SIEVE: A Space-Efficient Algorithm for Viterbi Decoding

Martino Ciaperoni
Aalto University

Aristides Gionis
KTH Royal Institute of Technology

Athanasios Katsamanis
Athena R.C., Behavioral Signals

Panagiotis Karras
Aarhus University

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

ACM Reference Format:
Martino Ciaperoni, Aristides Gionis, Athanasios Katsamanis, and Panagiotis Karras. 2022. SIEVE: A Space-Efficient Algorithm for Viterbi Decoding. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3514221.3526170

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 in *speech recognition* [4, 10], i.e., finding the most likely sequence of utterances for an input acoustic signal. Other applications are found in networking and telecommunications; international standards for second- and third-generation cellular phones employ convolutional codes and Viterbi decoding [24].

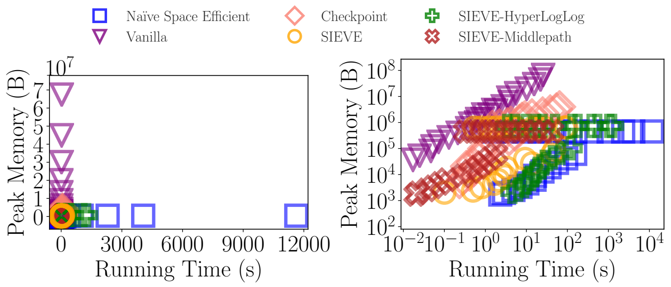

Figure 1: Peak memory vs. runtime with growing path length or state-space size; shown in both linear and log scale.

As a concrete real-world data science application, consider speech recognition [10]. Modern large-vocabulary continuous speech-recognition systems are based on HMMs specified by transition probabilities between states (words and their phonemes) and emission probabilities, which indicate the likelihood of hearing a specific sound signal when the model is in a given state. In this context, Viterbi decoding runs on a composite HMM comprising many smaller HMMs, one for each basic speech unit (a phoneme in a particular context, e.g., /*ah*/ preceded by /*s*/ and followed by /*t*/, as in the word 'sat'), to recognize long speech utterances, which may correspond to multiple sentences.

Unfortunately, despite this large range of applications, the Viterbi algorithm's default dynamic-programming formulation incurs space complexity linearly dependent in *both* observations and states; the space-complexity dependence on the number of observations constrains the algorithm's scalability and applicability to large problem instances and embedded systems with hard memory constrains [12]. For instance, in our speech recognition experiments, standard Viterbi takes up to 50 Megabytes to tabulate intermediate solutions by dynamic programming; such memory usage typically exhausts the limited memory resources of embedded systems and handheld devices, where this pipeline is often performed and for which memory is the key constraint [12, 26].

To enable Viterbi decoding in constrained memory, in this paper we introduce SIEVE: a space-efficient reformulation of the Viterbi algorithm that eliminates its space-complexity dependence on the number of observations (i.e., path length) by introducing a divide-and-conquer element in its design. SIEVE does not incur a time-complexity overhead on acyclic HMMs. We also provide a variant, SIEVE-Middlepath, whose time complexity bears an added factor that is logarithmic in path length on all graph types.

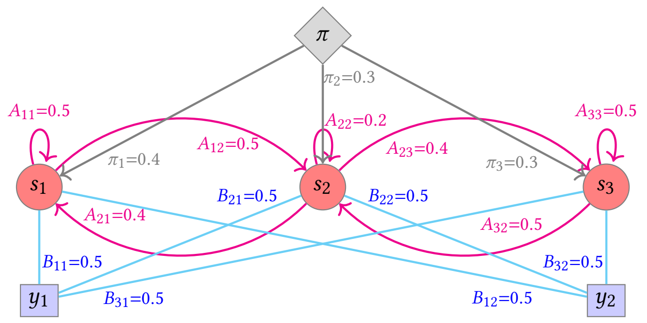

Figure 2: An example HMM with initial distribution π and three states s_1, s_2, s_3 emitting two discrete symbols y_1, y_2.

