Ciaperoni, Martino; Gionis, Aristides; Katsamanis, Athanasios; Karras, Panagiotis

**SIEVE: A Space-Efficient Algorithm for Viterbi Decoding**

*Published in:*
SIGMOD 2022 - Proceedings of the 2022 International Conference on Management of Data

*DOI:*
10.1145/3514221.3526170

Published: 10/06/2022

*Document Version*
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

*Please cite the original version:*

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.
SIEVE: A Space-Efficient Algorithm for Viterbi Decoding

Martino Claperoni
Aalto University

Athanasios Katsamanis
Athena R.C., Behavioral Signals

ABSTRACT

Can we get speech recognition tools to work on limited-memory devices? The Viterbi algorithm is a classic dynamic programming (DP) solution used to find the most likely sequence of hidden states in a Hidden Markov Model (HMM). While the algorithm finds universal application ranging from communication systems to speech recognition to bioinformatics, its scalability has been scarcely addressed, stranding it to a space complexity that grows with the number of observations.

In this paper, we propose SIEVE (Space Efficient Viterbi), a reformulation of the Viterbi algorithm that eliminates its space-complexity dependence on the number of observations to be explained. SIEVE discards and recomputes parts of the DP solution for the sake of space efficiency, in divide-and-conquer fashion, without incurring a time-complexity overhead. Our thorough experimental evaluation shows that SIEVE is highly effective in reducing the memory usage compared to the classic Viterbi algorithm, while avoiding the runtime overhead of a naïve space-efficient solution.

CCS CONCEPTS

• Theory of computation → Algorithm design techniques; • Computing methodologies → Model development and analysis.

KEYWORDS

Viterbi decoding, space efficiency, divide-and-conquer

1 INTRODUCTION

The Viterbi algorithm [23] finds the sequence of hidden states — the Viterbi path — that best explains an observed event sequence in the context of Markov information sources and Hidden Markov models (HMMs), where the states that give rise to the event sequence are not directly observable. It finds application in several fields, particularly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9249-5/22/06 $15.00
https://doi.org/10.1145/3514221.3526170
An HMM is fully and uniquely described by the triplet \((\pi, A, B)\); the state space of the hidden variables is discrete, while observations can either be discrete, generated from a categorical distribution, or continuous, e.g., described by a Gaussian Mixture Model (GMM) or a Deep Neural Network (DNN) [6]. Figure 2 shows a simple example. The matrix of transition probabilities \(A\) is also represented by a directed graph \(G = (V, E)\), the HMM graph, where each vertex in \(V\) stands for a state in \(S\) and a directed edge \((a, b)\) in \(E\) indicates a non-zero transition probability from state \(a\) to state \(b\).

### 2.1 HMM-based inference

Given an HMM and a sequence of observations \(Y = \{y_1, y_2, \ldots, y_T\}\), the probability that \(Y\) is generated by a sequence of hidden states \(Q = \{q_1, q_2, \ldots, q_T\}\) is:

\[
P(Q, Y) = \pi_{q_1} \cdot B_{q_1y_1} \prod_{i=2}^{T} A_{q_{i-1}q_i} \cdot B_{q_iy_i},
\]

where \(\pi(q_1)\) is the probability of observing the initial state \(q_1\), \(A_{q_{i-1}q_i}\) is the probability of transitioning from state \(q_{i-1}\) to state \(q_i\), and \(B_{q_iy_i}\) is the probability of observing \(y_i\) at state \(q_i\). The problem of determining the sequence of states that best explains a sequence of observations is formally defined as follows.

**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 \(Q = \{q_1, q_2, \ldots, q_T\}\) that maximizes the probability \(P(Q, Y)\).

This formulation also applies to automatic speech recognition, where states correspond to phoneme parts. There are typically three states per phoneme (the essentially smaller-scale HMM) and the phonemes are connected to each other with certain transition probabilities (based on how probable a specific word pronunciation is or how probable a sequence of words may be) forming a large HMM. The goal is to find the optimal phoneme sequence for the sequence of acoustic observations. These observations are estimated from the input speech recording every 10-30 ms and comprise spectral energy measurements (or variants) in a selection of frequency bands; Benzeghiba et al. [2] provide a comprehensive review.

In this paper, we demonstrate how our proposed methodology improves the efficiency of HMM-based inference in a simplified speech-recognition problem, better known as *speech-text forced alignment*, which aims at providing the optimal state sequence of the HMM when the orthographic transcription of the speech recording is known. The exact output of the forced aligner is the sequence of time-aligned, i.e., with start and end times, phonemes or words. This problem arises in multimedia-indexing applications [16] or when training large-vocabulary speech recognition or speech-synthesis systems using long audio recordings [5].

