Efficient Optimization of Fixed-Length Paths

Martino Ciaperoni[1] , **Nikolaos Tziavelis**[2] and **Panagiotis Karras**[3]
[1]Scuola Normale Superiore, Pisa, Italy
[2]UC Santa Cruz, USA
[3]University of Copenhagen, Denmark
martino.ciaperoni@sns.it, ntziavel@ucsc.edu, piekarras@gmail.com

Abstract

Optimization problems such as *Viterbi decoding* and *V-optimal histogram* construction seek a path of *exact* length L through a state space that minimizes a cost function. These problems are traditionally solved using dynamic programming (DP). A best-first-search (BFS) solution is also applicable, yet requires maintaining a priority queue. In all cases, memory usage grows linearly with both state space size and path length. In this paper, we propose COMPACTBFS, a framework that limits the growth of the BFS priority queue to space-efficiently determine the exact optimal cost for a fixed-length path and then constructs such a path by a divide-and-conquer strategy that eliminates the memory overhead. We apply COMPACTBFS to Viterbi decoding, which remains relevant to speech recognition, and V-optimal histogram construction. Our experimental results demonstrate significant gains over state-of-the-art solutions in runtime and memory consumption.

1 Introduction

Several problems call for finding a sequence of given length L over a space of n states and m connections that optimizes a cost; they are conventionally solved by A* search or dynamic programming (DP). Examples include *Viterbi decoding*, i.e., finding a most likely sequence of hidden states—a *Viterbi path*—in a Hidden Markov Model (HMM) to explain an observed sequence of length L [Viterbi, 1967], used in speech recognition [Pratap *et al.*, 2024]; finding $L-1$ boundaries—a *V-optimal histogram* ('V' for variance)—over a numeric sequence that minimize the error of constant approximation within each partition [Jagadish *et al.*, 1998]; and identifying the most probable syntactic structure—a Viterbi parse tree—that generates a length-L input sequence of tokens under a probabilistic context-free grammar (PCFG) [Klein and Manning, 2003; Huang *et al.*, 2012].

Such algorithms iteratively derive solutions to a subproblem of length k+1 from solutions to subproblems of length k in $\mathcal{O}(mL)$ time. A*-based algorithms maintain a priority queue to prioritize subproblems that aid the solution in a *best-first* manner with an $\mathcal{O}(nL \log nL)$ complexity overhead, while DP prioritizes them in a *breadth-first* manner by length. Both *memoize* partial solutions; after determining the optimal cost, they backtrack over those memoized solutions to retrieve the path having that cost. In the *shortest-path* problem, path length is *arbitrary* [Russell and Norvig, 2010]; thus, a lower-cost arrival at node i in step ℓ dominates a higher-cost arrival in a different step ℓ'. In effect, it suffices to track, for each node, the minimum arrival cost and the incoming neighbor from which it is reached, in $\mathcal{O}(n)$ space. Still, when path length is *fixed* [Klein and Manning, 2003; Huang *et al.*, 2012] as L, arrivals at node i in different steps face continuations of different costs. We should thus memoize the lowest cost $\mathsf{Opt}(i, \ell)$ of arriving at node i in each step ℓ in $\mathcal{O}(nL + m)$ space, leading to memory constraints in GPU-based speech recognition [Pratap *et al.*, 2024]. If we discard partial solutions to work *in-place* in $\mathcal{O}(n + m)$ space, we need to recurse the algorithm $L-1$ times to build the optimal path after finding its cost in $\mathcal{O}(n^2L^2)$ time. A reformulation of the DP solution [Ciaperoni *et al.*, 2022; Ciaperoni *et al.*, 2024] constrains space to $\mathcal{O}(n + m)$ and time to $\mathcal{O}(n^2L \log L)$, yet does not directly apply when prioritizing subproblems by *best-first* search with a priority queue.

In this paper, we introduce COMPACTBFS, a framework for the efficient optimization of *fixed*-length paths that regulates the priority queue and avoids memoization, using $\mathcal{O}(n + m)$ space (m for the in-memory model). We vet COMPACTBFS on Viterbi decoding and V-optimal segmentation with real and synthetic data, gaining in memory use and runtime vs. prior work, especially under skewed cost distributions.

2 Background and Related Work

Dynamic programming (DP) [Bellman, 1966] explores the problem space exploiting an *optimal substructure* property, by which a globally optimal solution can be assembled from locally optimal solutions, to solve a problem by recursively *expanding* partial solutions in a breadth-first manner. **Best-first-search** algorithms [Pearl, 1984] may also exploit optimal substructures, as dynamic programming does [Sniedovich, 2006], to find a minimum-cost path from a start to an end node by repetitively *expanding*, i.e., visiting the neighbors of, the *most promising* unexpanded visited node. The A* algorithm [Russell and Norvig, 2010; Foead *et al.*, 2021], an instance of best-first-search, finds a path of *arbitrary* length (i.e., number of steps) that minimizes a cost function, prioritizing paths by a cost estimation heuristic

that is *admissible* (i.e., never overestimates a path's cost) and *consistent* (i.e., never estimates the cost via an intermediary node as lower) [Russell and Norvig, 2010]; the search may proceed from both start and end by bidirectional search [Pohl, 1969]. A* generalizes Dijkstra's algorithm [Dijkstra, 1959], by which the heuristic cost estimate of any unexplored path is 0. Dijkstra retains only the cheapest arrival at each node i and returns the cheapest path upon reaching a certain goal state, length L, or cost [Bemten *et al.*, 2019]; it is based on the assumption that any continuation from node i adds the same cost to all paths. However, when seeking a minimum-cost path of exactly L steps, arrivals at the same node i at different steps ℓ and ℓ' face different feasible continuations, whose cost depends on the exact steps remaining. In effect, a lower-cost arrival at i does not dominate a higher-cost arrival at a different step. An algorithm retaining only the cheapest arrival at each node i may thus miss the globally optimal length-L path; instead, we must track the lowest cost $\mathsf{Opt}(i, \ell)$ of arriving at node i at each step ℓ.

