queries. To make this possible, we use *hash-and-evict*, similar to Cuckoo hashing: when there is no pilot that leads to a collision-free placement of the corresponding keys, some other pilots are *evicted* and have to search for a new value.

The result is an MPHF with twice faster queries (37 ns/key) than any other method (at least 77 ns/key) for datasets larger than L3 cache. Further, due to its simplicity, queries can be processed in *streaming* fashion, giving another two times speedup (as low as 16 ns/key). At this point, the hashing of string inputs becomes a bottleneck. For integer keys, such as k-mers, much higher throughput of up to 8 ns/key can be obtained, close to the 7.4 ns per cache line bandwidth, or when using multiple cores even saturating the main memory (2.5 ns/key).

Future work. A theoretical analysis of our method is currently missing. While the hash-evict strategy works well in practice, we currently have no relation between the bucket size λ, load factor α, and the number of evicts arising during construction. Such an analysis could help to better understand the optimal bucket assignment function, like PHOBIC [HLP+24a] did for the case without eviction. Possibly, the analysis of [Her23, Section 5] could be extended to fully cover our method.

Second, the size of pilots could possibly be improved by further parameter tuning. In particular we use 8-bit pilots, while slightly fewer or more bits may lead to smaller data structures. An experiment with 4-bit pilots was not promising, however.

To further improve the throughput, we suggest that more attention is given to the exact input format. As already seen, hashing all queries at once can provide significant performance gains via prefetching. For string input specifically, it is more efficient when the strings are consecutively packed in memory rather than separately allocated, and it might be more efficient to explicitly hash multiple strings in parallel. More generally, applications should investigate whether they can be rewritten to take advantage of streaming queries. Furthermore, current throughput is limited by the fact that nearly every query needs to fetch a new cache line. It would be interesting to design an MPHF that only requires, say, half a cache line per query, or to disprove the existance of such an MPHF.

Lastly, we refer the reader to PHast [BS25], an MPHF that introduces a number of interesting simplifications, leading to a datastructure is both smaller and faster to query than PtrHash, although it is somewhat slower to construct. It remains an open problem whether it is possible to construct an MPHF with space within 0.1 bits/key from the lower bound that is as fast to query as PtrHash and PHast.

12.A Query Throughput

12.A.1 Batching and Streaming

Throughput. In practice in bioinformatics applications such as SSHash, we expect many independent queries to the MPHF. This means that queries can be answered in parallel, instead of one by one. Thus, we should optimize for query *throughput* rather than individual query latency. We report throughput as *inverse throughput* in amortized nanoseconds per query, rather than the usual queries per second.

Out-of-order execution. An MPHF on 10^9 keys requires memory at least 1.5 bits/key $\times 10^9$ keys = 188 MB, which is much larger than the L3 cache of size around 16 MB. Thus, most queries require reading a pilot from main memory (RAM), which usually has a latency around 80 ns. Nevertheless, existing MPHFs such as FCH [FCH92] achieve an inverse throughput as low as 35 ns/query on such a dataset [PT21]. This is achieved by *pipelining* and the *reorder buffer*. For example, Intel Skylake CPUs can execute over 200 instructions ahead while waiting for memory to become available [Won13, Dow17]. This allows the CPU to already start processing future queries and fetch the required cache lines from RAM while waiting for the current query. Thus, when each iteration requires less than 100 instructions and there are no branch-misses, this effectively makes up to two reads in parallel. A large part of speeding up queries is then to reduce the length of each iteration so that out-of-order execution can fetch memory more iterations ahead.

Prefetching. Instead of relying on the CPU hardware to parallellize requests to memory, we can also explicitly *prefetch*[7] cache lines from our code. Each prefetch requires a *line fill buffer* to store the result before it is copied into the L1 cache. Skylake has 12 line fill buffers [Dow18], and hence can support up to 12 parallel reads from memory. In theory, this gives a maximal random memory throughput around 80/12 = 6.67 ns per read from memory, but in practice experiments show that the limit is 7.4 ns per read. Thus, our goal is to achieve a query throughput of 7.4 ns.

We consider two models to implement prefetching: batching and streaming.

Batching. In this approach, the queries are split into batches (chunks) of size B, and are then processed one batch at a time. In each batch, two passes are made over all keys. In the first pass, each key is hashed, its bucket it determined, and the cache line containing the corresponding pilot is prefetched. In the second pass, the hashes are iterated again, and the corresponding slots are computed.

Streaming. A drawback of batching is that at the start and end of each batch, the memory bandwidth is not fully saturated. Streaming fixes this by prefetching the cache line for the pilot B iterations ahead of the current one, and is able to sustain the maximum possible number of parallel prefetches throughout, apart from at the very start and end.

[7] There are multiple types of prefetching instructions that prefetch into a different level of the cache hierarchy. We prefetch into all levels of cache using `prefetcht0`. Other prefetch variants give similar results.

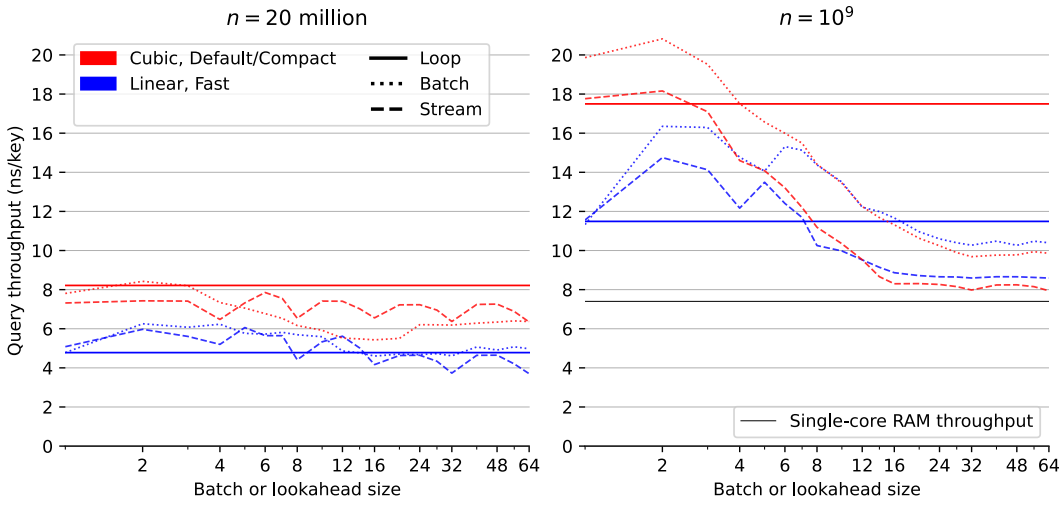


Figure 12.6 Query throughput of prefetching via batching (dotted) and streaming (dashed) with various batch/lookahead sizes, compared to a plain for loop (solid), for $n = 20 \cdot 10^6$ (left) and $n = 10^9$ (right) keys. Blue shows the results for the fast parameters, and red for the compact parameters. Default parameters give performance nearly identical to the compact parameters, since the main differentiating factor is the use of γ_1 versus γ_3. All times are measured over a total of 10^9 queries, and for (non-minimal) perfect hashing only, *without* remapping.

12.A.2 Evaluation

A note on benchmarking throughput. To our knowledge, all recent papers on (minimal) perfect hashing measure query speed by first creating a list of keys, and then querying all keys in the list, as in `for key in keys { ptr_hash.query(key); }`. One might think this measures the average latency of a query, but that is not the case, as the CPU will execute instructions from adjacent iterations at the same time. Indeed, as can be seen in Table 12.1, this loop can be as fast as 12 ns/key for $n = 10^9$, which is over 6 times faster than the RAM latency of around 80 ns (for an input of size 300 MB), and thus, at least 6 iterations are being processed in parallel.

Hence, we argue that existing benchmarks measure (and optimize for) throughput and that they assume that the list of keys to query is known in advance. We make this assumption explicit by changing the API to benchmark all queries at once, as in `ptr_hash.query_all(keys)`. This way, we can explicitly process multiple queries in parallel.