### 2.2 The Viterbi Algorithm

The Viterbi algorithm [23] solves the decoding and forced alignment problems in HMMs, as it finds the sequence of \(T\) hidden states \(Q = \{q_1, q_2, \ldots, q_T\}\), selected from a universe of \(K\) states \(S = \{s_1, s_2, \ldots, s_K\}\), which is most likely to have generated a sequence of \(T\) observations \(Y = \{y_1, y_2, \ldots, y_T\}\), coming from a vocabulary of \(N\) possible observations \(O = \{o_1, o_2, \ldots, o_N\}\). This sequence is called Viterbi path. The algorithm uses a dynamic-programming (DP) recursion:

\[
T[i, 1] = \pi_i \cdot B_{iy_1},
\]

\[
T[i, j] = \max_{s_k \in N_{in}(s_i)} \{T[k, j-1] \cdot A_{ki} \cdot B_{iy_j}\} \tag{2}
\]

Here, \(T[i, j]\) stores the probability of the most likely path of \(j\) states ending at state \(s_i\); \(s_k\) is a state that precedes \(s_i\), i.e., any of the in-neighbours \(N_{in}(s_i)\) of \(s_i\) in the HMM graph; \(\pi_i, A_{ki}\) and \(B_{iy_j}\) follow the notation of Equation (1). The probability of being in a state depends only on the previous state, defining a discrete Markov chain. The recursion in Equation (2) finds \(T[i, j]\) and the corresponding Viterbi path using the precomputed probabilities of the most likely paths of \(j-1\) states ending at any state \(s_k\) that links to \(s_i\).

To tabulate all \(T[i, j]\) values, the DP recursion needs \(O(K^2T)\) time and \(O(KT)\) space, where \(T\) is the path length and \(K\) the number of states; these complexities are agnostic of the HMM graph \(G\). In the *edge-aware* case where the structure of \(G\) is known, we iterate only over states \(s_k\) that link to each state \(s_i\), hence visit each edge in the HMM only once and time complexity becomes \(O(|E|T)\).

### 2.3 Extensions

Since its introduction in 1967, the Viterbi algorithm has found applications in diverse fields and has prompted several extensions. Hagenauer and Hoehler [14] extended the algorithm by enriching the retrieved path with reliability information. The *iterative Viterbi algorithm* (IVA) [25] finds the subsequence of observations
within a given sequence \( Y \) that is most likely to have come from the given HMM. Feldman et al. [7] introduced the lazy Viterbi algorithm, which modifies the classic Viterbi algorithm making use of additional data structures (a trellis and a priority queue) to increase time efficiency at the expense of memory usage. Other efficient implementations of the Viterbi algorithm have been proposed for particular classes of HMMs [8, 20]. The Token Passing [28] algorithm provides an alternative formulation of the Viterbi algorithm that fits the continuous speech recognition scenario. At each time \( t \), each state holds a token and passes it to its connected states, increasing the path log-probability accordingly. Afterwards, all tokens in a given state except the one with largest probability are discarded. The tokens carry certain pieces of information: the log-probability and pointers to reconstruct the route that a token has followed, which may refer to words (WordLink Records) or individual phonemes. In the end, the token corresponding to the path of largest probability contains a linked list that allows tracing that path. To tackle the problem of tracking multiple objects in image sequences, Ardó et al. [1] proposed a variant of the Viterbi algorithm that finds suboptimal paths in real time. Nevertheless, the size of the state space \( K \) is relatively small and in practice, the exact value of \( K \) is determined by data availability and human expertise. The standard approach resorts to criteria such as the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC) [18], which express the accuracy and conciseness by which an HMM fits the observed data. On the other hand, the length of the observation sequence \( T \) is determined by the observed data and is often large, rendering the algorithm’s memory requirements prohibitive for practical data science problem instances. Still, previous work has paid scarce attention to space complexity in the Viterbi algorithm. An attempt to reduce its space complexity [21] stores the values in a subset of \( \sqrt{T} \) rows (checkpoints) of the DP table and locally recomputes the Viterbi path between checkpoints, yet preserves a \( O(K\sqrt{T}) \) space complexity dependence on \( T \).

### Table 1: Notations.

| \( T \) | Length of observation sequence; in a subproblem |
| \( K, K' \) | State space size; in a subproblem |
| \( N \) | Vocabulary size |
| \( Y \) | Sequence of observations |
| \( S \) | State space |
| \( G \) | HMM graph |
| \( \pi \) | Initial state probabilities |
| \( A \) | State transition probability matrix |
| \( O \) | Vocabulary |
| \( E \) | Emission probability matrix |
| \( s_m \) | Median pair |
| \( N_\pi, N_e \) | Predecessors; successors of state \( s_i \) |
| \( M \) | Array of median pairs |
| \( M_\pi, M_e \) | Array of median pair scores |
| \( N_\pi \) | Number of observations before; after the median pair |
| \( N_\pi \) | Array of number of observations before the median pair |

### 3 THE SIEVE ALGORITHM

Here, we present our space-efficient alternative to the Viterbi algorithm, SIEVE. Table 1 collects the notations we employ. SIEVE discards some parts of a DP solution and re-computes them later, for the sake of space efficiency, in a divide-and-conquer fashion that keeps time complexity in check. This algorithm design paradigm has been used in the context of data summarization [13, 15], where it recursively divides a problem into two subproblems, straightforwardly identified via the middle element of a data sequence, which is known in advance. In the context of HMMs, the application of the paradigm is not straightforward, as it calls for identifying the equivalent of such a middle element, as we explain in the following.