Hidden Markov Models (HMMs) explain observation sequences. An HMM comprises a set of K hidden states, each with probabilities to be an initial state, transition to other states, and emit an observation. *Decoding* seeks a sequence of states most likely to generate a sequence of observations:

Problem 1 (Decoding)**.** *Given an HMM and a sequence of T observations $Y = \{y_1, y_2, \ldots, y_T\}$, find the sequence of hidden states $Q = \{s_1^*, s_2^*, \ldots, s_T^*\}$ that maximizes the likelihood $P(Q, Y)$.*

The Viterbi algorithm [Viterbi, 1967; Viterbi, 2006] solves Problem 1 by DP; it finds application from telecommunications [Viterbi, 2006] to speech recognition [Gales and Young, 2007; Braun *et al.*, 2020; Pratap *et al.*, 2024], where it finds the most probable transcription for an acoustic signal, and, in forced alignment, aligning transcriptions to audio recordings, running on a composition of HMMs in which states represent words and their phonemes. However, it raises high memory and runtime requirements. Recently, [Jo *et al.*, 2019] proposed a heuristic for word recognition without proving its correctness. [Ciaperoni *et al.*, 2022; Ciaperoni *et al.*, 2024] enhanced decoding space efficiency at the cost of a runtime overhead via *divide-and-conquer*. A similar strategy has been applied to *arbitrary*-length sequence alignment [Hirschberg, 1975; Hirschberg, 1977; Korf, 1999; Korf and Zhang, 2000; Spouge, 1989; Spouge, 1991], but not to fixed-length path optimization, as in selecting and aligning L characters from each of two input strings. Other works reduce the state space representation for HMM classes [Siddiqi and Moore, 2005; Felzenszwalb *et al.*, 2003]. Improving on time complexity is a tough undertaking, due to lower bounds [Backurs and Tzamos, 2017]. [Klein and Manning, 2003] and [Huang *et al.*, 2012] apply A*-like policies to enhance Viterbi's time efficiency for PCFG parsing, yet neglect space efficiency. In contrast, we offer a time-efficient and parsimonious formulation.

Histogram construction splits a data series into a number of representative buckets that minimize an error function:

Problem 2 (Histogram Construction)**.** *Given $I = \{x_1, \ldots x_n\}$, $x_i \in \mathbb{R}$, and $B \in \mathbb{Z}^+$, find a segmentation (or histogram) H_B of I into B non-overlapping subsequences (or buckets) I_b with associated bucket representatives $\hat{x}_b$, $b \in \{1, \ldots, B\}$, that minimizes error function $E_I(H_B)$.*

We focus on *V-optimal* histograms [Jagadish *et al.*, 1998], i.e., $E_I(H_B) = \sum_{b=1}^{B} E_b$, where $E_b = \sum_{x_i \in I_b} (x_i - \hat{x}_b)^2$, and $\hat{x}_b$ is the mean of values in bucket I_b. This problem is solved by a DP algorithm [Jagadish *et al.*, 1998; Guha *et al.*, 2006; Halim *et al.*, 2009] quadratic in n. CompactBFS offers gains extensible to any monotonic and distributive error measure [Karras and Mamoulis, 2008].

Semirings and Dioids [Gondran and Minoux, 2008] A *semiring* is a 5-tuple $(D, \oplus, \otimes, \bar{0}, \bar{1})$, where D is a non-empty set, $\oplus$ is a binary, associative, and commutative operator, $\otimes$ is a binary and associative operator, $\bar{0}$ is a neutral element for $\oplus$ (i.e., $x \oplus \bar{0} = x$, for all $x \in D$), $\bar{1}$ is a neutral element for $\otimes$ (i.e., $x \otimes \bar{1} = \bar{1} \otimes x = x$, for all $x \in D$), the operator $\otimes$ distributes over $\oplus$ and $\bar{0}$ is absorbing for $\otimes$ (i.e., $x \otimes \bar{0} = \bar{0} \otimes x = \bar{0}$, for all $x \in D$). A *selective dioid* is a semiring in which $\oplus$ is also selective (i.e., $(x \oplus y = x) \vee (x \oplus y = y)$, for all $x, y \in D$). Selective dioids provide an abstract expressive framework for shortest-path and DP problems [Mohri, 2002; Huang, 2008; Tziavelis *et al.*, 2025].

3 The CompactBFS Framework

We define CompactBFS, which finds a fixed-length sequence that optimizes a cost measure, and apply it to Viterbi decoding (§ 3.1) and histogram construction (§ 3.2). Unlike dynamic programming, which visits subproblems in a fixed order even if some do not contribute to the solution, CompactBFS solves subproblems in a best-first fashion, as Figure 1 illustrates.

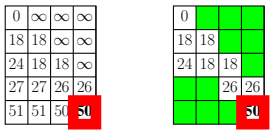

Figure 1: Cells stand for subproblems; DP solves all subproblems; CompactBFS does not need to solve subproblems in green.

CompactBFS starts out with the following components:

1. A *data space* $\mathcal{X}$ of n elements;
2. A *neighbor* function $\mathcal{N}: \mathcal{X} \to 2^{\mathcal{X}}$ that generates a *set* $\mathbb{X}$ of length-L *eligible* sequences (x_{i_ℓ}), $x_{i_\ell} \in \mathcal{X}$, $\ell \in \{1, \ldots, L\}$, with $x_{i_{\ell-1}} \in \mathcal{N}(x_{i_\ell})$ for $\ell \in \{2, \ldots, L\}$.
3. A *gap function* $G(j, i, \ell)$ associating a value with the transition from item $x_j \in \mathcal{X}$ to item $x_i \in \mathcal{X}$ at step ℓ.
4. A selective dioid $\mathcal{D} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ expressing the *value* of a sequence: $f(\mathbf{x}) = \bigotimes_{\ell=2}^{\ell=L} G(x_{i_{\ell-1}}, x_{i_\ell}, \ell)$, whereby extending a sequence never improves its value.
5. A *problem* that seeks an eligible *length-L* sequence $\mathbf{x}^* \in \mathbb{X}$ of *optimal* $f(\mathbf{x}^*)$, compared via the $\oplus$ operator.
6. A recursive *function* $\mathsf{Opt}(i, \ell)$ that stores the *optimal* value for an eligible length-ℓ sequence ending at $x_i \in \mathcal{X}$.
7. A *solution* to the problem in Item 5 by DP over sequences of increasing length from $\mathcal{X}$.

The solution in Item 7 finds an eligible sequence of L data items $\mathbf{x}^* = \{x_1^*, x_2^*, \ldots, x_L^*\}$ that optimizes $\mathsf{Opt}(\cdot, L)$; the

selective dioid properties, in particular distributivity, guarantee correctness. The DP computation takes the form:

$$\mathsf{Opt}(i,\ell) = \bigoplus_{j \in \mathcal{N}(i)} \{\mathsf{Opt}(j,\ell-1) \otimes G(j,i,\ell)\} \quad (1)$$

The recursion of Equation (1) requires $\Theta(n^2L)$ time and $\Theta(nL)$ space, iterating over items i and lengths ℓ and storing, for each (i,ℓ) pair, a predecessor needed to backtrack the optimal sequence. The entailed solution may be implemented by either DP or *token passing* [Young *et al.*, 1989]; both calculate *all* solutions of length ℓ, $\mathsf{Opt}(\cdot,\ell)$, before those of length $\ell+1$, $\mathsf{Opt}(\cdot,\ell+1)$, by *breadth-first search*. While DP draws from solutions of length ℓ to build each solution of length $\ell+1$, token passing broadcasts a *token* for each solution of length ℓ to its continuations of length $\ell+1$. In both cases, solutions solidify at length ℓ before moving to length $\ell+1$.