We also argue that properly optimizing for throughput is relevant for applications. SSHash, for example, queries all minimizers of a DNA sequence, which can be done by first computing and storing those minimizers, followed by querying them all at once.

We now explore the effect of the batch size and number of parallel threads on query throughput.

Batching and Streaming. In Figure 12.6, we compare the query throughput of a simple for loop with the batching and streaming variants with various batch/lookahead sizes. We see that both for small $n = 20 \cdot 10^6$ and large $n = 10^9$, the fast parameters yield higher throughput than the compact parameters when using a for loop. This is because of the overhead of computing $\gamma_3(x)$. For small n, batching and streaming do not provide much benefit, indicating that memory latency is not a bottleneck. However, for large n, both batching and streaming improve over the plain for loop. As expected, streaming is faster than batching here. For streaming, throughput saturates when

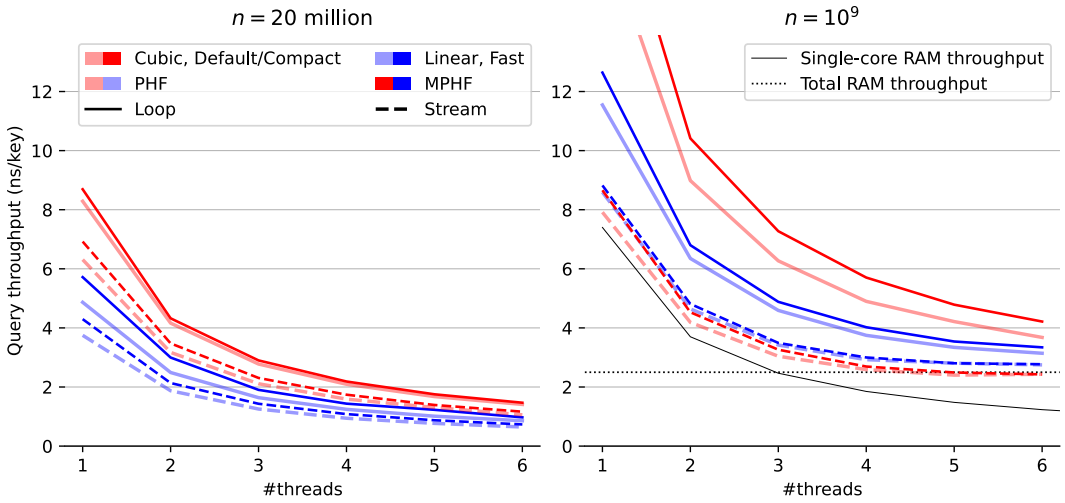


Figure 12.7 In this plot we compare the throughput of a for loop (solid) versus streaming (dashed) for multiple threads, for both non-minimal (dimmed) and minimal (bright) perfect hashing. The left shows results for $n = 20 \cdot 10^6$, and the right shows results for $n = 10^9$. In blue are the results for the fast parameters with γ_1, while results for the compact parameters with γ_3 are in red, which performs identical to the default parameters. On the right, the solid black line shows the maximum throughput based on 7.4 ns per random memory access per thread, and the solid black line shows the maximum throughput based on the total memory bandwidth of 25.6 GB/s.

prefetching around 16 iterations ahead. At this point, memory throughput is the bottleneck, and the difference between the compact and fast parameters disappears. In fact, compact parameters with γ_3 are slightly *faster*. This is because γ_3 has a more skew distribution of bucket sizes with more large buckets. When the pilots for these large buckets are cached, they are more likely to be hit by subsequent queries, and hence avoid some accesses to main memory.

For further experiments we choose streaming over batching, and use a lookahead of 32 iterations. The final throughput of 8 ns per query is very close to the optimal throughput of 7.4 ns per random memory read.

12.A.3 Multi-threaded Throughput.

In Figure 12.7 we compare the throughput of the fast and compact parameters for multiple threads. When $n = 20 \cdot 10^6$ is small and the entire data structure fits in L3 cache, the scaling to multiple threads is nearly perfect. As expected, minimal perfect hashing (bright) tends to be slightly slower than perfect hashing (dimmed), but the difference is small. The fast γ_1 is faster than the compact γ_3, and streaming provides only a small benefit over a for loop. For large $n = 10^9$, all methods converge towards the limit imposed by the full RAM throughput of 25.6 GB/s. Streaming variants hit this starting at around 4 threads, and remain faster than the for loop. As before, the compact version is slightly faster because of its more efficient use of the caches, and is even slightly better than the maximum throughput of random reads to RAM. Minimal perfect hashing is only slightly slower than perfect hashing.

12.B Sharding

When the number of keys is large, say over 10^{10}, their 64-bit (or 128-bit) hashes may not all fit in memory at the same time, even though the final PtrHash data structure (the list of pilots) would fit. Thus, we can not simply sort all hashes in memory to partition them. Instead, we split the set of all n hashes into, say $s = \lceil n/2^{32} \rceil$ *shards* of $\approx 2^{32}$ elements each, where the i'th shard corresponds to hash values in $s_i := [2^{64} \cdot i/s, 2^{64} \cdot (i+1)/s)$. Then, shards are processed one at a time. The hashes in each shard are sorted and split into parts, after which the parts are constructed as usual. This way, the shards only play a role during construction, and the final constructed data structure is independent of which sharding strategy was used.

In-memory sharding. The first approach to sharding is to iterate over the set of keys s times. In the i'th iteration, all keys are hashed, and only those hashes in the corresponding interval s_i are stored and processed. This way, no disk space is needed for construction.

On-disk sharding. A drawback of the first approach is that keys are potentially hashed many times. This can be avoided by writing hashes to disk. Specifically, we can create one file per shard and append hashes to their corresponding file. These files are then read and processed one by one.

Hybrid sharding. A hybrid of the two approaches above only requires disk space for $D < s$ shards. This iterates and hashes the keys $\lceil s/D \rceil$ times, and in each iteration writes hashes for D shards to disk. Those are then processed one by one as before.

On-disk PtrHash. When the number of keys is so large that even the pilots do not fit in memory, they can also be stored to disk and read on-demand while querying. This is supported using ε-serde [VF24, FVZ24].

12.B.1 Evaluation

We tested the in-memory and hybrid sharding by constructing PtrHash with default parameters on $5 \cdot 10^{10}$ random integer keys on a laptop with only 64 GB of memory, using 6 cores in parallel. All 64-bit hashes would take 400 GB, so we use 24 shards of around 2^{31} keys, that each take 16 GB. The final data structure takes 2.40 bits/key, or 15 GB in total, and the peak memory usage is around 50 GB.

The in-memory strategy iterates through and hashes the integer keys 24 times, and takes 3098 seconds in total or 129 s per shard. Of this, 67 s (52%) is spent on hashing the keys, 14 s (11%) is spent sorting hashes into buckets, and 45 s (35%) is spent searching for pilots.

The hybrid strategy is allowed to use up to 128 GB of disk space, and thus writes hashes to disk in 3 batches of 8 shards at a time. This brings the total time down to 2494 s (19% faster), and uses 104 s per shard. Of this, an amortized 31 s (30%) per shard is spent writing hashes to disk, and 9 s (9%) is spent reading hashes from disk, which together is faster than the 67 s that was previously spent on hashing all keys.

13 Discussion

In this thesis, we have worked on optimizing algorithms and implementations for several problems in bioinformatics. These contributions fall into two categories: for some problems, we focused on achieving practical speedups by using highly efficient implementations of algorithms that are amenable to this. For other problems, we took a more theoretical approach, and tried to reach a linear time algorithm, for pairwise alignment, or to reach an optimal density minimizer scheme.

Building on an earlier observation of Paul Medvedev [Med23a], my main thesis is:

> Provably optimal software consists of two parts: a provably optimal algorithm, and a provably optimal implementation of this algorithm, given the hardware constraints. This can only be achieved through algorithm/implementation co-design, where hardware capabilities influence design choices in the algorithm.

13.1 Pairwise Alignment

We first looked at the problem of *pairwise alignment*, where the differences (mutations) between two biological sequences are to be found. We reviewed many early improvements to theoretical algorithms, and a number of techniques for implementing these algorithms efficiently.