As a starting point, we can reduce Viterbi’s space complexity to \( O(K) \) at the cost of a time complexity increase, as follows: instead of maintaining all \( T[i, j] \) values throughout the run of the algorithm, we keep only two rows at any time, \( T[\ast, j] \) and \( T[i, \ast] \) for \( i < j \). Upon completion, we know the Viterbi path probability, as well as the second-to-last path element, \( s^*_T-1 \). Yet, miss path elements from \( s^*_1 \) to \( s^*_T-2 \). To get the last missing path element, a \( s^*_T-2 \) we rerun the algorithm up to the last known node, \( s^*_T-1 \). In general, we rerun the algorithm on a problem of size \( T - i \) to get the path element \( T - i \). By this approach, space complexity falls to \( O(K) \), yet time complexity rises to \( O\left(\sum_{i=1}^{T} K^2\right) = O(K^2T^2) \) in the default case and \( O\left(\sum_{i=1}^{T} |E|\right) = O(|E|T^2) \) in the edge-aware case. We refer to this algorithm as Naïve Space Efficient Viterbi.

#### 3.1 Acyclic Case

We show that we can avoid the time complexity overhead of Naïve Space Efficient Viterbi. To do so, we define the concepts of predecessor state, successor state, and candidate median state.

**Definition 1 (Predecessor State).** Given a state \( s \) in a universe of \( K \) states \( S \) that defines a HMM, another state \( s' \) is a predecessor state of \( s \) iff there is a directed path from \( s' \) to \( s \) in the HMM graph \( G \).

**Definition 2 (Successor State).** Given a state \( s \) in a universe of \( K \) states \( S \) that defines a HMM, another state \( s' \) is a successor state of \( s \) iff there is a directed path from \( s \) to \( s' \) in the HMM graph \( G \).

In an HMM defined by a cyclic transition graph \( G \), a state \( s' \) may be both a predecessor and successor of \( s \). However, in an HMM defined by a directed acyclic graph (DAG), a state \( s' \) cannot be both a predecessor and successor of another state \( s \).

**Definition 3 (Candidate Median State).** A state \( s_m \) in a universe of \( K \) states \( S \) that defines a HMM is a candidate median state iff it has at most \( \frac{K}{2} \) predecessors and at most \( \frac{K}{2} \) successors.

**Theorem 1.** An HMM defined by a simple DAG \( G \) has at least one candidate median state.

**Proof.** Consider a simple DAG of \( K \) nodes, \( G \). Since \( G \) is a simple DAG (i.e., has no double edges, no doubly-directed edges, and no cycles), it has at most \( \frac{K(K-1)}{2} \) edges. In other words, it may correspond, in the worst case, to an acyclic orientation of a complete undirected graph. Consider a topological ordering of \( G \), and any path \( P = [u_1, u_2, \ldots, u_{|P|}] \) from a root to a leaf in \( G \). As we traverse \( P \), the number of predecessors of the current node \( u_i \) increases, and the number of successors decreases. Assume a median state does not exist in \( P \), i.e., each node \( u_i \) in \( P \) has either more than \( \frac{K}{2} \) predecessors, or more than \( \frac{K}{2} \) successors. Given this monotonicity, there exists a transition in \( P \) from a node \( u \) having more than \( \frac{K}{2} \) successors, to a node \( v \) having more than \( \frac{K}{2} \) predecessors. These two sets of nodes cannot be disjoint, as then \( G \) would have more than \( K \) nodes. Thus, the successors of \( u \) and the predecessors of \( v \) contain at least one common element \( w \). Since \( w \) is a successor of \( u \), it has fewer successors than \( u \) by at least 1, and since \( w \) is an
predecessor of \( v \), it has fewer predecessors than \( v \) by at least 1. If any of those sets has cardinality higher than \( \frac{k}{2} \), we proceed in the same way to find a sequence of \( w', w'' \ldots \), each in the intersection of two sets having more than \( \frac{k}{2} \) elements each, having progressively fewer predecessors than \( v \) and fewer successors than \( u \). Eventually we arrive at a node having no more than \( \frac{k}{2} \) predecessors and no more than \( \frac{k}{2} \) successors, i.e., a candidate median state.

An HMM graph may contain several candidate median states.

**Definition 4 (Pre-Median State).** A state \( s_m^- \) in a universe of \( K \) states \( S \) that defines a HMM is called pre-median state if it has more than \( \frac{k}{2} \) successors states in the HMM graph \( G \).

**Definition 5 (Post-Median State).** A state \( s_m^+ \) in a universe of \( K \) states \( S \) that defines a HMM is called post-median state if it has more than \( \frac{k}{2} \) predecessors in the HMM graph \( G \).

Note that a state cannot have both more than \( \frac{k}{2} \) predecessors and successors in \( G \); thus, a pre-median state has at most \( \frac{k}{2} \) predecessors and a post-median state has at most \( \frac{k}{2} \) successors.

**Definition 6 (Median Pair).** A pair of consecutive states, \( s_m^− \) and \( s_m^+ \), in a path \( P \) over a universe of \( K \) states \( G \) is a median pair of \( P \) iff \( s_m^- \) has at most \( \frac{k}{2} \) predecessors and \( s_m^+ \) at most \( \frac{k}{2} \) successors in the HMM graph \( G \), i.e., iff any of the following holds:

1. at least one of \( s_m^- \) \( s_m^+ \) is a median state; or
2. \( s_m^- \) is a pre-median state and \( s_m^+ \) is a post-median state.