COMPACTBFS drops this *breadth-first* orientation for a *best-first* one; it organizes sub-problem solutions (represented by tokens) in a priority queue $\mathbf{Q}$ in which it initially inserts all tokens $(\cdot,1)$. Thereafter, in each step, it selects the most promising token (h,ℓ) from $\mathbf{Q}$ and updates the priority in $\mathbf{Q}$ of each eligible successor $(i,\ell+1)$ not already extracted from $\mathbf{Q}$ to $\mathsf{Opt}(h,\ell) \otimes G(h,i,\ell+1)$, when beneficial relative to the existing value. COMPACTBFS resembles Dijkstra's shortest-path algorithm [Dijkstra, 1959]; however, whereas Dijkstra minimizes a cost objective regardless of length (i.e., number of steps), COMPACTBFS optimizes the objective under a *fixed*-length constraint. It derives tokens (corresponding to DP table cells) selectively, without considering all options in order and without producing every token, resulting in significant savings. In problem-tailored variants, we prioritize tokens by anticipating the cost a sequence may obtain as it expands. We also derive bidirectional-search variants that craft both prefixes and suffixes of sequences until they reach the target length.

Nevertheless, the priority queue $\mathbf{Q}$ may reach size $\Theta(nL)$, holding one token for each state-length pair. Additionally, we need to memoize the subproblem solution each token represents even after we pop it from $\mathbf{Q}$, to enable *backtracking* the solution sequence. To ensure $\mathcal{O}(n)$ space complexity, which is highly desirable [Pratap *et al.*, 2024], we employ the following measures, which still calculate the same exact objective.

Controlling the priority queue size. First, to keep the size of $\mathbf{Q}$ in $\mathcal{O}(n)$, when the best-first-search queue $\mathbf{Q}$ exceeds a predefined size threshold θ (or memory budget) upon inserting a token (s_j,t), we invoke a *containment mechanism*, which is either a tailored *depth-first-search* (DFS) mechanism, or a mechanism that we propose, *earliest-first search* (EFS).

DFS. By the DFS mechanism, we pick the lowest-cost token (s_j,t^*) from the set of tokens for s_j, $\mathcal{S}^j = \{(s_j,\cdot)\}$, and generate all its derivative tokens via a DFS traversal of the HMM graph G starting at s_j. Each DFS branch terminates upon reaching the last frame (i.e., step) or upon injecting into the token set of a state $s_{j'}$ a token that either displaces or yields to a pre-existing one, without increasing memory usage. This DFS mechanism ensures queue size $\mathcal{O}(n)$ by constraining the size of $\mathbf{Q}$ within θ; it incurs $\mathcal{O}(n^2L^2)$ time in the worst case, as it may perform a DFS over all $O(nL)$ tokens for each token; in practice we do not encounter a large overhead.

EFS. Small θ values in the DFS mechanism may cause excessive DFS calls, reprocessing the same tokens multiple times and slowing runtime, even compared to that of regular dynamic programming which processes all nL tokens. To address this predicament, we propose an alternative search mechanism that processes each token at most once, *Earliest-First Search* (EFS). EFS processes tokens in increasing time frame order, starting form the earliest frame represented in the priority queue, until the gap between the earliest and latest time frames in the queue falls below a user-specified constant $\Delta_{\mathbf{Q}}$. In effect, the queue size is bounded by $\Delta_{\mathbf{Q}}n$ and best-first search resumes. While the EFS mechanism generates and must store additional tokens, at most $2n-1$ additional tokens will co-exist in the queue, occupying the earliest time frame and its successor. In effect, EFS ensures queue size $\mathcal{O}(n)$ and retains the compact $\mathcal{O}(nL(n+\log L))$ time including the queue overhead.

Reconstructing the optimal path. Second, instead of memoizing subproblem solutions for backtracking the final solution path, we construct that path by a *divide-and-conquer* strategy as in [Binder *et al.*, 1997; Ciaperoni *et al.*, 2022; Ciaperoni *et al.*, 2024]. Upon reaching the *middle* frame of a solution path, we record, with each subsequent token, the edge at that middle frame (or *middle pair*). After establishing the last-frame optimal cost, we recursively rerun the algorithm on the $L/2$-hop predecessors and successors of that token's middle pair to construct the entire path, retrieving middle pairs as in an in-order tree traversal [Ciaperoni *et al.*, 2022]. The main DP operation takes $\mathcal{O}(n^2L)$ time; maintaining the priority queue incurs an $\mathcal{O}(nL\log nL)$ overhead and the divide-and-conquer iteration brings an $\log L$ factor, hence the dominant terms are $\mathcal{O}(n^2L\log L + nL\log^2 L)$, with $\mathcal{O}(n)$ space.

3.1 The MINT algorithm

The Viterbi algorithm selects a sequence of T states $Q = \{s_1^*, s_2^*, \ldots, s_T^*\}$ from a universe of K HMM states $S = \{s_1, s_2, \ldots, s_K\}$ connected via a set of edges E, *most likely* to have generated a sequence of T observations $Y = \{y_1, y_2, \ldots, y_T\}$. Q is called *Viterbi path*. By the Markov property, the likelihood to be in a state depends only on the previous state. The algorithm thus uses the DP recursion:

$$\begin{aligned} \mathbf{T}[s_i,1] &= \pi_{s_i} \cdot B_{s_i,y_1}, \\ \mathbf{T}[s_i,t] &= \max_{s_h \in \mathcal{N}_{in}(s_i)} \{\mathbf{T}[s_h,t-1] \cdot A_{s_h,s_i}\} \cdot B_{s_i,y_t} \end{aligned} \quad (2)$$

where $\mathbf{T}[s_i,t]$ stores the probability of the most likely path ending at state s_i in t steps, or *time frames*, $\mathcal{N}_{in}(s_i)$ is the set of in-neighbors of s_i, π_i is the initial probability of s_i, A_{s_h,s_i} is the probability of transiting from state s_h to state s_i on a directed graph G capturing eligible transitions in the HMM, and B_{s_i,y_t} is the probability of observing y_t at state s_i. This setting suits COMPACTBFS as follows:

1. The *data space* $\mathcal{X}$ is the HMM of K hidden states $S = \{s_1, s_2, \ldots, s_K\}$.
2. The *neighbor* function $\mathcal{N}_{in} : \mathcal{X} \to 2^{\mathcal{X}}$ generates a *set* $\mathbb{X}$ of *eligible* sequences (x_{i_t}), $x_{i_t} \in \mathcal{X}$, $t \in \{1,\ldots,T\}$ as length-T paths of consecutive states in the HMM.
3. The *gap function* $G(j,i,t)$ is $A_{s_j,s_i}B_{s_i,y_t}$, or, in log-probabilities, as $\log A_{s_j,s_i} + \log B_{s_i,y_t}$.