Nearly all existing methods are based on some variant of dynamic programming. In A*PA (Chapter 3), we use the A* shortest path algorithm, which is a graph-based method instead. This allows us to use a heuristic that can quickly and closely "predict" the edit distance in many cases. We additionally introduced *pruning*, which dynamically improves the heuristic as the A* search progresses, thereby leading to near-linear runtime. To my knowledge, this is the first heuristic of this type, and this same technique may have wider applications, such as in classic navigation software.

As it turns out, even though A*PA has near-linear complexity, the constant overhead is large: each visited state requires a memory access. This makes the method completely impractical whenever the scaling is super-linear, for example due to noisy regions or gaps in the alignment. Thus, in A*PA2 (Chapter 4), we revert back to a DP-based method, and we incorporate the A* heuristic into the classic band-doubling algorithm. Alongside additional optimizations, this yields up to 19× speedup over previous methods.

A lesson here is that a lot of time was spent on optimizing A*PA, even though this an inherently slow algorithm. In hindsight, it would have been more efficient to not try too many hacky optimizations, and instead shift focus towards the inherently faster DP-based methods earlier.

In Chapter 5, a start has been made to extend the aligner to both semi-global alignment and to affine costs, but a large part of this remains as future work.

13.2 Low Density Minimizers

We then looked at *minimizer schemes* (Chapter 6), which are used to sub-sample the k-mers of a genomic sequence as a form of compressing the sequence. The constraint is that at least one k-mer must be sampled every w positions, and the goal is to minimize the fraction (*density*) of sampled k-mers.

We were able to answer a number of open questions in this field. We proved a near-tight lower bound (Chapter 7) that is the first to show that the density is at least $2/(w+1)$ when $k = 1$, and generally this new bound is near-tight as $k \to \infty$. Alongside this, we introduced the mod-minimizer (Chapter 8), which matches the scaling of the lower bound, making this the first near-optimal scheme for large k.

Open problems. We also started the exploration of optimal schemes for $k = 1$ (Chapter 9), and introduced the *anti-lexicographic sus-anchor*, which is nearly optimal in practice. However, it is not quite theoretically optimal, and improving this remains an interesting open problem. Similarly, experiments suggest that perfectly optimal schemes exist for $k = w + 1$, but also here no general construction has been found so far. On the other hand, for $1 < k \leq w$, our lower bound appears to not be tight, and it would be interesting to improve it.

Lastly, our analysis focused mostly on *forward* schemes. *Local* schemes are a more general class of schemes that break our lower bound on forward scheme density. In practice, though, they are only marginally better, and it remains an open problem to prove this.

13.3 High Throughput Bioinformatics

Lastly, we optimized two specific applications in bioinformatics to achieve high throughput. In the case of PtrHash (Chapter 12), we were able to achieve throughput within 10% of what the hardware is capable of, and up to nearly 5× faster than the second fastest alternative. In the cases of both A*PA2 (Chapter 4) and `simd-minimizers` (Chapter 11), we were able to achieve on the order of 10× speedups over previous implementations. In all these cases, this was achieved by designing the algorithm with the implementation in mind, and by optimizing the implementation to fully utilize the capabilities of modern CPUs.

The implementation matters. Concluding, it seems inconsistent that so many papers start by stating the need for faster algorithms, but then never discuss implementation details. We reached 10× speedups on multiple applications by closely considering the implementation. On the other hand, many papers introduce new algorithmic techniques that yield significantly smaller speedups. Thus, this raises the suggestion that more attention should be given to the implementation of methods, rather than just the high level algorithm.

13.4 Propositions

I will end this thesis with a number of opinionated *propositions*.

1. Complexity theory's days are numbered.
2. $\log \log n \leq 6$
3. Succinct data structures are cute, but it's better to use some more space and not be terribly slow.

4. There is beauty in chasing mathematical perfection.
5. Too many PhDs are wasted shaving of small factors of complexities that will never be practical.
6. It is a fallacy to open a paper with "there is too much data, faster methods are needed" and then not say a word about optimizing code for modern hardware.
7. Fast code must exploit all assumptions on the input.
8. Fast code puts requirements on the input format, and the input has to adjust.
9. Optimizing ugly code is a waste of time – prettier methods will replace it.
10. Flat, unstructured text should be avoided at all costs. We research text indices, so index the text you write.
11. Assembly is not scary.

Bibliography

ABP25 Jarno N. Alanko, Elena Biagi, and Simon J. Puglisi, *Finimizers: Variable-length bounded-frequency minimizers for k-mer sets*, IEEE Transactions on Computational Biology and Bioinformatics **22** (2025), no. 2, 899–910.

ABW15 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams, *Tight hardness results for lcs and other sequence similarity measures*, 2015 IEEE 56th Annual Symposium on Foundations of Computer Science, IEEE, October 2015, p. 59–78.

ACG95 Bowen Alpern, Larry Carter, and Kang Su Gatlin, *Microparallelism and high-performance protein matching*, Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '95 (1995).

ADKF70 V. L. Arlazarov, Y. A. Dinitz, M. A. Kronrod, and I. A. Faradzhev, *On economical construction of the transitive closure of an oriented graph*, Dokl. Akad. Nauk SSSR **194** (1970), no. 3, 487–488.

AE86 Stephen F. Altschul and Bruce W. Erickson, *Optimal sequence alignment using affine gap costs*, Bulletin of Mathematical Biology **48** (1986), no. 5–6, 603–616.

AFGK+25 Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P. Pissis, *U-Index: A Universal Indexing Framework for Matching Long Patterns*, SEA 2025, LIPIcs, vol. 338, 2025, pp. 4:1–4:18.

AGM+90 Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology **215** (1990), no. 3, 403–410.

AKBP24 Anna Abramova, Antti Karkman, and Johan Bengtsson-Palme, *Metagenomic assemblies tend to break around antibiotic resistance genes*, BMC Genomics **25** (2024), no. 1.

ALP23 Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis, *Text indexing for long patterns: Anchors are all you need*, Proceedings of the VLDB Endowment **16** (2023), no. 9, 2117–2131.

ARD+21 Mohammed Alser, Jeremy Rotman, Dhrithi Deshpande, Kodi Taraszka, Huwenbo Shi, Pelin Icer Baykal, Harry Taegyun Yang, Victor Xue, Sergey Knyazev, Benjamin D. Singer, Brunilda Balliu, David Koslicki, Pavel Skums, Alex Zelikovsky, Can Alkan, Onur Mutlu, and Serghei Mangul, *Technology dictates algorithms: recent developments in read alignment*, Genome Biology **22** (2021), no. 1.

Bac78 John Backus, *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*, Communications of the ACM **21** (1978), no. 8, 613–641.

Bak24 Denis Bakhvalov, *Performance analysis and tuning on modern CPUs*, 2024.

BBD09 Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009, Springer Berlin Heidelberg, 2009, p. 682–693.

BCG+25 Ruben Becker, Davide Cenzato, Travis Gagie, Ragnar Groot Koerkamp, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza, *Compressing suffix trees by path decompositions*, arXiv, 2025.

BDH+19 Rory Bowden, Robert W Davies, Andreas Heger, Alistair T Pagnamenta, Mariateresa de Cesare, Laura E Oikkonen, Duncan Parkes, Colin Freeman, Fatima Dhalla, Smita Y Patel, et al., *Sequencing of human genomes with nanopore technology*, Nature communications **10** (2019), no. 1, 1–9.

Bel23 Piotr Beling, *Fingerprinting-based minimal perfect hashing revisited*, ACM Journal of Experimental Algorithmics **28** (2023), 1–16.

BFC08 Philip Bille and Martin Farach-Colton, *Fast and compact regular expression matching*, Theoretical Computer Science **409** (2008), no. 3, 486–496.

BGK25 Rick Beeloo and Ragnar Groot Koerkamp, *Sassy: Searching short DNA strings in the 2020s.*

BHL13 Gary Benson, Yozen Hernandez, and Joshua Loving, *A bit-parallel, general integer-scoring sequence alignment algorithm*, Lecture Notes in Computer Science (2013), 50–61.