To divide a problem instance into two subproblems, SIEVE finds a median pair in the Viterbi path \( P \) that minimizes the total count of predecessor states of \( s_m^- \), \( N_m^- \), and successor states of \( s_m^+ \), \( N_m^+ \). To do so, in a pre-processing stage, it counts the numbers of predecessors \( N_i^- \) and successors \( N_i^+ \) of each node \( v_i \) in the HMM DAG \( G \). While computing (and re-computing) the DP recursion of Equation 2, we use these counts to decide whether two consecutive nodes in any path of \( j \) observations ending at state \( s_i \) form a candidate median pair \( M[i,j] \) of that path. If a newly found candidate median pair \(( s_i, s_j \) \) has a lower sum of \( N_i^- + N_j^+ \) than the running \( M[i,j] \), we update \( M[i,j] \) to \(( s_i, s_j \) \). The size \( n \) of the problem instance we work with is the number of predecessor states of \( s_i \) in \( G \). We decide whether two consecutive nodes constitute a candidate median pair based on this value of \( n \) throughout the DP run that gives the predecessor state of \( s_i \), even when working with intermediate states \( s_k \) in that run; we move to another value of problem size \( n \) only when we re-initiate a DP calculation with a final state \( s_i \). We tabulate each median pair \( M[i,j] \) and the number of path elements preceding it, \( N[i,j] \), along with probabilities \( T[i,j] \). These pairs are recursively propagated across values of path length \( j \) for the same \( s_i \); if \( M[k,j-1] \) is defined and \( T[k,j-1] \) is assigned the value of \( T[k,j-1] \), then \( M[k,j] \) is assigned the value of \( M[k,j-1] \). Notably, the middle pair of a path \( P \) is independent of the length of \( P \). The only exception to this rule is that, if the middle pair \( M[k,j-1] \) consists of the last two nodes in a \(( j-1)\)-node path \( P \), we may update \( M[i,j] \) to a more preferable middle pair, so as to minimize the total count \( N_m^- + N_m^+ \), upon extending \( P \) from \( j-1 \) to \( j \) nodes.

Having tabulated middle pairs, we perform the re-computation for the sake of space efficiency in a time-efficient manner. Instead of re-running the algorithm up to the last known node in the path, we rerun two problem instances of size at most \( \frac{k}{2} \): one problem to find a most likely path of \( N[i,j] \) states ending at \( s_m^- \), working with the \( N_m^- \leq \frac{k}{2} \) predecessors of \( s_m^- \) in \( G \), and another to find a most likely path of \( j - N[i,j] - 1 \) states ending at \( s_j \), working with the \( N_m^+ \leq \frac{k}{2} \) successors of \( s_m^+ \) in \( G \). To do so, we only need to work with \( \{N[i,j]\} \)-hop predecessors of \( s_m^- \) and the \( \{N[i,j]\} \)-hop successors of \( s_m^+ \); all other states in the HMM are pruned. We thus move from a problem of size \( K \) to two subproblems of size at most \( \frac{k}{2} \), in divide-and-conquer fashion, and repeat recursively. Thereby, we achieve space complexity \( O(K) \) and time complexity \( O \left( \sum_{i=1}^{\log K} \left( \frac{k}{2} \right)^2 T \right) = O \left( \sum_{i=1}^{\log K} \frac{k^2}{2} T \right) = O(K^2T) \) in the default case and \( O \left( \sum_{i=1}^{\log K} |E|T \right) = O(|E|T \log K) \) in the edge-aware case.

![Figure 3: An acyclic HMM, Viterbi path (thick arrows), median pairs (triangles), predecessors (squares) and successors (diamonds) across two levels of the recursion in SIEVE.](image)

Figure 3 shows an example of a directed acyclic HMM graph. The edges of a Viterbi path are highlighted in purple. The two nodes of the median pair are indicated as triangles, the predecessors as squares and the successors as diamonds. SIEVE detects this median pair of the Viterbi path and recursively recomputes the optimal subpaths among its predecessors and successors. The nodes indicated as circles are neither predecessors nor successors of the median pair, hence are pruned in the next level of the recursion.

### 3.2 General Case

The above analysis holds in the case the HMM is a DAG, hence covers some real-world applications such as speech recognition. In case there are cycles in the HMM graph, we can still define a median pair, provided there is no path from \( s_m^- \) to \( s_m^+ \). Otherwise, a complication arises. In the presence of cycles in \( G \), a node \( v_i \) may be reachable from another node \( v_j \) in \( G \) at the same time \( v_j \) may be reachable from \( v_i \). Consequently, there is no guarantee that there exists a median pair that partitions the problem into two disjoint subproblems, each having no more than half the number of states in the parent problem. Even so, we generalize the median pair notion in the presence of cycles. Ideally, we would like to split the problem into two parts of exactly equal size. We define a generalized median pair, or pseudomedian pair, as a pair of states, \(( s_m^- \), \( s_m^+ \)), that minimizes the largest of the number of predecessors of \( s_m^- \) and successors of \( s_m^+ \). Formally:

**Definition 7 (Pseudomedian Pair).** A pair of consecutive states, \( s_m^- \) and \( s_m^+ \), in a path \( P \) over a universe of \( K \) states \( G \) is a pseudomedian pair iff:
\[(s_m^*, s_p^*) \leftarrow \arg\min_{(s_a, s_p) \in E^P} \{N_{s_a}^-, N_{s_p}^+\}\] (3)