4. The selective dioid is $([0,1], \max, \cdot, 0, 1)$, or, in the domain of log-probabilities, $([-\infty, 0], \max, +, -\infty, 0)$. Thus, the *value function* f assigns probabilities to paths, given the sequence of observations $Y = \{y_1, y_2, \ldots, y_T\}$; the probability that Y is generated by a sequence of hidden states $Q = \{s_1, s_2, \ldots, s_T\}$ is $P(Q, Y) = \pi_{s_1} \cdot B_{s_1 y_1} \prod_{t=2}^{T} A_{s_{t-1} s_t} \cdot B_{s_t y_t}$, where $\pi(s_1)$, $A_{s_{t-1} s_t}$, and $B_{s_t y_t}$ are defined as above.
5. The *problem* seeks an eligible sequence of states Q of length T that best explains the given sequence of observations Y, i.e., maximizes probability ($\oplus := \max$).
6. The recursive *function* $\mathsf{Opt}(i, \ell)$ that stores the *optimal* value for an eligible sequence of length ℓ that ends at data item $x_i \in \mathcal{X}$ is the function $\mathbf{T}[s_i, t = \ell]$.
7. The *solution* by DP over sequences of increasing length from $\mathcal{X}$ is given by Equation (2).

The recursion of Equation (2) requires $\mathcal{O}(K^2 T)$ time and $\mathcal{O}(KT)$ space, as it iterates over states s_i and time frames t. For the sake of efficiency and accuracy, we replace products of likelihoods by sums of log-likelihoods. In case the structure of G is known, we iterate only over states s_h that link to state s_i, hence visit each HMM graph edge only once; then time complexity becomes $\mathcal{O}((K + |E|)T)$.

The Viterbi algorithm and its *token passing* variant [Young *et al.*, 1989] operate by *breadth-first search*. MINT (Time Efficient Viterbi) replaces this strategy with *best-first* search. It organizes partial solutions in a priority queue $\mathbf{Q}$ and expands the most promising one in each step, as in [Klein and Manning, 2003]. Like Viterbi, it derives $\mathbf{T}[s_i, t+1]$ entries from $\mathbf{T}[s_h, t]$ entries. Still, while Viterbi calculates *all* $\mathbf{T}[\cdot, t]$ entries before $\mathbf{T}[\cdot, t+1]$ entries, MINT first inserts all tokens $(s_h, 1)$ in $\mathbf{Q}$ and then iteratively pops the most promising token (s_h, t) from $\mathbf{Q}$ and, for each *outgoing* neighbor $s_i \in \mathcal{N}_{out}(s_h)$ such that $(s_i, t+1)$ has not been extracted from $\mathbf{Q}$, it computes $\mathbf{T}[s_h, t] \cdot A_{s_h, s_i} \cdot B_{s_i, y_{t+1}}$ as a provisional $\mathbf{T}[s_i, t+1]$ value and updates the priority of $(s_i, t+1)$ accordingly. Thanks to the *monotonicity* of $\mathbf{T}[\cdot, \cdot]$ along a path, $\mathbf{T}[s_i, t]$ satisfies Equation (2) when (s_i, t) is popped from $\mathbf{Q}$. In the worst case, MINT visits each HMM edge once per time frame, hence takes $\mathcal{O}((K \log KT + |E|)T)$ time, where $\log KT$ expresses the priority queue overhead [Fredman and Tarjan, 1987]. In practice, it never considers some HMM edges. Next, we illustrate four MINT variants.

Standard MINT. In more detail, to achieve the $\max_{s_i} \mathbf{T}[s_i, T]$ objective, we *minimize* the positive path log-likelihood, $-\log P(Q, Y) \geq 0$, defined as *cost* of a path. For a given path Q ending at state i at frame t, we define its priority as $p(s_i, t) = -\log P(Q, Y)$, with $Q = \{s^1, s^2, \ldots, s^t\}$, such that $s^t = s_i$ and $Y = \{y_1, y_2, \ldots, y_t\}$. The pseudocode of the plain MINT variant is shown in the full version [Ciaperoni *et al.*, 2026]. First we insert in $\mathbf{Q}$ a token for each state in the first frame with priority $-\log \pi_{s_i} - \log B_{s_i, y_1}$. If a start state s is given, we only enqueue a token for s at $t = 0$. The main loop iterates until reaching the last frame. In each iteration, we dequeue from $\mathbf{Q}$ the top token (s_h, t), add it to a set $\mathbf{V}$ of visited tokens, compute the cost of reaching $(s_i, t+1)$ via (s_h, t) for each out-neighbor s_i of s_h that has no such token in $\mathbf{V}$, and insert or update $(s_i, t+1)$ in $\mathbf{Q}$ to the lowest known cost. When a pair (s, T) is dequeued, no path spanning T frames at lower cost exists, hence we return its cost as the Viterbi path log-likelihood $\max_{s_i} \mathbf{T}[s_i, T]$.

Proposition 1. *Standard MINT is correct.*

To return the optimal path, MINT by default appends a path to each token and, when updating the priority of $(s_i, t+1)$, also updates the corresponding path to s_i. An alternative implementation, MINT-Backtracking, stores only the *predecessor* of each token, retains all explored tokens, and at the end constructs the optimal path by *backtracking*. Next, we present three extensions to MINT.

MINT Bound. We propose a variant of MINT that orders tokens by lower-bounding the path cost from each token (s_h, t) until the final frame T using a *lower bound* $\hat{c}_1$ on the cost for moving from one frame to another for the remaining $T - t$ frames. We call the ensuing algorithm MINT Bound.

We first insert all states (or the source state, if given) in $\mathbf{Q}$ with $p(s, 1) = (T-1) \cdot \hat{c}_1$. As the search proceeds, we replace lower bounds with exact costs. The priority of (s_h, t) is $p(s_h, t) = -\log P(Q, Y) + (T-t) \cdot \hat{c}_1$, Q being the current optimal path ending at s_h in t frames; we compute the priority of a neighbour s_i for insertion or update in $\mathbf{Q}$ as $p(s_h, t) - \hat{c}_1 - \log A_{s_h, s_i} - \log B_{s_i, y_{t+1}}$. Upon reaching the last frame, all lower bounds capture exact costs. Setting $\hat{c}_1$ as the lowest value of $-\log A_{s_h, s_i} - \log B_{s_i, y_{t+1}}$ over all edges (s_h, s_i) at any t, found by pre-processing, ensures correctness.

Proposition 2. *MINT Bound is correct.*

MINT Bound encases information from unexamined time frames in the BestFS criterion and further explores already well-explored paths without compromising correctness.

Bidirectional MINT gains efficiency by bidirectional search [Pohl, 1969], exploring solutions from both the start and the end time frame. We denote the graph in which all edges are reversed as G_{rev}. In each direction, the search proceeds as in Standard MINT, yet with two priority queues, $\mathbf{Q}_f$ (forward search) and $\mathbf{Q}_b$ (backward search). If an initial state is given and a final state is not given, we find, in pre-processing, all states $\mathcal{S}(T)$ reachable from the initial state in T frames, and initiate $\mathbf{Q}_b$ with those. In each iteration, we expand both[1] searches, handling queues as in Standard MINT. Upon extracting (s_h^f, t^f) from $\mathbf{Q}_f$ and (s_h^b, t^b) from $\mathbf{Q}_b$, we update the associated visited-token sets, $\mathbf{V}_f$ and $\mathbf{V}_b$, and store associated costs in arrays d_f and d_b. We then consider tokens $(s_i^f, t^f + 1)$ for all neighbors s_i^f of s_h^f in G and tokens $(s_i^b, t^b - 1)$ for all neighbors s_i^b of s_h^b in the graph G_{rev}. We omit the details, which are the same as those in Standard MINT. When the two sides meet generating a path of length T, we update the hitherto best path cost μ, if the newly found path improves on it. Such a path is provably optimal, hence the algorithm terminates, when the sum of costs of tokens dequeued from $\mathbf{Q}_f$ and $\mathbf{Q}_b$ exceeds μ. To avoid double-counting emission probabilities, we add the emission probability of the last vertex visited while building a path only in the priority of $\mathbf{Q}_f$, thus the cost of any path through (w, t) is $d_f[(w, t)] + d_b[(w, t)]$.