BHR L. Bergroth, H. Hakonen, and T. Raita, *A survey of longest common subsequence algorithms*, Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, SPIRE-00, IEEE Comput. Soc, p. 39–48.

BI18 Arturs Backurs and Piotr Indyk, *Edit distance cannot be computed in strongly subquadratic time (unless SETH is false)*, SIAM Journal on Computing **47** (2018), no. 3, 1087–1097.

BKLS23 Dominik Bez, Florian Kurpicz, Hans-Peter Lehmann, and Peter Sanders, *High Performance Construction of RecSplit Based Minimal Perfect Hash Functions*, ESA (Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, eds.), vol. 274, 2023, pp. 19:1–19:16.

Bla15 Simon R. Blackburn, *Non-overlapping codes*, IEEE Transactions on Information Theory **61** (2015), no. 9, 4890–4894.

BLS13 Gary Benson, Avivit Levy, and B. Riva Shalom, *Longest common subsequence in k length substrings*, p. 257–265, Springer Berlin Heidelberg, 2013.

BN14 Djamal Belazzougui and Gonzalo Navarro, *Alphabet-independent compressed text indexing*, ACM Transactions on Algorithms **10** (2014), no. 4, 1–19.

Bre Christopher Breeden, *fxhash: A fast, non-secure, hashing algorithm derived from an internal hasher in FireFox.*, `https://crates.io/crates/fxhash`.

BRJ+24 Gaëtan Benoit, Sébastien Raguideau, Robert James, Adam M Phillippy, Rayan Chikhi, and Christopher Quince, *High-quality metagenome assembly from long accurate reads with metaMDBG*, Nature Biotechnology (2024), 1–6.

Bro97 A.Z. Broder, *On the resemblance and containment of documents*, Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171) (1997).

BS25 Piotr Beling and Peter Sanders, *Phast – perfect hashing with fast evaluation*, arXiv, 2025.

BYG92 Ricardo Baeza-Yates and Gaston H. Gonnet, *A new approach to text searching*, Communications of the ACM **35** (1992), no. 10, 74–82.

BYN96 Ricardo Baeza-Yates and Gonzalo Navarro, *A faster algorithm for approximate string matching*, Lecture Notes in Computer Science (1996), 1–23.

Cha05 C.-C. Chang, *Perfect hashing schemes for mining association rules*, The Computer Journal **48** (2005), no. 2, 168–179.

CL92 William I. Chang and Jordan Lampe, *Theoretical and empirical comparisons of approximate string matching algorithms*, Lecture Notes in Computer Science (1992), 175–184.

CLJ+14 Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev, *On the representation of De Bruijn graphs*, RECOMB, Springer International Publishing, 2014, p. 35–55.

Coh97 Jonathan D. Cohen, *Recursive hashing functions for n-grams*, ACM Trans. Inf. Syst. **15** (1997), no. 3, 291–320.

CT23 Andrea Cracco and Alexandru I. Tomescu, *Extremely fast construction and querying of compacted and colored De Bruijn graphs with GGCAT*, Genome Research (2023).

Dai16 Jeff Daily, *Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments*, BMC Bioinformatics **17** (2016), no. 1.

DB46 Nicolaas Govert De Bruijn, *A combinatorial problem*, Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam **49** (1946), no. 7, 758–764.

DG14 Sebastian Deorowicz and Szymon Grabowski, *Efficient algorithms for the longest common subsequence in k-length substrings*, Information Processing Letters **114** (2014), no. 11, 634–638.

DGKM19 Dan DeBlasio, Fiyinfoluwa Gbosibo, Carl Kingsford, and Guillaume Marçais, *Practical universal k-mer sets for minimizer schemes*, Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, 2019, pp. 167–176.

Dij59 Edsger W Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), no. 1, 269–271.

DKF+99 A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg, *Alignment of whole genomes*, Nucleic Acids Research **27** (1999), no. 11, 2369–2376.

DKGDG15 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz, *KMC 2: fast and resource-frugal k-mer counting*, Bioinformatics **31** (2015), no. 10, 1569–1576.

Dow17 Travis Downs, *Measuring reorder buffer capacity on Skylake*, `https://www.realworldtech.com/forum/?threadid=166772&curpostid=167685`, April 2017.

Dow18 ———, *Investigating line fill buffers*, `https://github.com/Kobzol/hardware-effects/issues/1#issuecomment-441111396`, November 2018.

Dre07 Ulrich Drepper, *What every programmer should know about memory*.

DWRR08 Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert, *Seqan: An efficient, generic C++ library for sequence analysis*, BMC Bioinformatics **9** (2008), no. 1.

EBC21 Barış Ekim, Bonnie Berger, and Rayan Chikhi, *Minimizer-space De Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer*, Cell Systems **12** (2021), no. 10, 958–968.e6.

EBO20 Barış Ekim, Bonnie Berger, and Yaron Orenstein, *A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets*, RECOMB, Springer International Publishing, 2020, p. 37–53.

Edg21 Robert Edgar, *Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences*, PeerJ **9** (2021), e10805.

EGGI92a David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano, *Sparse dynamic programming i: linear cost functions*, Journal of the ACM **39** (1992), no. 3, 519–545.

EGGI92b ———, *Sparse dynamic programming ii: convex and concave cost functions*, Journal of the ACM **39** (1992), no. 3, 546–567.

EGV20 Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna, *RecSplit: Minimal perfect hashing via recursive splitting*, 2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics, January 2020, p. 175–185.

Eli74 Peter Elias, *Efficient storage and retrieval by content and address of static files*, Journal of the ACM **21** (1974), no. 2, 246–260.

EP22 Jordan M. Eizenga and Benedict Paten, *Improving the time and space complexity of the WFA algorithm and generalizing its scoring*.

ESM+23 Barış Ekim, Kristoffer Sahlin, Paul Medvedev, Bonnie Berger, and Rayan Chikhi, *Efficient mapping of accurate long reads in minimizer space with mapquik*, Genome Research (2023).

FAKM14 Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher, *Cuckoo filter*, Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (2014).

Fan71 R.M. Fano, *On the number of bits required to implement an associative memory*, Memorandum 61, Computation Structures Group, MIT Project MAC Computer Structures Group, 1971.

Far06 Michael Farrar, *Striped Smith–Waterman speeds database searches six times over other SIMD implementations*, Bioinformatics **23** (2006), no. 2, 156–161.

FCH92 Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath, *A faster algorithm for constructing minimal perfect hash functions*, Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '92 (1992).

Fic84 James W. Fickett, *Fast optimal alignment*, Nucleic Acids Research **12** (1984), no. 1Part1, 175–179.

FN07 Paolo Ferragina and Gonzalo Navarro, *The Pizza&Chili Corpus*, `https://pizzachili.dcc.uchile.cl/texts.html`, 2007.

FNK20 Martin C Frith, Laurent Noé, and Gregory Kucherov, *Minimally overlapping words for sequence similarity search*, Bioinformatics **36** (2020), no. 22–23, 5344–5350.

Fog24 Agner Fog, *Software optimization resources*, `https://www.agner.org/optimize/`, 2024.

FPSS04 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis, *Space efficient hash tables with worst case constant access time*, Theory of Computing Systems **38** (2004), no. 2, 229–248.

Fri15 J.T. Frielingsdorf, *Improving optimal sequence alignments through a SIMD-accelerated library*, Master's thesis, University of Oslo, 2015.

FVZ24 Tommaso Fontana, Sebastiano Vigna, and Stefano Zacchiroli, *Webgraph: The next generation (is in rust)*, Companion Proceedings of the ACM Web Conference 2024, WWW '24, ACM, May 2024, p. 686–689.

Gin23 Olivier Giniaux, *Gxhash: A high-throughput, non-cryptographic hashing algorithm leveraging modern CPU capabilities*, 2023.

GK24 Ragnar Groot Koerkamp, *A*PA2: Up to 19× Faster Exact Global Alignment*, WABI 2024, LIPIcs, vol. 312, 2024, pp. 17:1–17:25.