In our experimental study, we evaluate SIEVE on synthetic and real-world data derived from a speech recognition application against (i) standard (*Vanilla*) Viterbi decoding, (ii) a *checkpointing* variant that achieves limited space efficiency, and (iii) a *naïve space-efficient* variant that confers space-efficiency at the price of a time-complexity overhead. Figure 1 summarizes results reported on in Section 4.2. SIEVE achieves running time comparable to that of Vanilla Viterbi and memory usage comparable to that of the naïve space-efficient variant, while avoiding both the memory overhead of the former and the runtime overhead of the latter.

2 BACKGROUND AND RELATED WORK

A hidden Markov model (HMM) is expressed by a set of K states S, a set of N observable symbols (or vocabulary) O, a vector π specifying initial probabilities of starting in each state, a matrix A specifying probabilities of transition between states and a matrix B specifying probabilities of observing a certain symbol in any state; the entries of π, as well as those of A and B corresponding to each state sum to 1. An HMM is fully and uniquely described by the triplet (π, 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 $Q = \{s_1, s_2, \ldots, s_T\}$ is:

$$P(Q, Y) = \pi_{s_1} \cdot B_{s_1 y_1} \prod_{i=2}^{T} A_{s_{i-1} s_i} \cdot B_{s_i y_i}, \tag{1}$$

where $\pi(s_1)$ is the probability of observing the initial state s_1, $A_{s_{i-1}s_i}$ is the probability of transiting from state s_{i-1} to state s_i, and $B_{s_i y_i}$ is the probability of observing y_i at state s_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 $Q = \{s_1^*, s_2^*, \ldots, s_T^*\}$ 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 (that's essentially a 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 will 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 $Q = \{s_1^*, s_2^*, \ldots, s_T^*\}$, 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:

$$\begin{aligned} \mathbf{T}[i,1] &= \pi_i \cdot B_{iy_1}, \\ \mathbf{T}[i,j] &= \max_{s_k \in \mathcal{N}_{in}(s_i)} \{\mathbf{T}[k, j-1] \cdot A_{ki}\} \cdot B_{iy_j} \end{aligned} \tag{2}$$

Here, $\mathbf{T}[i, j]$ 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 $\mathcal{N}_{in}(s_i)$ of s_i in the HMM graph; π_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 $\mathbf{T}[i, j]$ 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 $\mathbf{T}[i, j]$ 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 Hoeher [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, Ardö 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.

$T; t$	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
π	Initial state probabilities
A	State transition probability matrix
O	Vocabulary
B	Emission probability matrix
s_{m^-}, s_{m^+}	Median pair
$N_i^-; N_i^+$	Predecessors; successors of state s_i
M	Array of median pairs
M_v	Array of median-pair scores
$N_p; N_s$	Number of observations before; after the median pair
N_o	Array of number of observations before the median pair
T_1	Array of path probabilities

Table 1: Notations.

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 $\mathbf{T}[i, j]$ values throughout the run of the algorithm, we keep only two rows at any time, $\mathbf{T}[*, j]$ and $\mathbf{T}[*, j-1]$. 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, 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 $T - i - 1$. By this approach, space complexity falls to $O(K)$, yet time complexity rises to $O\left(\sum_{i=1}^{T} K^2 i\right) = O\left(K^2 T^2\right)$ in the default case and $O\left(\sum_{i=1}^{T} |E| i\right) = O(|E|T^2)$ 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 iff *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 iff *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^-_ℓ and successors N^+_ℓ of each node v_ℓ in the HMM DAG G. While computing (and recomputing) 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 (ℓ, ℓ') has a lower sum of $N^-_\ell + N^+_{\ell'}$ than the running $M[i,j]$, we update $M[i,j]$ to (ℓ, ℓ'). 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 new 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 $\mathbf{T}[i,j]$. These pairs are recursively propagated across values of path length j for the same s_i: if $M[k, j-1]$ is defined and $\mathbf{T}[i,j]$ is assigned the value of $\mathbf{T}[k, j-1]$, then $M[i,j]$ 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 rerunning 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 $N^-_{m^-} \le \frac{K}{2}$ predecessors of s_{m^-} in G, and another to find a most likely path of $j - N[i,j] - 1$ states ending at s_i, working with the $N^+_{m^+} \le \frac{K}{2}$ successors of s_{m^+} in G. To do so, we only need to work with the $N[i,j]$-hop predecessors of s_{m^-} and the $(T - N[i,j])$-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 $O\left(\sum_{i=1}^{\log K} 2^i \left(\frac{K}{2^i}\right)^2 T\right) = O\left(\sum_{i=1}^{\log K} \frac{K^2}{2^i} T\right) = O(K^2 T)$ 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.