Proposition 3. *Bidirectional MINT is correct.*

[1] A more refined strategy would choose which side to expand.

Bidirectional MINT Bound combines Bidirectional MINT and MINT Bound in one algorithm that searches in both directions and lower-bounds the total T-frames-long path costs used as priority values in both queues. The search in both directions follows the order determined by the cost of arriving to a state in a given number of steps (frames) plus a lower bound on the cost of arriving from there to the the end of the path. The single-frame lower bound for the forward search is as in MINT Bound. For backward search, it is the lowest cost of moving from a frame to the next one in the reverse HMM graph G_{rev}, i.e., the lowest value of $-\log A_{s_i,s_h} - \log B_{s_i,y_t}$ over all edges (s_i, s_h) in G_{rev}. We obtain both bounds by pre-processing with a single graph traversal. The correctness of Bidirectional MINT Bound follows from the correctness of MINT Bound and Bidirectional MINT.

Linear-space MINT (MINT-LS) Here, we propose a COMPACTBFS space-efficient variant, MINT-LS, which guarantees $\mathcal{O}(K)$ space by keeping the size of $\mathbf{Q}$ in $\mathcal{O}(K)$ by a *containment* strategy and reconstructs the solution by a *divide-and-conquer* strategy. Algorithm MINT-LS, shown in the full version [Ciaperoni *et al.*, 2026], avoids storing tokens when the queue size reaches a threshold θ. Instead, it invokes the DFS subroutine, which only stores tokens either as replacements of others or at the last frame; with each such token (s_h, t), it stores its predecessor state and the running middle pair for its path. These details help identify the subproblems to be solved recursively. Upon reaching the final frame (and the final state, if specified), it extracts the middle pair associated with the solution path and reruns recursively among N_p-hop predecessors of the middle pair in as many preceding frames and among N_s-hop successors in as many following frames, identified through breadth-first search. MINT-LS+ applies our EFS strategy that visits each token at most once prioritizing tokens by time frame with an auxiliary queue $\mathbf{Q}_t$; it keeps track of the earliest and latest time frame in $\mathbf{Q}_t$ and toggles between BestFS and EFS as required. As results in Section 4.1 show, these strategies ensure low space consumption. With large θ, DFS is preferable; otherwise, EFS is preferred. MINT-LS naturally combines with bidirectional search.

3.2 The TECH algorithm

We apply COMPACTBFS to V-Optimal histogram construction [Jagadish *et al.*, 1998], outlined in Section 2. A V-Optimal histogram uses the mean value $\hat{x} = \frac{1}{i-j+1}\sum_{k=j}^{i} x_k$ as a representative to minimize the *Euclidean* error in a bucket I_b extending from the j^{th} to the i^{th} value in the sequence, $E(j,i) = \sum_{k=j}^{i}(x_k - \hat{x})^2$. The total error is aggregated over all buckets. V-Optimal algorithms use incremental sums S and sums of squares SS to obtain any $E(j,i)$ as:

$$E(j,i) = (SS[i] - SS[j-1]) - \frac{(S[i] - S[j-1])^2}{(i-j+1)}. \quad (3)$$

The algorithm finds, for each combination of a value index i and number of buckets b, the cost of the optimal b-bucket histogram covering the first i values in the sequence, as:

$$E^*(i,b) = \min_{1 \le j < i} E^*(j, b-1) + E(j+1, i), \quad (4)$$

$E^*(i,b) = 0$ for $i \le b$ and $E^*(i,1) = E(1,i)$. After finding the optimal cost $E^*(n,B)$ for a length-n sequence and B buckets, we backtrack the histogram. The problem maps to COMPACTBFS as follows:

1. The *data space* $\mathcal{X}$ is the data series $I = \{x_1, \dots x_n\}$.
2. The *neighbor* function $\mathcal{N}(x_i) = \{x_j | 1 \le j < i\}$ generates a *set* $\mathbb{X}$ of *eligible sequences* (x_{i_t}), $x_{i_t} \in \mathcal{X}$, $t \in \{1, \dots, B+1\}$ as *ordered* boundaries from x_1 to x_n.
3. The *gap function* $G(j,i,b)$ is the bucket error $E(j+1,i)$ irrespective of b.
4. The selective dioid is $(\mathbb{R} \cup \{+\infty\}, \min, +, +\infty, 0)$. Thus, the *value function* $f : \mathbb{X} \to \mathbb{R}$ assigns an approximation error $\sum_{b=1}^{B} E_b$ to a histogram.
5. The *problem* seeks an error-minimizing eligible histogram of $B+1$ boundaries ($\oplus := \min$).
6. The recursive *function* $\mathsf{Opt}(i, \ell)$ that stores the *optimal* value for an eligible sequence of length ℓ ending at item $x_i \in \mathcal{X}$ is the function $E^*(i, b = \ell)$.
7. The *solution* by DP over sequences of increasing length from $\mathcal{X}$ is given by Equation (4).

V-OPT histogram construction by DP [Jagadish *et al.*, 1998] requires $O(n^2 B)$ time and $O(nB)$ space. We discuss COMPACTBFS variants, TECH (Time-Efficient Histogram), aligned to the variants of MINT.

Standard TECH. Standard TECH, like MINT, employs a priority queue $\mathbf{Q}$ to prune computations, where the priority of entry (h,b) is the cost of the b-bucket histogram for the first h values, $p(h,b) = E^*(h,b)$. After computing the S and SS arrays, used to compute the error measure by Equation (3), in each iteration, TECH dequeues the pair (h,b) of lowest error, adds it to a set $\mathbf{V}$ of visited tokens, and, provided $b < B$, computes via (h,b) the error for each pair $(i, b+1)$ with $i > h$ that is not in $\mathbf{V}$ and inserts or updates $(i, b+1)$ in $\mathbf{Q}$ accordingly; thereby, it explores possible next buckets. We do not iterate over $(i, b+1)$ pairs for all $i > h$, but stop at $i = n - B + b + 1$ since there must be at least $B - b - 1$ values after i to make B buckets in total. The algorithm terminates after it dequeues (n, B). Correctness follows as in Proposition 1: after (n, B) is dequeued, there can be no lower-cost histogram of the same series and B. For further pruning, we use an upper bound $UB[h,b]$ on the cost of a $(B-b)$-bucket histogram for the series $\{h+1, \dots n\}$, derived in Proposition 4 below. If, after visiting (h,b), we find $i^* > h$ such that $E(h+1, i^*) \ge UB[h,b]$, we eschew computing $E(h+1,i)$ for $i > i^*$, as we have already exceeded the upper bound on the error therefrom.