We adopt this definition for all graphs, whether cyclic or acyclic, and refer to pseudomedian pairs as median pairs. In both acyclic and cyclic HMM graphs, SIEVE collects, in a pre-processing stage, for each node \(s_j\) in \(G\), the number of its predecessors \(N_{s_j}^-\) and successors \(N_{s_j}^+\) within a \(T\)-hop neighbourhood of \(s_j\). In the acyclic case, this operation amounts to a linear scan over nodes and is repeated for each subproblem in each iteration. In the cyclic case, we explore the \(T\)-hop neighbourhood of each node in a breadth-first manner, counting each node only once; by default, we do so only in the first iteration, and reuse those counts throughout the algorithm. Eventually, the presence of cycles in the HMM graph does not impede the applicability of SIEVE, and, as we will see in our empirical evaluation, may only slightly increase its execution time; thus, we can apply SIEVE even in the presence of cycles.

Algorithm 1 SIEVE

```
input initStat, lastStat ; \pi, \alpha, \beta; y; N^-, N^+

1. \( t \leftarrow \{}\) size(); \( K^' \leftarrow \{}\) size()
2. if initStat \emptyset then
3. \( \pi \leftarrow 0 \) known initial state
4. \( \pi[initStat] \leftarrow 1 \)
5. \( T_i \leftarrow \{ y[i], y[1] \} \) int previous values
6. \( M_v^+ \leftarrow \{ m_v, M_v^+ \} = 0 \)
7. for \( j = 2, \ldots , K^' \) do
8. \( M_v^+ \leftarrow m_v, M_v^-, T_j \leftarrow 0 \) int current values
9. for \( i = 1, \ldots , K^' \) do
10. \( T_i[1] \leftarrow \max_{M[k][y[i]]}[\alpha[i,k]] \) \( A[k][i] \) \( B(i, y[j]) \)
11. \( x^* \leftarrow \arg\max_{M[k][y[i]]}[\alpha[i,k]] \) \( B(i, y[j]) \)
12. \( m \leftarrow \max_{M[k][y[i]]}[\alpha[i,k]] \)
13. if \( m < M_v^+[x^*] \) then
14. \( M_v^+[x^*] \leftarrow (x^*, i), N_v^+[x^*] \leftarrow j \) new median pair
15. else
16. \( M_v^+[x^*] \leftarrow M_v^+[x^*], M[1][x^*] \leftarrow M_v^+[x^*] \) inherited median pair
17. \( M_v^+[x^*] \leftarrow M_v^+[x^*], M[1][x^*] \leftarrow N_v^+[x^*] \) update previous values
18. if lastStat \emptyset then
19. \( lastStat \leftarrow \arg\max_T T_1 \)
20. \( s_{mp}\{lastStat\}[y[p]] \leftarrow \arg\max_T T_1 \) extract median pair
21. \( N_p \leftarrow N_p[lastStat] \) \( y[p] = y[p] \) \( N_p \)
22. if \( N_p > 1 \) then
23. \( s_{mp}\{lastStat\}[y[p]] \leftarrow B(lastStat) \) \( y[p] = \pi[lastStat] \) \( B(lastStat) \)
24. \( s_{mp}\{lastStat\}[y[p]] \leftarrow B(lastStat) \) \( y[p] = \pi[lastStat] \) \( B(lastStat) \)
25. \( s_{mp}\{lastStat\}[y[p]] \leftarrow B(lastStat) \) \( y[p] = \pi[lastStat] \) \( B(lastStat) \)
26. \( s_{mp}\{lastStat\}[y[p]] \leftarrow B(lastStat) \) \( y[p] = \pi[lastStat] \) \( B(lastStat) \)
27. \( N_v^+ \leftarrow t \) \( N_p \leftarrow y[p] \) \( y_N[p] \)
28. if \( s_{mp}\{lastStat\}[y[p]] \leftarrow \) in-order print
29. if \( N_p > 1 \) then
30. \( s_{mp}\{lastStat\}[y[p]] \leftarrow \) in-order print
31. if \( N_p > 1 \) then
32. \( s_{mp}\{lastStat\}[y[p]] \leftarrow \) in-order print
33. \( s_{mp}\{lastStat\}[y[p]] \leftarrow \) in-order print
```

Algorithm 1 presents SIEVE. Each iteration handles a subset of the original state space of size \(K' \leq K\) and a subset of observations of length \(t \leq T\). We solve each subproblem as in standard Viterbi (Lines 7–11), but do not tabulate results. Instead, we use current and previous values of four arrays of size \(K'\); the \(i\)-th entry of each array refers to a path ending at \(i\); array contents are updated as we move from path length \(j\) to \(j + 1\), while maintaining previous values to enable recursive computations. Arrays \(M\) and \(M_v\) store median pairs \((s_m, s_p)\) and their scores (i.e., the maximum number among \(N^-\) and the count of predecessors of \(s_m\)), and \(N^+\), the count of successors of \(s_p\); array \(N_o\) stores the number \(N_o\) of observations that lie before the found median pair, whence we also calculate the number \(N_s\) of observations that lie ahead of the median pair; whenever a new median pair is found, the arrays \(N_o\), \(M\), and \(M_v\) are updated accordingly (Line 14); lastly, array \(T_i\) stores path probabilities, initialized using the initial distribution \(\pi\) and the emission probabilities of the first observation (Line 5).