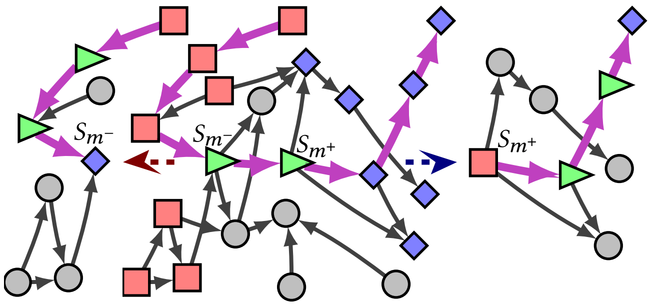

Figure 3: An acyclic HMM, Viterbi path (thick arrows), median pairs (triangles), predecessors (squares) and successors (diamonds) across two levels of the recursion in SIEVE.

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 $v_i \in G$ may be reachable from another node $v_\ell \in G$ and at the same time v_ℓ 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*:*

$$(s_{m^-}, s_{m^+}) \leftarrow \arg \min_{(s_a,s_b)\in P} \max\{N_a^-, N_b^+\} \tag{3}$$

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_i in G, the number of its predecessors N_i^+ and successors N_i^- within a T-hop neighbourhood of s_i. 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 $initSt, lastSt, \pi, A, B, y, N^-, N^+$
1:  $t \leftarrow y.size(); K' \leftarrow \pi.size()$
2:  if $initSt \neq \emptyset$ then                                   ▷ known initial state
3:      $\pi \leftarrow \mathbf{0}$
4:      $\pi[initSt] \leftarrow 1$
5:  $T_1' \leftarrow \pi \cdot B[, y[1]]$                              ▷ init. previous values
6:  $M_v' \leftarrow \infty; M', N_o' \leftarrow \mathbf{0}$
7:  for $j = 2, \ldots, t$ do
8:      $M_v \leftarrow \infty, M, N_o, T_1 \leftarrow \mathbf{0}$         ▷ init. current values
9:      for $i = 1, \ldots, K'$ do
10:         $T_1[i] \leftarrow \max_k T_1'[k] \cdot A[k,i] \cdot B[i, y[j]]$
11:         $s^* \leftarrow \arg\max_k T_1'[k] \cdot A[k,i] \cdot B[i, y[j]]$
12:         $m \leftarrow \max\{N_{s^*}^-, N_i^+\}$
13:         if $m < M_v[s^*]$ then                                   ▷ new median pair
14:             $M_v[i] \leftarrow m; M[i] \leftarrow (s^*, i); N_o[i] \leftarrow j$
15:         else                                                     ▷ inherited median pair
16:             $M_v[i] \leftarrow M_v'[s^*]; M[i] \leftarrow M'[s^*]; N_o[i] \leftarrow N_o'[s^*]$
17:     $M_v', M', N_o', T_1' \leftarrow M_v, M, N_o, T_1$                 ▷ update previous values
18: if $lastSt = \emptyset$ then                                     ▷ known end state
19:     $lastSt \leftarrow \arg\max T_1$
20: $s_{m^-}, s_{m^+} \leftarrow M[lastSt]$                              ▷ extract median pair
21: $N_p \leftarrow N_o[lastSt]; y_p = y[: N_p]$
22: if $N_p > 1$ then                                   ▷ continue recursion in predecessors
23:     $states_p \leftarrow$ Find-t-HopPredecessors$(s_{m^-}, N_p)$          ▷ BFS
24:     $A_p = A[states_p]; B_p = B[states_p]; \pi_p = \pi[states_p]$
25:     update $N^-[states_p], N^+[states_p]$
26:     Sieve$(\emptyset, s_{m^-}, \pi_p, A_p, B_p, y_p, N^-, N^+)$
27: $N_s \leftarrow t - N_p; y_s \leftarrow y[N_s :]$
28: print $(s_{m^-}, s_{m^+})$                                        ▷ in-order print