TECH Bound. This variant uses bounds on the cost of a B-bucket histogram instead of the cost of a partial histogram with $b \le B$ buckets. Given (h,b), we partition the series $\{h+1, \dots n\}$ in $B-b$ equal-width buckets. The minimum error among such buckets is a lower bound to the V-optimal histogram cost, while the sum of those errors is an upper bound, which we may use for pruning.

Proposition 4. *The minimum error* $\min_b E_b$ *among* b *buckets of an equal-width partitioning a sequence* I *is a lower bound to the V-optimal histogram cost and* $\sum_b E_b$ *is an upper bound.*

TECH Bound adds the lower bound $LB[h,b]$ to the priority of each pair (h,b). Upon arrival at the end of the series, it

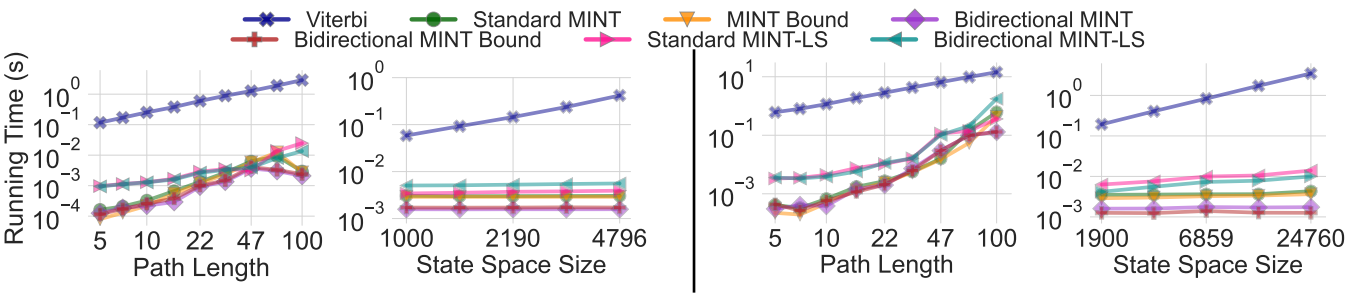

Figure 2: Runtime vs. T and K; forced alignment (L), standard decoding (R); axes on log scale; real data.

outputs the same cost as Standard TECH. We find these bounds by building equal-width histograms in a pre-processing step. Correctness follows as in the case of MINT Bound.

Other TECH variants work by analogy to MINT variants. **Bidirectional TECH** applies forward and backward search. When we pop a pair (h, b) from $\mathbf{Q}_f$, we consider entries $(i, b+1)$ with $i > h$, and, if the backward search has already visited $(i, B-b-1)$, we check whether the cost $d_f[(h,b)] + E(h,i) + d_b[(i, B-b-1)]$ improves upon the current best cost μ, and likewise in backward search. The search terminates when the sum of the costs of pairs from both queues exceeds the current best cost μ. **Bidirectional TECH Bound** combines Bidirectional TECH with TECH Bound, with reversed lower bounds to consider sequences of the form $\{1, \dots h\}$ rather than $\{h+1, \dots n\}$. **TECH-LS** variants limit space needs; in place of a *middle pair* of states, they detect a *middle bucket* that splits the data series in halves. [Guha, 2005] applied such a space-saving solution on DP.

4 Experiments

We experimented[2] on a 128GB, 2×12 core Xeon E5 2680 v3 machine @2.5GHz. For Viterbi decoding, we use as baselines the edge-aware Viterbi algorithm (§3.1), its recomputation-based space-efficient variant [Ciaperoni *et al.*, 2022], checkpoint Viterbi [Tarnas and Hughey, 1998], which segments a sequence into $\sqrt{T}$ parts, and SIEVE variants [Ciaperoni *et al.*, 2022], including SIEVE-Middlepath, Standard SIEVE (partitioning the state space), and SIEVE-Hyperloglog (using approximate counts) [Flajolet *et al.*, 2007]. For histogram construction, the baseline is the DP algorithm of [Jagadish *et al.*, 1998] (§ 3.2). The full version [Ciaperoni *et al.*, 2026] details data, measures, parameters, and further experiments.

4.1 Results on Viterbi decoding

Real data, runtime vs. T and K. Figure 2 plots results on forced-alignment and standard decoding with real data. MINT achieves savings of up to three orders of magnitude over Viterbi by focusing on promising paths. Enlarging the state space via snowball samples of the HMM does not affect MINT, but heavily impacts Viterbi. MINT-LS variants also gain lower runtime than Viterbi even while controlling memory.

Real data, runtime and memory vs. T. Figure 3 plots runtime and memory usage, including SIEVE variants. As the memory footprint of MINT-LS and SIEVE variants is *dynamic*, we report its median across recursion levels. The memory needs of MINT-LS (using DFS) are the lowest, while those of MINT-LS+ (using EFS) remain below those of the most memory-intensive SIEVE variants. In the full version we report the corresponding maximum memory usage. Overall, MINT-LS variants require memory comparable to SIEVE variants. We show results vs. T, as subsampling the state space on real data barely affects MINT-LS, as established in Figure 2.

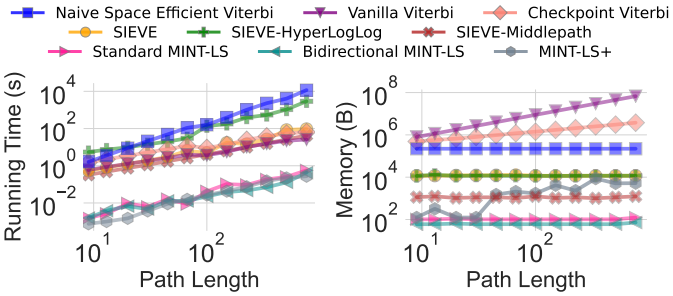

Figure 3: Runtime & (median over recursive calls) memory use vs T; axes on log scale; real forced alignment data.

Effect of θ on linear-space algorithms. Figure 4 presents runtimes vs. the queue size threshold θ on ranges of θ where MINT-LS may process tokens more than once (Section 3.1). MINT-LS+ never processes the same token twice. SIEVE [Ciaperoni *et al.*, 2022], a DP-based algorithm, also ensures $\mathcal{O}(K)$ space, yet incurs a practical runtime overhead *(not time-complexity overhead)* compared to Viterbi; we compare runtime to the latter as a baseline. MINT-LS consistently performs the fastest on synthetic data with skewed path likelihood distribution, where it explores only a few paths, yet becomes inefficient for small θ with uniform path likelihood distribution, where it explores more paths. On real data with forced alignment, the runtime of MINT-LS grows modestly as θ falls; with larger decoding data, runtime grows beyond that of Viterbi. Overall, MINT-LS is preferable for large enough θ, while MINT-LS+ maintains runtime within the same order of magnitude as, and typically lower than, that of Viterbi.