In each iteration, SIEVE moves on to the two subproblems among the \(N^-\) predecessors and the \(N^+\) successors of the detected median pair \((s_m, s_p)\) stored in \(M\) for the last state in the solution, to find subpaths of length \(N_o\) and \(N_p\), respectively. In each recursive call after the first, the start or end state of the path to be retrieved is known, given the states of the median pair \((s_m, s_p)\). To find the number of predecessors and successors, in the acyclic case, we perform BFS traversals and return the visited nodes (Lines 23 and 30); in the cyclic case, by default we count such nodes only in the first iteration and reuse those counts throughout the algorithm. We discuss another option in Section 3.5. The recursion stops when there are no more observations to be explained (Lines 22 and 29).

To retrieve the Viterbi path \(P^*\) after finding the final state, standard Viterbi backtracks over the \(O(TK)\) tabulated solutions from last to first observation. SIEVE, on the other hand, builds \(P^*\) by returning the detected median pairs in inorder fashion (Line 28), after the recursive call to left-side subproblem and before the call to the right-side subproblem; when reaching a leaf of the recursive tree, it returns the corresponding median pair. Figure 4 shows the relationship between the preorder binary tree traversal representing the order of recursive calls and the inorder traversal generating the Viterbi path. To avoid numerical errors, in our implementation we add log-probabilities rather than multiplying raw probability values. While SIEVE’s complexity bounds rely on the existence of median states, which is guaranteed on acyclic HMM graphs, as Theorem 1 shows, the algorithm works on cyclic graphs too.

3.3 Proof of Correctness
To demonstrate the correctness of SIEVE, we show that it returns the same solution as the standard Viterbi algorithm.

Theorem 2. SIEVE retrieves the optimal Viterbi path.
Proof. Each path element SIEVE outputs is detected as a median pair of the original Viterbi path or one of its subpaths that are recomputed in the process. Each recomputed subpath is optimal in its subproblem, has start and end points delivered by its ancestor solutions in the recursion, and resolves ties by the same procedure as the ancestor solution. Thus, each element SIEVE outputs comprises part of the optimal Viterbi path. In reverse, since Viterbi and SIEVE output paths of the same length, by the pigeonhole principle it follows that each part of the optimal Viterbi path is output
by SIEVE. Thus, SIEVE outputs nothing more, and nothing less, than classic Viterbi. As SIEVE prints out each median pair after the subpath among its predecessors and before the subpath among its successors, it returns Viterbi path edges in the correct order.

### 3.4 SIEVE-Middlepath

As seen in Section 3.1, SIEVE reduces the space complexity of the Viterbi algorithm, while guaranteeing a worst-case time complexity of $O(K^2T)$ on acyclic HMM graphs, in which a median pair divides the state space in two parts of at most half the initial size. However, this hypothesis does not hold for general graph topologies; in the worst case, if the HMM graph is a complete directed graph, every node has $K$ one-hop predecessors and $K$ one-hop successors, hence the concept of a median pair is trivialized: all pairs along a path are median pairs. In the worst case, if we choose a terminal pair along a path as median pair, the worst-case time complexity rises to $O(K^2T^2)$, as in Naïve Space Efficient Viterbi.

To address problem instances approaching such a worst-case scenario, we modify the criterion that the dividing pair in SIEVE’s divide-and-conquer recursion should satisfy: instead of looking for a median pair of states that divides the state space evenly among its predecessors and successors, we select the middle pair of states that divides the Viterbi path of length $T$ into two halves, i.e., subpaths of length at most $T/2$. We refer to this alternative formulation of SIEVE as SIEVE-Middlepath; its space complexity remains $O(K)$, while its time complexity is $O(K^2T \log T)$, regardless of graph type. Moreover, SIEVE-Middlepath raises practical space requirements more lightweight than those of standard SIEVE, since the former needs to store neither any numbers of predecessors and successors per node, nor observations before a middle pair, as the latter does. However, as we will see in Section 4, standard SIEVE is the preferred algorithm in terms of runtime dependence on observation sequence length $T$ not only in acyclic graphs, where it has a time-complexity advantage, but also in general graphs.

### 3.5 HyperLogLog Counting in Cyclic Case

In the case of acyclic HMM graphs, the recounting of $t$-hop predecessors and successors within each subproblem (Algorithm 1, Lines 23 and 30) is necessary to efficiently identify median pairs. Yet, in the case of cyclic HMM graphs, it would incur a significant runtime overhead. As discussed in Section 3.2, by default in the cyclic case SIEVE counts $T$-hop neighbours only in the first iteration and reuses those counts in subsequent iterations. Here, we examine whether we can enhance this computation using an efficient approximate counting procedure that would allow counting within each subproblem. To that end, we consider HyperLogLog [9], a probabilistic algorithm that approximates the cardinality (i.e., number of distinct elements) of a multiset, starting from the idea that, in random data in base $b$, a sequence of $x$ zeros occurs in average once in every $b^x$ elements, thus the cardinality of a multiset of uniformly distributed random numbers can be estimated as $2^x$, where $x$ is the maximum number of leading zeros in those hashes. To increase the robustness of the algorithm, we may aggregate the results of multiple independent hash functions; more conveniently, we may simulate multiple independent hash functions by splitting the items into buckets and aggregating the estimates from those buckets by a harmonic mean. This procedure yields an asymptotically almost unbiased estimator with bounded variance.