GK25 ———, *PtrHash: Minimal Perfect Hashing at RAM Throughput*, SEA 2025, LIPIcs, vol. 338, 2025, pp. 21:1–21:21.

GKdV24 Ragnar Groot Koerkamp and Mees de Vries, *PACE solver description: OCMu64, a solver for one-sided crossing minimization*, IPEC, 2024.

GKI24 Ragnar Groot Koerkamp and Pesho Ivanov, *Exact global alignment using A* with chaining seed heuristic and match pruning*, Bioinformatics **40** (2024), no. 3.

GKLP25 Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri, *The open-closed mod-minimizer algorithm*, Algorithms for Molecular Biology **20** (2025), no. 4.

GKM25 Ragnar Groot Koerkamp and Igor Martayan, *SimdMinimizers: Computing Random Minimizers, fast*, SEA 2025, LIPIcs, vol. 338, 2025, pp. 20:1–20:19.

GKP24 Ragnar Groot Koerkamp and Giulio Ermanno Pibiri, *The mod-minimizer: A simple and efficient sampling algorithm for long k-mers*, WABI 2024, LIPIcs, vol. 312, 2024, pp. 11:1–11:23.

GKvdW19 Ragnar Groot Koerkamp and Marieke van der Wegen, *Stable gonality is computable*, Discrete Mathematics & Theoretical Computer Science **vol. 21 no. 1, ICGT 2018** (2019).

GKŽ21 Ragnar Groot Koerkamp and Stanislav Živný, *On rainbow-free colourings of uniform hypergraphs*, Theoretical Computer Science **885** (2021), 69–76.

Got82 Osamu Gotoh, *An improved algorithm for matching biological sequences*, Journal of Molecular Biology **162** (1982), no. 3, 705–708.

Got90 ———, *Optimal sequence alignment allowing for long gaps*, Bulletin of Mathematical Biology **52** (1990), no. 3, 359–373.

GR17 Szymon Grabowski and Marcin Raniszewski, *Sampled suffix array with minimizers*, Software: Practice and Experience **47** (2017), no. 11, 1755–1771.

Gra16 Szymon Grabowski, *New tabulation and sparse dynamic programming based techniques for sequence similarity problems*, Discrete Applied Mathematics **212** (2016), 96–103.

GS25 Shay Golan and Arseny M. Shur, *Expected density of random minimizers*, SOFSEM 2025: Theory and Practice of Computer Science, Springer Nature Switzerland, 2025, p. 347–360.

GTK⁺25 Shay Golan, Ido Tziony, Matan Kraus, Yaron Orenstein, and Arseny Shur, *GreedyMini: generating low-density DNA minimizers*, Bioinformatics **41** (2025), 275–284.

Had88 Frank O. Hadlock, *Minimum detour methods for string or sequence comparison*, Congressus Numerantium **61** (1988), 263–274.

Ham50 R. W. Hamming, *Error detecting and error correcting codes*, Bell System Technical Journal **29** (1950), no. 2, 147–160.

Her23 Stefan Hermann, *Accelerating minimal perfect hash function construction using GPU parallelization*, Master's thesis, Karlsruher Institut für Technologie (KIT), 2023.

HFN05 Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro, *Increased bit-parallelism for approximate and multiple string matching*, ACM Journal of Experimental Algorithmics **10** (2005).

Hir75 Daniel S. Hirschberg, *A linear space algorithm for computing maximal common subsequences*, Communications of the ACM **18** (1975), no. 6, 341–343.

Hir77 ———, *Algorithms for the longest common subsequence problem*, Journal of the ACM **24** (1977), no. 4, 664–675.

HLP⁺24a Stefan Hermann, Hans-Peter Lehmann, Giulio Ermanno Pibiri, Peter Sanders, and Stefan Walzer, *PHOBIC: Perfect hashing with optimized bucket sizes and interleaved coding*, ESA, 2024.

HLP⁺24b ———, *PHOBIC: Perfect hashing with optimized bucket sizes and interleaved coding*, arXiv, 2024.

HLR⁺21 Peter W Harrison, Rodrigo Lopez, Nadim Rahman, Stefan Gutnick Allen, Raheela Aslam, Nicola Buso, Carla Cummins, Yasmin Fathy, Eloy Felix, Mihai Glont, Suran Jayathilaka, Sandeep Kadam, Manish Kumar, Katharina B Lauer, Geetika Malhotra, Abayomi Mosaku, Ossama Edbali, Young Mi Park, Andrew Parton, Matt Pearce, Jose Francisco Estrada Pena, Joseph Rossetto, Craig Russell, Sandeep Selvakumar, Xènia Pérez Sitjà, Alexey Sokolov, Ross Thorne, Marianna Ventouratou, Peter Walter, Galabina Yordanova, Amonida Zadissa, Guy Cochrane, Niklas Blomberg, and Rolf Apweiler, *The COVID-19 data portal: accelerating SARS-CoV-2 and COVID-19 research through rapid open access data sharing*, Nucleic Acids Research **49** (2021), no. W1, W619–W623.

HM20 Guillaume Holley and Páll Melsted, *Bifrost: highly parallel construction and indexing of colored and compacted De Bruijn graphs*, Genome Biology **21** (2020), no. 1.

HNR68 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), no. 2, 100–107.

HNR72 ———, *Correction to "a formal basis for the heuristic determination of minimum cost paths"*, ACM SIGART Bulletin (1972), no. 37, 28–29.

Hol10 Robert C. Holte, *Common misconceptions concerning heuristic search*, Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010 (Ariel Felner and Nathan R. Sturtevant, eds.), AAAI Press, 2010.

HS77 James W. Hunt and Thomas G. Szymanski, *A fast algorithm for computing longest common subsequences*, Communications of the ACM **20** (1977), no. 5, 350–353.

HST17 Martin Hirzel, Scott Schneider, and Kanat Tangwongsan, *Sliding-window aggregation algorithms: Tutorial*, Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017, ACM, 2017, pp. 11–14.

Hug96 Richard Hughey, *Parallel hardware for sequence comparison and alignment*, Bioinformatics **12** (1996), no. 6, 473–479.

HZK22 Minh Hoang, Hongyu Zheng, and Carl Kingsford, *Differentiable learning of sequence-specific minimizer schemes with DeepMinimizer*, Journal of Computational Biology **29** (2022), no. 12, 1288–1304.

IBM⁺20 Pesho Ivanov, Benjamin Bichsel, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin Vechev, *Astarix: Fast and optimal sequence-to-graph alignment*, RECOMB, Springer International Publishing, 2020, p. 104–119.

IBR⁺22 Luiz Irber, Phillip T Brooks, Taylor Reiter, N Tessa Pierce-Ward, Mahmudur Rahman Hera, David Koslicki, and C Titus Brown, *Lightweight compositional analysis of metagenomes with FracMinHash and minimum metagenome covers*, BioRxiv (2022), 2022–01.

IBV22 Pesho Ivanov, Benjamin Bichsel, and Martin Vechev, *Fast and optimal sequence-to-graph alignment guided by seeds*, RECOMB, Springer International Publishing, 2022, p. 306–325.

JGT22 Chirag Jain, Daniel Gibney, and Sharma V. Thankachan, *Algorithms for colinear chaining with overlaps and gap costs*, Journal of Computational Biology **29** (2022), no. 11, 1237–1251.

JRH+22 Chirag Jain, Arang Rhie, Nancy F Hansen, Sergey Koren, and Adam M Phillippy, *Long-read mapping to repetitive reference sequences using Winnowmap2*, Nature Methods **19** (2022), no. 6, 705–710.

KGKM+24 Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J Treangen, *A near-tight lower bound on the density of forward sampling schemes*, Bioinformatics (2024).

KGTP23 Bryce Kille, Erik Garrison, Todd J Treangen, and Adam M Phillippy, *Minmers are a generalization of minimizers that enable unbiased local Jaccard estimation*, Bioinformatics **39** (2023), no. 9.