Effect of T on linear-space algorithms. Figure 5 plots runtimes vs. path length T for fixed values of the queue size threshold θ small enough to challenge MINT-LS, reconfirming that it becomes inefficient as more paths are explored.

[2]Code and data at https://github.com/maciap/CompactBestFirst.

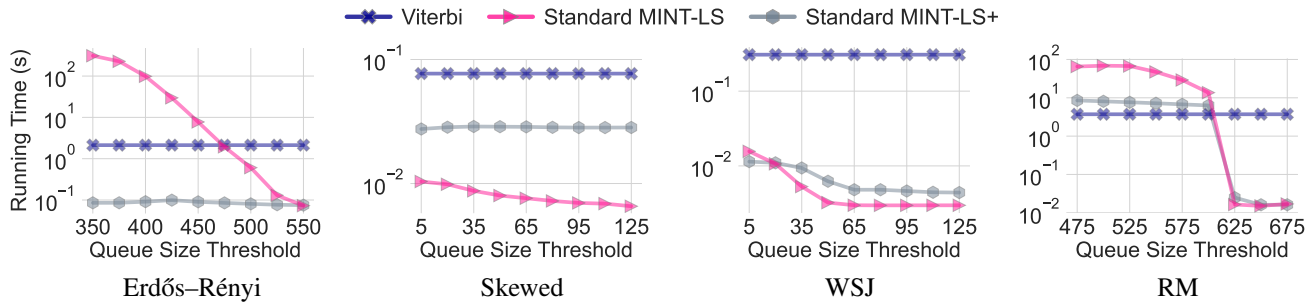

Figure 4: Runtime vs. queue size threshold θ; synthetic data with uniform (Erdős–Rényi) and skewed path likelihood distribution (Skewed), and real data on forced alignment (WSJ) and standard decoding (RM); y-axis in log scale.

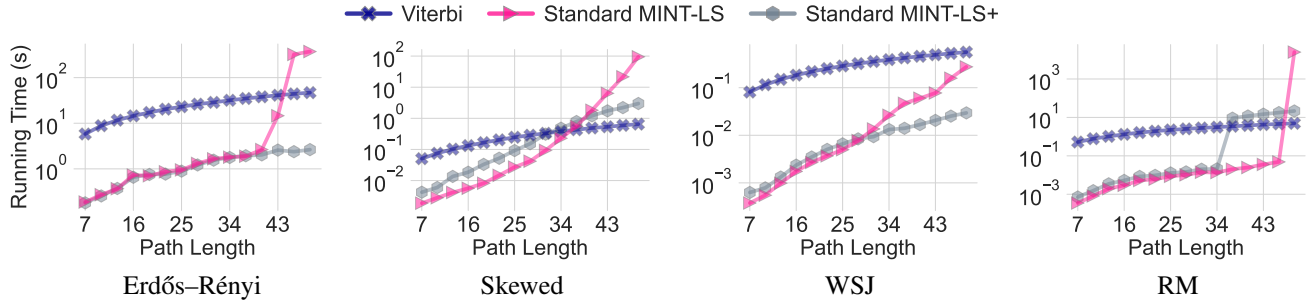

Figure 5: Runtime vs. path length T; synthetic data with uniform (Erdős–Rényi) and skewed path likelihood distribution (Skewed), real data on forced alignment (WSJ) and standard decoding (RM); y-axis in log scale.

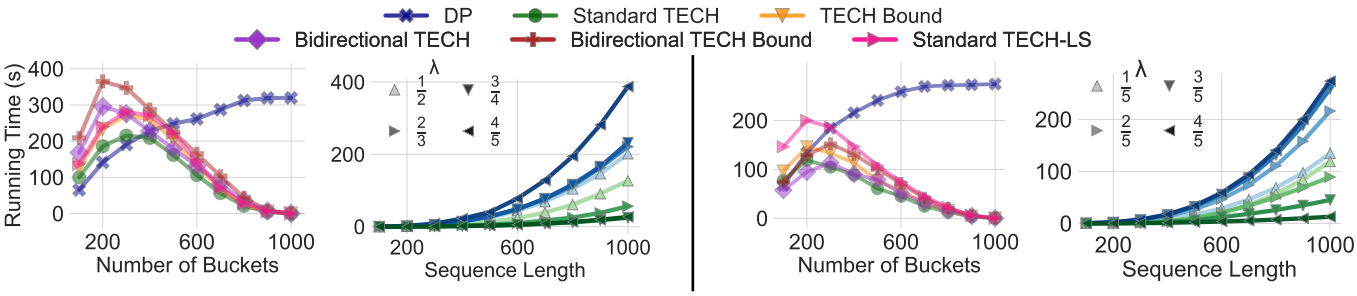

Figure 6: Histogram construction: runtime vs. B, synthetic (L) and real (R) data; n varying $\lambda = \frac{B}{n}$.

As in Figure 4, this effect is evident in synthetic data with Erdős–Rényi transitions and uniform path likelihood distribution and in the real decoding data. MINT-LS+ emerges as the algorithm of choice, achieving competitive runtime even while maintaining memory usage under a small θ threshold.

4.2 Results on histogram construction

Figure 6 shows the histogram construction runtime on synthetic data vs. B for input sequence length $n = 1010$, and that of Standard TECH and the standard V-OPT algorithm vs. n for different values of $\lambda = \frac{B}{n}$. Standard TECH often suffices, as its variants do not bring significant gains. Linear-space TECH incurs a manageable runtime overhead for the sake of space efficiency. All CompactBFS variants gain in time efficiency as B grows, delimiting the search space for each bucket. Contrariwise, standard DP cannot contain the search space, hence its runtime surges for large B. Figure 6 also shows results on real data. The gains of TECH solutions are *more emphatic* here, as TECH exploits data patterns that facilitate summarization, whereas standard DP lacks such capacity.

5 Conclusion

We introduced CompactBFS, a framework that efficiently solves fixed-length path optimization problems by best-first search, while delimiting space complexity. We crafted CompactBFS-based algorithms for Viterbi decoding and histogram construction. Our experiments evince CompactBFS to gain up to four orders of magnitude in time and space efficiency, most pronouncedly on real data with nonuniform cost distributions. In the future, we aim to apply space-saving strategies to other applications of best-first search, such as nearest-neighbor search [Papadopoulos *et al.*, 2011]. In the full version [Ciaperoni *et al.*, 2026], we apply CompactBFS to temporal-graph community search.

References

[Backurs and Tzamos, 2017] Arturs Backurs and Christos Tzamos. Improving Viterbi is hard: Better runtimes imply faster clique algorithms. In *ICML*, pages 311–321, 2017.