HyperLogLog has been applied to approximate the neighbourhood function of large graphs [3], i.e., the number of pairs of nodes such that one node is reachable from the other in less than $t$ hops, estimated as the sum of the sizes of $t$-balls centered at each node, using HyperLogLog counters. We adapt this usage of HyperLogLog counters to our purposes, where we do not need to approximate the sum over all nodes, but only the set of $t$-hop predecessors and successors of specific nodes. For a fixed precision, the memory usage for cardinality estimation based on HyperLogLog scales almost linearly, as $O(K \log \log K)$. We evaluate this method, among others, in Section 4. As we will see, SIEVE with HyperLogLog performs slightly worse than the default SIEVE which uses the counts of $T$-hop neighbours obtained in the first iteration.

### 4 EXPERIMENTS

In this section, we present our experimental evaluation to assess the performance of SIEVE on synthetic and real-world data.

#### 4.1 Experimental Setting

Here, we describe the data used, the baselines we compare against, the parameter settings, and the performance metrics.

**Data.** We obtain synthetic HMMs by generating Erdős–Rényi transition graphs, discrete uniform emission probabilities with a vocabulary size fixed to $|O| = 50$ and vectors of integer-valued observations sampled uniformly from $\{1, \ldots |O|\}$. We generate Erdős–Rényi transition graphs where each edge exists with default probability $p = 0.2$ independently of others. To investigate the impact of parameter $p$ we carry out experiments varying $p$ in a geometric progression. We also use a real-world composite HMM for speech-text forced alignment. The model is built using the HTK software toolkit [27], containing 5529 states out of which 3204 are emitting, while the remaining are non-emitting, aiming to align speech recordings from the TIMIT corpus [11]. The 5-state (3 emitting, 2 non-emitting), phoneme-level, context-dependent HMMs were trained using the WSJ corpus for continuous speech recognition [17, 22]. Transition and probabilities and observation probability distributions (multivariate GMMs) are learned using the Baum-Welch algorithm [27]. As observations, we extract a standard 39-dimensional feature vector (13 Mel-Cepstrum Cepstral Coefficients, augmented by their first and second order derivatives). In the case of synthetic data, $G$ contains arbitrary cycles. On the other hand, in forced alignment data, cycles are present only in transitions within small sequences of phonemes associated with silent breaks. As we will see, such localized cycles bear no effect on the performance of SIEVE.

**Baselines.** We compare SIEVE in terms of space and time requirements against (i) Vanilla Viterbi, (ii) Checkpoint Viterbi and (iii) Naïve Space Efficient Viterbi; the first is the standard Viterbi algorithm using an $O(KT)$ tabulation to recover the optimal path; the second is the checkpointing approach [21] mentioned in Section 2.3,
which stores rows of the dynamic programming matrix at intervals of size $\sqrt{T}$, reducing space complexity to $O(K\sqrt{T})$; the last is the naive space-efficient approach discussed in Section 3 that runs standard Viterbi $T = 1$ times without storing the full dynamic programming table, thus lowering space complexity to $O(K)$ at the expense of time complexity growth to $O(K^2T^2)$. We also test an implementation of SIEVE that uses HyperLogLog counters, as described in Section 3.5, henceforth referred to as SIEVE-HyperLogLog, and SIEVE-Middlepath, described in Section 3.4. By our theoretical analysis, SIEVE uses $O(K)$ space and $O(K^2T)$ time guaranteed when the HMM graph $G$ is acyclic, while SIEVE-Middlepath uses $O(K)$ space and $O(K^2T \log T)$ time for general graphs. We aim to examine how runtime and memory usage behave empirically.

**Experiment Parameters.** We study performance with respect to state space size $K$ and the observation sequence length $T$. Unless specified otherwise, we vary the state space size $K$ and observation sequence length $T$ in a geometric progression with ratio 1.5. In the forced-alignment case, we vary $K$ from 9 up to 760 by snowball sampling while fixing $T$ to 250, and vary $T$ from 9 up to 760 by considering subsets of the observation sequence while fixing $K$ to the size of the entire HMM network; with synthetic HMMs, we vary $K$ from 9 to 1,298,2, while fixing $T$ to 500, and vice versa; likewise, when we vary the probability $p$ of edge existence in synthetic HMMs, we fix both $T$ and $K$ to 500.

**Metrics.** We report runtime in seconds and memory usage in bytes; runtime includes pre-processing plus decoding; memory usage reflects the total memory occupied by all data structures and by the input data, including the HMM graph with emission probabilities and the observation sequence. We average results over three executions. Since memory usage in SIEVE varies while solving different subproblems, we report the minimum and maximum memory usage over the recursion as a shaded region, and the median memory usage across recursive calls as a line; notably, the maximum memory usage arises in the first SIEVE call, when the problem is defined over the entire state space and observation sequence.