KL06 Sven Koenig and Maxim Likhachev, *Real-time adaptive A**, Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, 2006, pp. 281–288.

KR87 Richard M. Karp and Michael O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), no. 2, 249–260.

Kru83 Joseph B. Kruskal, *An overview of sequence comparison: Time warps, string edits, and macromolecules*, SIAM Review **25** (1983), no. 2, 201–237.

KWN+22 Parham Kazemi, Johnathan Wong, Vladimir Nikolić, Hamid Mohamadi, René L Warren, and Inanç Birol, *ntHash2: recursive spaced seed hashing for nucleotide sequences*, Bioinformatics **38** (2022), no. 20, 4812–4813.

Leh24 Hans-Peter Lehmann, *Fast and space-efficient perfect hashing*, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2024.

Leh25 ———, *MPHF-Experiments*, `https://github.com/ByteHamster/MPHF-Experiments`, January 2025.

Lem17 Daniel Lemire, *Removing duplicates from lists quickly*, `https://lemire.me/blog/2017/04/10/removing-duplicates-from-lists-quickly/`, April 2017.

Lem19 ———, *Fast random integer generation in an interval*, ACM Transactions on Modeling and Computer Simulation **29** (2019), no. 1, 1–12.

LHB14 Joshua Loving, Yozen Hernandez, and Gary Benson, *BitPAl: a bit-parallel, general integer-scoring sequence alignment algorithm*, Bioinformatics **30** (2014), no. 22, 3166–3173.

Li16 Heng Li, *Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences*, Bioinformatics **32** (2016), no. 14, 2103–2110.

Li18 ———, *Minimap2: pairwise alignment for nucleotide sequences*, Bioinformatics **34** (2018), no. 18, 3094–3100.

Liu23 Daniel Liu, *minimizer.rs re-scan implementation*, `https://gist.github.com/Daniel-Liu-c0deb0t/7078ebca04569068f15507aa856be6e8`, July 2023.

LKK19 Daniel Lemire, Owen Kaser, and Nathan Kurz, *Faster remainder by direct computation: Applications to compilers and software libraries*, Software: Practice and Experience **49** (2019), no. 6, 953–970.

LP21 Grigorios Loukides and Solon P. Pissis, *Bidirectional string anchors: A new string sampling mechanism*, ESA, 2021.

LPB06 Yi Lu, Balaji Prabhakar, and Flavio Bonomi, *Perfect hashing for network applications*, 2006 IEEE International Symposium on Information Theory, IEEE, July 2006.

LPS23 Grigorios Loukides, Solon P. Pissis, and Michelle Sweering, *Bidirectional string anchors for improved text indexing and top-k similarity search*, IEEE Transactions on Knowledge and Data Engineering **35** (2023), no. 11, 11093–11111.

LS23 Daniel Liu and Martin Steinegger, *Block Aligner: an adaptive SIMD-accelerated aligner for sequences and position-specific scoring matrices*, Bioinformatics (2023).

LSW23a Hans-Peter Lehmann, Peter Sanders, and Stefan Walzer, *ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force*, 2023.

LSW23b _____, *SicHash - small irregular Cuckoo tables for perfect hashing*, 2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics, January 2023, p. 176–189.

LSW24 _____, *ShockHash: Towards optimal-space minimal perfect hashing beyond brute-force*, 2024 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics, January 2024, p. 194–206.

LSWZ25 Hans-Peter Lehmann, Peter Sanders, Stefan Walzer, and Jonatan Ziegler, *Combined search and encoding for seeds, with an application to minimal perfect hashing*, arXiv, 2025.

LV89 Gad M Landau and Uzi Vishkin, *Fast parallel and serial approximate string matching*, Journal of Algorithms **10** (1989), no. 2, 157–169.

Maj96 B. S. Majewski, *A family of perfect hashing methods*, The Computer Journal **39** (1996), no. 6, 547–554.

Mar Igor Martayan, *minimizer-iter: Iterate over minimizers of a DNA sequence.*, `https://github.com/rust-seq/minimizer-iter`.

MCVB16 Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol, *ntHash: recursive nucleotide hashing*, Bioinformatics **32** (2016), no. 22, 3492–3494.

MDK18 Guillaume Marçais, Dan DeBlasio, and Carl Kingsford, *Asymptotically optimal minimizers schemes*, Bioinformatics **34** (2018), no. 13, i13–i22.

MDP+18 Guillaume Marçais, Arthur L. Delcher, Adam M. Phillippy, Rachel Coston, Steven L. Salzberg, and Aleksey Zimin, *Mummer4: A fast and versatile genome alignment system*, PLOS Computational Biology **14** (2018), no. 1, e1005944.

Med23a Paul Medvedev, *Theoretical analysis of edit distance algorithms*, Communications of the ACM **66** (2023), no. 12, 64–71.

Med23b _____, *Theoretical analysis of sequencing bioinformatics algorithms and beyond*, Communications of the ACM **66** (2023), no. 7, 118–125.

Meh82 Kurt Mehlhorn, *On the program size of perfect and universal hash functions*, 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982) (1982).

MEK24 Guillaume Marçais, C S Elder, and Carl Kingsford, *k-nonical space: Sketching with reverse complements*, Bioinformatics (2024).

MKL21 Camille Marchet, Mael Kerbiriou, and Antoine Limasset, *Blight: efficient exact associative structure for k-mers*, Bioinformatics **37** (2021), no. 18, 2858–2865.

MM88 Gene Myers and Webb Miller, *Optimal alignments in linear space*, Bioinformatics **4** (1988), no. 1, 11–17.

MM95 _____, *Chaining multiple-alignment fragments in sub-quadratic time*, Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (USA), SODA '95, Society for Industrial and Applied Mathematics, 1995, p. 38–47.

Moo06 Gordon E. Moore, *Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff.*, IEEE Solid-State Circuits Society Newsletter **11** (2006), no. 3, 33–35.

MP80 William J. Masek and Michael S. Paterson, *A faster algorithm computing string edit distances*, Journal of Computer and System Sciences **20** (1980), no. 1, 18–31.

MPB+17 Guillaume Marçais, David Pellow, Daniel Bork, Yaron Orenstein, Ron Shamir, and Carl Kingsford, *Improving the performance of minimizers and winnowing schemes*, Bioinformatics **33** (2017), no. 14, i110–i117.

MRSL25 Igor Martayan, Lucas Robidou, Yoshihiro Shibuya, and Antoine Limasset, *Hyper-k-mers: Efficient streaming k-mers representation*, RECOMB, Springer Nature Switzerland, 2025, p. 330–335.

MSEG+23 Santiago Marco-Sola, Jordan M Eizenga, Andrea Guarracino, Benedict Paten, Erik Garrison, and Miquel Moreto, *Optimal gap-affine alignment in $O(s)$ space*, Bioinformatics **39** (2023), no. 2.

MSMME21 Santiago Marco-Sola, Juan Carlos Moure, Miquel Moreto, and Antonio Espinosa, *Fast gap-affine pairwise alignment using the wavefront algorithm*, Bioinformatics **37** (2021), no. 4, 456–463.

MSSGR12 Santiago Marco-Sola, Michael Sammeth, Roderic Guigó, and Paolo Ribeca, *The GEM mapper: fast, accurate and versatile alignment by filtration*, Nature Methods **9** (2012), no. 12, 1185–1188.

Mye86 Gene Myers, *An $O(ND)$ difference algorithm and its variations*, Algorithmica **1** (1986), no. 1–4, 251–266.

Mye99 ———, *A fast bit-vector algorithm for approximate string matching based on dynamic programming*, Journal of the ACM **46** (1999), no. 3, 395–415.

Myk72 Johannes Mykkeltveit, *A proof of Golomb's conjecture for the De Bruijn graph*, Journal of Combinatorial Theory, Series B **13** (1972), no. 1, 40–45.

Nav01 Gonzalo Navarro, *A guided tour to approximate string matching*, ACM Computing Surveys **33** (2001), no. 1, 31–88.