[Bellman, 1966] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[Bemten *et al.*, 2019] Amaury Van Bemten, Jochen W. Guck, Carmen Mas Machuca, and Wolfgang Kellerer. Bounded Dijkstra (BD): Search space reduction for expediting shortest path subroutines. *CoRR*, abs/1903.00436, 2019.

[Binder *et al.*, 1997] John Binder, Kevin P. Murphy, and Stuart Russell. Space-efficient inference in dynamic probabilistic networks. In *IJCAI*, pages 1292–1296, 1997.

[Braun *et al.*, 2020] Hugo Braun, Justin Luitjens, Ryan Leary, Tim Kaldewey, and Daniel Povey. Gpu-accelerated viterbi exact lattice decoder for batched online and offline speech recognition. In *ICASSP*, pages 7874–7878, 2020.

[Ciaperoni *et al.*, 2022] Martino Ciaperoni, Aristides Gionis, Athanasios Katsamanis, and Panagiotis Karras. SIEVE: A space-efficient algorithm for Viterbi decoding. In *ACM SIGMOD*, pages 1136–1145, 2022.

[Ciaperoni *et al.*, 2024] Martino Ciaperoni, Athanasios Katsamanis, Aristides Gionis, and Panagiotis Karras. Beam-search SIEVE for low-memory speech recognition. In *Interspeech 2024*, pages 272–276, 2024.

[Ciaperoni *et al.*, 2026] Martino Ciaperoni, Nikolaos Tziavelis, and Panagiotis Karras. Efficient optimization of fixed-length paths (full version). https://github.com/maciap/CompactBestFirst/blob/main/full_version.pdf, 2026.

[Dijkstra, 1959] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Felzenszwalb *et al.*, 2003] Pedro F. Felzenszwalb, Daniel P. Huttenlocher, and Jon M. Kleinberg. Fast algorithms for large-state-space HMMs with applications to web usage analysis. In *NeurIPS*, pages 409–416, 2003.

[Flajolet *et al.*, 2007] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Conference on Analysis of Algorithms (AofA)*, pages 137–156, 2007.

[Foead *et al.*, 2021] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. A systematic literature review of A* pathfinding. *Procedia Computer Science*, 179:507–514, 2021.

[Fredman and Tarjan, 1987] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[Gales and Young, 2007] Mark J. F. Gales and Steve J. Young. The application of hidden markov models in speech recognition. *Foundations and Trends in Signal Processing*, 1(3):195–304, 2007.

[Gondran and Minoux, 2008] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*. Springer, 2008.

[Guha *et al.*, 2006] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Trans. Database Syst.*, 31(1):396–438, 2006.

[Guha, 2005] Sudipto Guha. Space efficiency in synopsis construction algorithms. In *VLDB*, pages 409–420, 2005.

[Halim *et al.*, 2009] Felix Halim, Panagiotis Karras, and Roland HC Yap. Fast and effective histogram construction. In *CIKM*, pages 1167–1176, 2009.

[Hirschberg, 1975] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[Hirschberg, 1977] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

[Huang *et al.*, 2012] Zhiheng Huang, Yi Chang, Bo Long, Jean-François Crespo, Anlei Dong, S. Sathiya Keerthi, and Su-Lin Wu. Iterative Viterbi A* algorithm for k-best sequential decoding. In *ACL*, pages 611–619, 2012.

[Huang, 2008] Liang Huang. Advanced dynamic programming in semiring and hypergraph frameworks. In *Coling 2008: Advanced Dynamic Programming in Computational Linguistics: Theory, Algorithms and Applications - Tutorial notes*, pages 1–18, 2008.

[Jagadish *et al.*, 1998] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugavelayutham Muthukrishnan, Viswanath Poosala, Kenneth Clem Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

[Jo *et al.*, 2019] Jihyuk Jo, Han-Gyu Kim, In-Cheol Park, Bang Chul Jung, and Hoyoung Yoo. Modified viterbi scoring for hmm-based speech recognition. *Intelligent Automation & Soft Computing*, 25(2):351–358, 2019.

[Karras and Mamoulis, 2008] Panagiotis Karras and Nikos Mamoulis. Hierarchical synopses with optimal error guarantees. *ACM Transacation on Database Systems*, 33(3):18:1–18:53, 2008.

[Klein and Manning, 2003] Dan Klein and Christopher D. Manning. A* parsing: Fast exact Viterbi parse selection. In *HLT-NAACL*, pages 119–126, 2003.

[Korf and Zhang, 2000] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *AAAI*, pages 910–916, 2000.

[Korf, 1999] Richard E. Korf. A divide and conquer bidirectional search: First results. In *IJCAI*, pages 1184–1191, 1999.

[Mohri, 2002] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *J. Autom. Lang. Comb.*, 7(3):321–350, 2002.

[Papadopoulos *et al.*, 2011] Stavros Papadopoulos, Lixing Wang, Yin Yang, Dimitris Papadias, and Panagiotis Karras. Authenticated multistep nearest neighbor search. *IEEE Trans. Knowl. Data Eng.*, 23(5):641–654, 2011.

[Pearl, 1984] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[Pohl, 1969] Ira Pohl. Bi-directional and heuristic search in path problems. Technical report, Stanford Linear Accelerator Center, Calif., 1969.

[Pratap *et al.*, 2024] Vineel Pratap, Andros Tjandra, Bowen Shi, Paden Tomasello, Arun Babu, Sayani Kundu, Ali Elkahky, Zhaoheng Ni, Apoorv Vyas, Maryam Fazel-Zarandi, Alexei Baevski, Yossi Adi, Xiaohui Zhang, Wei-Ning Hsu, Alexis Conneau, and Michael Auli. Scaling speech technology to 1, 000+ languages. *J. Mach. Learn. Res.*, 25:97:1–97:52, 2024.

[Russell and Norvig, 2010] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[Siddiqi and Moore, 2005] Sajid M. Siddiqi and Andrew W. Moore. Fast inference and learning in large-state-space HMMs. In *Proceedings of the 22nd International Conference on Machine learning (ICML)*, pages 800–807, 2005.

[Sniedovich, 2006] Moshe Sniedovich. Dijkstra's algorithm revisited: the dynamic programming connexion. *Control and cybernetics*, 35(3):599–620, 2006.

[Spouge, 1989] John L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal on Applied Mathematics*, 49(5):1552–1566, 1989.

[Spouge, 1991] John L. Spouge. Fast optimal alignment. *Comput. Appl. Biosci.*, 7(1):1–7, 1991.

[Tarnas and Hughey, 1998] Christopher Tarnas and Richard Hughey. Reduced space hidden Markov model training. *Bioinformatics*, 14(5):401–406, 1998.

[Tziavelis *et al.*, 2025] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries. *ACM Trans. Database Syst.*, 51(1), September 2025.

[Viterbi, 1967] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[Viterbi, 2006] Andrew J. Viterbi. A personal history of the viterbi algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, 2006.

[Young *et al.*, 1989] Steve J. Young, N.H. Russell, and J.H.S. Thornton. *Token passing: a simple conceptual model for connected speech recognition systems*. University of Cambridge, Department of Engineering, 1989.