**Implementation.** Experiments ran on a 2x10 core Xeon E5 2680 v2 2.80 GHz, 256 GB machine, except those exploring different hardware configurations. Algorithms are implemented in Python; the code and forced-alignment data are available online.¹

¹https://github.com/VITERBI-SPACE-EFFICIENT/SIEVE

### 4.2 Results

**Synthetic data.** To investigate the effect of the presence of cycles in the HMM graph, we measure runtime and memory usage against the probability $p$ that a node is connected to any other, varied in a geometric progression with ratio 1.5; this parameter determines the density of the HMM graph and hence the likelihood that cycles are formed. To focus on the effect of cycles as such, rather than the effect of sheer number of edges $|E|$, we use non-edge-aware implementations that retain an $O(K^2)$ rather than $O(|E|)$ time-complexity factor. Figure 5 shows our results. On the baseline algorithms, the growth of $p$ only slightly increases the cost of storing the input. On SIEVE variants, we discern that the minimum and median memory requirements increase with $p$, which is a reasonable outcome as the number of nodes in the explored $T$-hop neighbourhoods of median/middle states grows with graph density. However, the growth of $p$ has little effect on maximum memory consumption and execution time. Overall, as resource use does not grow significantly with $p$, we infer that the presence of cycles as such has little effect on the algorithms under comparison.

Figure 6 shows runtime and memory usage results vs. state space size $K$ and observation sequence length $T$ on synthetic data. While the time complexity of both SIEVE and Vanilla Viterbi is $O(K^2T^2)$, the former’s runtime is larger as it reiterates the main task of Vanilla Viterbi a logarithmic number of rounds. Nevertheless, by virtue of SIEVE’s amortization of runtime and pruning of the state space size across iterations, its runtime overhead is low and asymptotically matches that of Vanilla Viterbi. On the other hand, Naïve Space Efficient Viterbi incurs quadratic runtime in both $K$ and $T$ due to its $O(K^2T^2)$ time complexity. In terms of memory usage, the peak memory of SIEVE, reached in the first iteration, is slightly larger than that of Naïve Space Efficient Viterbi due to the additional memory required to identify the median pair; yet the median memory usage of SIEVE across iterations is remarkably lower. Most remarkably, the memory consumption of SIEVE and Naïve Space Efficient is independent of $T$ and more than two orders of magnitude lower than that of Vanilla Viterbi in all cases, while that of Vanilla Viterbi grows with $T$, due to the storage of the $K \times T$ dynamic programming tables. The variant of SIEVE using HyperLogLog counters performs slightly worse than SIEVE in both runtime and peak memory; this result indicates that the overhead of counting in each iteration to find more precise medians, rather than reusing precomputed counts, does not pay off in terms of reduced runtime. On the other hand, SIEVE-Middlepath requires lower median runtime.
in the figure) than the other two SIEVE variants; even though it may work with larger state space sizes in subproblems than standard SIEVE, it stores neither numbers of predecessors and successors per node, nor observations before middle pairs, as discussed in Section 3.4. However, the difference is rather negligible in terms of peak memory (shown by shaded areas). For small path length $T$, SIEVE-Middlepath outperforms standard SIEVE in runtime, as it performs neither pre-processing nor median-pair-finding; for the smallest values of $T$ and $K$, SIEVE-Middlepath outperforms even standard Viterbi in runtime, as its lightweight storage of middle pairs directly provides the information standard Viterbi obtains via backtracking over tabulated solutions. Still, as path length $T$ grows, SIEVE-Middlepath’s log-linear time complexity dependence on $T$ manifests itself. Lastly, Checkpoint Viterbi has similar runtime to SIEVE, yet consumes much larger memory that grows with $T$.

**Forced Alignment.** Figure 7 presents our results on forced alignment. The naïve baseline again presents non-scalable runtime, while SIEVE and Viterbi variants exhibit similar trends. SIEVE variants and Naïve Space Efficient Viterbi require memory about two orders of magnitude lower than Vanilla Viterbi and about one order of magnitude lower than Checkpoint Viterbi; whereas the maximum memory SIEVE uses is slightly larger than that of the naïve baseline,

**Figure 6:** Synthetic data: runtime in seconds, memory usage in bytes vs. $K$ and $T$ (log-log scale).

**Figure 7:** Forced Alignment: runtime in seconds, memory usage in bytes vs. $K$ and $T$ (log-log scale).
We introduced SIEVE, a reformulation of the long-standing Viterbi algorithm that reduces space complexity with a negligible runtime overhead. SIEVE has the same time complexity as standard Viterbi on acyclic graphs and also performs well on generic HMM graphs. We also provide a variant, SIEVE-Middlepath, whose time complexity bears an added factor logarithmic in path length, but is valid on all graph types. Our experimental evaluation on synthetic data and a real-world application scenario demonstrates that SIEVE variants consistently provide a golden spot between memory usage and runtime, yielding runtime on par with Vanilla Viterbi and memory usage on par with a naive space-efficient variant. In effect, the SIEVE paradigm stands to contribute to the speech recognition capabilities of low-memory Internet of Things (IoT) devices. In the future, we aim to implement SIEVE as part of open-source deep neural network-based speech recognition toolkits (e.g., Kaldi [19]) to enable space-efficient large-vocabulary decoding, and parallelize it using GPU accelerators in the spirit of [4].

5 CONCLUSIONS
REFERENCES