NKR+22 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V. Bzikadze, Alla Mikheenko, Mitchell R. Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, Sergey Aganezov, Savannah J. Hoyt, Mark Diekhans, Glennis A. Logsdon, Michael Alonge, Stylianos E. Antonarakis, Matthew Borchers, Gerard G. Bouffard, Shelise Y. Brooks, Gina V. Caldas, Nae-Chyun Chen, Haoyu Cheng, Chen-Shan Chin, William Chow, Leonardo G. de Lima, Philip C. Dishuck, Richard Durbin, Tatiana Dvorkina, Ian T. Fiddes, Giulio Formenti, Robert S. Fulton, Arkarachai Fungtammasan, Erik Garrison, Patrick G. S. Grady, Tina A. Graves-Lindsay, Ira M. Hall, Nancy F. Hansen, Gabrielle A. Hartley, Marina Haukness, Kerstin Howe, Michael W. Hunkapiller, Chirag Jain, Miten Jain, Erich D. Jarvis, Peter Kerpedjiev, Melanie Kirsche, Mikhail Kolmogorov, Jonas Korlach, Milinn Kremitzki, Heng Li, Valerie V. Maduro, Tobias Marschall, Ann M. McCartney, Jennifer McDaniel, Danny E. Miller, James C. Mullikin, Gene Myers, Nathan D. Olson, Benedict Paten, Paul Peluso, Pavel A. Pevzner, David Porubsky, Tamara Potapova, Evgeny I. Rogaev, Jeffrey A. Rosenfeld, Steven L. Salzberg, Valerie A. Schneider, Fritz J. Sedlazeck, Kishwar Shafin, Colin J. Shew, Alaina Shumate, Ying Sims, Arian F. A. Smit, Daniela C. Soto, Ivan Sović, Jessica M. Storer, Aaron Streets, Beth A. Sullivan, Françoise Thibaud-Nissen, James Torrance, Justin Wagner, Brian P. Walenz, Aaron Wenger, Jonathan M. D. Wood, Chunlin Xiao, Stephanie M. Yan, Alice C. Young, Samantha Zarate, Urvashi Surti, Rajiv C. McCoy, Megan Y. Dennis, Ivan A. Alexandrov, Jennifer L. Gerton, Rachel J. O'Neill, Winston Timp, Justin M. Zook, Michael C. Schatz, Evan E. Eichler, Karen H. Miga, and Adam M. Phillippy, *The complete sequence of a human genome*, Science **376** (2022), no. 6588, 44–53.

NKY82 Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima, *A longest common subsequence algorithm suitable for similar text strings*, Acta Informatica **18** (1982), no. 2.

NPBnF+24 Malick Ndiaye, Silvia Prieto-Baños, Lucy M. Fitzgerald, Ali Yazdizadeh Kharrazi, Sergey Oreshkov, Christophe Dessimoz, Fritz J. Sedlazeck, Natasha Glover, and Sina Majidian, *When less is more: sketching with minimizers in genomics*, Genome Biology **25** (2024), no. 1.

NW70 Saul B. Needleman and Christian D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology **48** (1970), no. 3, 443–453.

OPM+16 Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford, *Compact universal k-mer hitting sets*, Algorithms in Bioinformatics, Springer International Publishing, 2016, p. 257–268.

OPM+17 ———, *Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing*, PLOS Computational Biology **13** (2017), no. 10, e1005777.

Pag99 Rasmus Pagh, *Hash and displace: Efficient evaluation of minimal perfect hash functions*, Algorithms and Data Structures, Springer Berlin Heidelberg, 1999, p. 49–54.

PBJ17 Prashant Pandey, Michael A. Bender, and Rob Johnson, *A fast x86 implementation of select*, arXiv, 2017.

Pea84 Judea Pearl, *Heuristics: intelligent search strategies for computer problem solving*, Addison-Wesley Longman Publishing Co., Inc., 1984.

Pib22 Giulio Ermanno Pibiri, *Sparse and skew hashing of k-mers*, Bioinformatics **38** (2022), i185–i194.

PKM+17 Filip Pavetić, Ivan Katanić, Gustav Matula, Goran Žužić, and Mile Šikić, *Fast and simple algorithms for computing both lcs_k and LCS_{k+}*, arXiv, 2017.

PM17 David L Poole and Alan K Mackworth, *Artificial intelligence: Foundations of computational agents*, second ed., Cambridge University Press, 2017.

PP09 Dimitris Papamichail and Georgios Papamichail, *Improved algorithms for approximate string matching (extended abstract)*, BMC Bioinformatics **10** (2009), no. S1.

PPE+23 David Pellow, Lianrong Pu, Bariş Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein, *Efficient minimizer orders for large values of k using minimum decycling sets*, Genome Research (2023).

PR01 Rasmus Pagh and Flemming Friche Rodler, *Cuckoo hashing*, Algorithms — ESA 2001, Springer Berlin Heidelberg, 2001, p. 121–133.

PR24 Chenxu Pan and Knut Reinert, *A simple refined DNA minimizer operator enables twofold faster computation*, Bioinformatics (2024).

PSL23 Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset, *Locality-preserving minimal perfect hashing of k-mers*, Bioinformatics **39** (2023), i534–i543.

PT21 Giulio Ermanno Pibiri and Roberto Trani, *PTHash: Revisiting FCH minimal perfect hashing*, Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (2021).

PT24 ———, *Parallel and external-memory construction of minimal perfect hash functions with PTHash*, IEEE Transactions on Knowledge and Data Engineering **36** (2024), no. 3, 1249–1259.

PŽŠ14 Filip Pavetić, Goran Žužić, and Mile Šikić, *LCSk++: Practical similarity metric for long strings*.

RHH+04 Michael Roberts, Wayne Hayes, Brian R. Hunt, Stephen M. Mount, and James A. Yorke, *Reducing storage requirements for biological sequence comparison*, Bioinformatics **20** (2004), no. 18, 3363–3369.

Rog11 Torbjørn Rognes, *Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation*, BMC Bioinformatics **12** (2011), no. 1.

RS00 Torbjørn Rognes and Erling Seeberg, *Six-fold speed-up of Smith–Waterman sequence database searches using parallel processing on common microprocessors*, Bioinformatics **16** (2000), no. 8, 699–706.

San72 David Sankoff, *Matching sequences under deletion/insertion constraints*, Proceedings of the National Academy of Sciences **69** (1972), no. 1, 4–6.

Sel74 Peter H. Sellers, *On the theory and computation of evolutionary distances*, SIAM Journal on Applied Mathematics **26** (1974), no. 4, 787–793.

SK18 Hajime Suzuki and Masahiro Kasahara, *Introducing difference recurrence relations for faster semi-global alignment of long sequences*, BMC Bioinformatics **19** (2018), no. S1.

SLKD08 Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz, *SWPS3 – fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2*, BMC Research Notes **1** (2008), no. 1, 107.

Šoš15 Martin Šošić, *An SIMD Dynamic Programming C/C++ Library*, Master's thesis, University of Zagreb, 2015.

Spo89 John L. Spouge, *Speeding up dynamic programming algorithms for finding optimal lattice paths*, SIAM Journal on Applied Mathematics **49** (1989), no. 5, 1552–1566.

Spo91 ———, *Fast optimal alignment*, Bioinformatics **7** (1991), no. 1, 1–7.

SR24 Haojing Shao and Jue Ruan, *BSAlign: A library for nucleotide sequence alignment*, Genomics, Proteomics & Bioinformatics (2024).

ŠŠ17 Martin Šošić and Mile Šikić, *Edlib: a C/C++ library for fast, exact sequence alignment using edit distance*, Bioinformatics **33** (2017), no. 9, 1394–1395.

SW81 T.F. Smith and M.S. Waterman, *Identification of common molecular subsequences*, Journal of Molecular Biology **147** (1981), no. 1, 195–197.

SWA03 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken, *Winnowing: local algorithms for document fingerprinting*, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD/PODS03, ACM, June 2003.

SWF81 T. F. Smith, M. S. Waterman, and W. M. Fitch, *Comparative biosequence metrics*, Journal of Molecular Evolution **18** (1981), no. 1, 38–46.

SY21 Jim Shaw and Yun William Yu, *Theory of local k-mer selection with applications to long-read alignment*, Bioinformatics **38** (2021), no. 20, 4659–4669.

TKPP18 Georgios Theodorakis, Alexandros Koliousis, Peter R. Pietzuch, and Holger Pirk, *Hammer Slide: Work- and CPU-efficient streaming window aggregation*, International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018 (Rajesh Bordawekar and Tirthankar Lahiri, eds.), 2018, pp. 34–41.

Ukk83 Esko Ukkonen, *On approximate string matching*, Lecture Notes in Computer Science (1983), 487–495.

Ukk85a ———, *Algorithms for approximate string matching*, Information and Control **64** (1985), no. 1–3, 100–118.

Ukk85b ———, *Finding approximate patterns in strings*, Journal of Algorithms **6** (1985), no. 1, 132–137.

vDME+24 Lucas R. van Dijk, Abigail L. Manson, Ashlee M. Earl, Kiran V Garimella, and Thomas Abeel, *Fast and exact gap-affine partial order alignment with POASTA*, bioRxiv (2024).

VF24 Sebastiano Vigna and Tommaso Fontana, *ε-serde*, `https://github.com/vigna/epserde-rs`, 2024.

Vig24 Sebastiano Vigna, *sux-rs*, `https://github.com/vigna/sux-rs`, 2024.

Vig25 ———, *ε-cost sharding: Scaling hypergraph-based static functions and filters to trillions of keys*, 2025.

Vin68 T. K. Vintsyuk, *Speech discrimination by dynamic programming*, Cybernetics **4** (1968), no. 1, 52–57.

WF74 Robert A. Wagner and Michael J. Fischer, *The string-to-string correction problem*, Journal of the ACM **21** (1974), no. 1, 168–173.

Wit23 Roland Wittler, *General encoding of canonical k-mers*, Peer Community Journal **3** (2023).

WL83 W. John Wilbur and David J. Lipman, *Rapid similarity searches of nucleic acid and protein data banks.*, Proceedings of the National Academy of Sciences **80** (1983), no. 3, 726–730.

WL84 ———, *The context dependent comparison of biological sequences*, SIAM Journal on Applied Mathematics **44** (1984), no. 3, 557–567.

WLL19 Derrick E Wood, Jennifer Lu, and Ben Langmead, *Improved metagenomic analysis with Kraken 2*, Genome biology **20** (2019), 1–13.

WM92 Sun Wu and Udi Manber, *Fast text searching*, Communications of the ACM **35** (1992), no. 10, 83–91.

WMM96 Sun Wu, U. Manber, and Gene Myers, *A subquadratic algorithm for approximate limited expression matching*, Algorithmica **15** (1996), no. 1, 50–67.

WMMM90 Sun Wu, Udi Manber, Gene Myers, and Webb Miller, *An $O(NP)$ sequence comparison algorithm*, Information Processing Letters **35** (1990), no. 6, 317–323.

Won13 Henry Wong, *Measuring reorder buffer capacity*, `https://blog.stuffedcow.net/2013/05/measuring-rob-capacity/`, may 2013.

Woz97 A. Wozniak, *Using video-oriented instructions to speed up sequence comparison*, Bioinformatics **13** (1997), no. 2, 145–150.

WSB76 M.S Waterman, T.F Smith, and W.A Beyer, *Some biological sequence metrics*, Advances in Mathematics **20** (1976), no. 3, 367–387.

WYB+24 Sumit Walia, Cheng Ye, Arkid Bera, Dhruvi Lodhavia, and Yatish Turakhia, *TALCO: Tiling genome sequence alignment using convergence of traceback pointers*, 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA), IEEE, March 2024.

YBR Wang Yi and Diego Barrios Romero, *wyhash-rs, fast portable non-cryptographic hashing algorithm*, `https://github.com/eldruin/wyhash-rs`.

ZKM20 Hongyu Zheng, Carl Kingsford, and Guillaume Marçais, *Improved design and analysis of practical minimizers*, Bioinformatics **36** (2020), i119–i127.

ZKM21a ———, *Lower density selection schemes via small universal hitting sets with short remaining path length*, Journal of Computational Biology **28** (2021), no. 4, 395–409.

ZKM21b ———, *Sequence-specific minimizers via polar sets*, Bioinformatics **37** (2021), i187–i195.

ZLGM13 Mengyao Zhao, Wan-Ping Lee, Erik P. Garrison, and Gabor T. Marth, *SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications*, PLoS ONE **8** (2013), no. 12, e82138.

ZMK23 Hongyu Zheng, Guillaume Marçais, and Carl Kingsford, *Creating and using minimizer sketches in computational genomics*, Journal of Computational Biology **30** (2023), no. 12, 1251–1276.

ZSWM00 Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller, *A greedy algorithm for aligning DNA sequences*, Journal of Computational Biology **7** (2000), no. 1–2, 203–214.

Curriculum Vitae

Oct 2021 - April 2025	PhD at the Biomedical Informatics Lab at **ETH Zurich** Thesis: *Optimal Throughput Bioinformatics*
Nov 2017 - April 2020	Software engineer III at **Google Zurich**
Oct 2016 - Sept 2017	MSc. Mathematics and the Foundations of Computer Science at **University of Oxford** (distinction) Thesis: *On Rainbow Free Colourings of Uniform Hypergraphs*
Sept 2013 - July 2016	BSc. Mathematics at **Utrecht University** (cum laude) BSc. Physics and Astronomy at **Utrecht University** (cum laude) Minor Computer Sciences at **Utrecht University** Thesis: *Calculating the Stable Gonality of Finite Graphs*

Publications

Publications presented in this thesis:

- [GK25] Ragnar Groot Koerkamp, *PtrHash: Minimal Perfect Hashing at RAM Throughput*, SEA 2025.
- [GKM25] Ragnar Groot Koerkamp and Igor Martayan, *SimdMinimizers: Computing Random Minimizers, Fast*, SEA 2025.
- [GKLP25] Ragnar Groot Koerkamp, Daniel Liu, and Giulio Ermanno Pibiri, *The Open-Closed Mod-Minimizer Algorithm*, AMB 2025.
- [KGKM+24] Bryce Kille, Ragnar Groot Koerkamp, Drake McAdams, Alan Liu, and Todd J. Treangen, *A Near-Tight Lower Bound on the Density of Forward Sampling Schemes*, Bioinformatics 2024.
- [GKP24] Ragnar Groot Koerkamp and Giulio Ermanno Pibiri, *The mod-minimizer: A Simple and Efficient Sampling Algorithm for Long k-mers*, WABI 2024.
- [GK24] Ragnar Groot Koerkamp, *A*PA2: Up to* $19\times$ *Faster Exact Global Alignment*, WABI 2024.
- [GKI24] Ragnar Groot Koerkamp, Pesho Ivanov, *Exact Global Alignment using A* with Chaining Seed Heuristic and Match Pruning*, Bioinformatics 2024.

Further publications:

- [BGK25] Rick Beeloo and Ragnar Groot Koerkamp, *Sassy: Searching Short DNA Strings in the 2020s*, biorXiv 2025.
- [BCG+25] Ruben Becker, Davide Cenzato, Travis Gagie, Ragnar Groot Koerkamp, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza, *Compressing Suffix Trees by Path Decompositions*, arXiv 2025.
- [AFGK+25] Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, Solon P. Pissis, *U-index: A Universal Indexing Framework for Matching Long Patterns*, SEA 2025.
- [GKdV24] Ragnar Groot Koerkamp and Mees de Vries, *PACE Solver Description: OCMu64, a Solver for One-Sided Crossing Minimization*, IPEC 2024.
- [GKŽ21] Ragnar Groot Koerkamp and Stanislav Živný, *On Rainbow-Free Colourings of Uniform Hypergraphs*, Theoretical Computer Science 2021.
- [GKvdW19] Ragnar Groot Koerkamp and Marieke van der Wegen, *Stable Gonality Is Computable.* Discrete Mathematics & Theoretical Computer Science 2018.