29: if $N_s > 1$ then                                   ▷ continue recursion in successors
30:     $states_s \leftarrow$ Find-t-HopSuccessors$(s_{m^+}, N_s)$            ▷ BFS
31:     $A_s = A[states_s]; B_s = B[states_s]; \pi_s = \pi[states_s]$
32:     update $N^-[states_s], N^+[states_s]$
33:     Sieve$(s_{m^+}, \emptyset, \pi_s, A_s, B_s, y_s, N^-, N^+)$
```

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 state 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_{m^+}) and their scores (i.e., the maximum number among $N_{m^-}^-$, the count of predecessors of s_{m^-}, and $N_{m^+}^+$, the count of successors of s_{m^+}); array N_o stores the number N_p 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, M_v are updated accordingly (Line 14); lastly, array T_1 stores path probabilities, initialized using the initial distribution π and the emission probabilities of the first observation (Line 5).

In each iteration, SIEVE moves on to the two subproblems among the $N_{m^-}^-$ predecessors and the $N_{m^+}^+$ successors of the detected median pair (s_{m^-}, s_{m^+}) stored in M for the last state in the solution, to find subpaths of length N_s 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_{m^+}). 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 recursion 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.

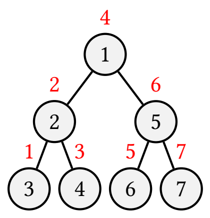

Figure 4: Recursion tree. Black labels within nodes show the preorder recursive calls; red labels above nodes show the inorder returning of median pairs.

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 the binary representation of each number in the set. The algorithm applies to each element in the set a hash function $h : D \rightarrow 2^L$ mapping each element in D to a binary sequence of bits occurring independently with probability $\frac{1}{2}$, and estimates multiset cardinality via 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 asymptoptically 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 that 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,

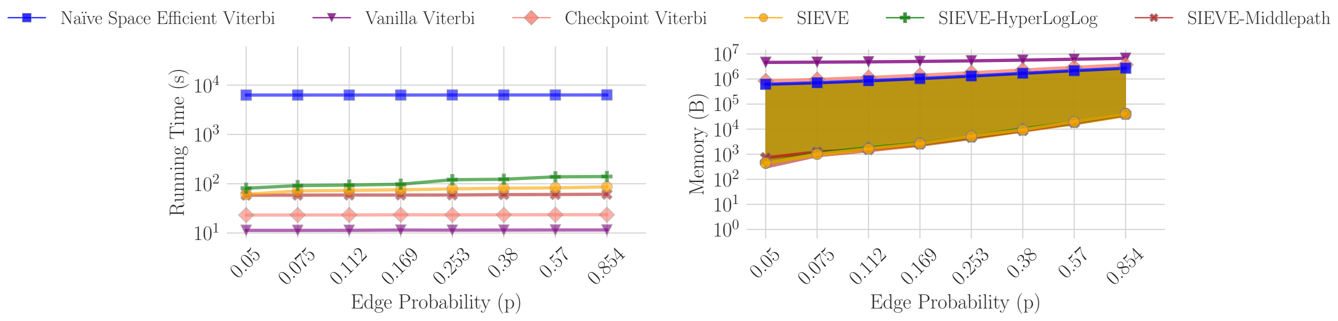

Figure 5: Syntenthic data: runtime and memory usage vs. probability of edge existence (log-log scale), $K = T = 500$.

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 naïve 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 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 12982, 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 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 2×10 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]

[1]https://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 $O(K^2T)$, 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 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 memory (shown by lines

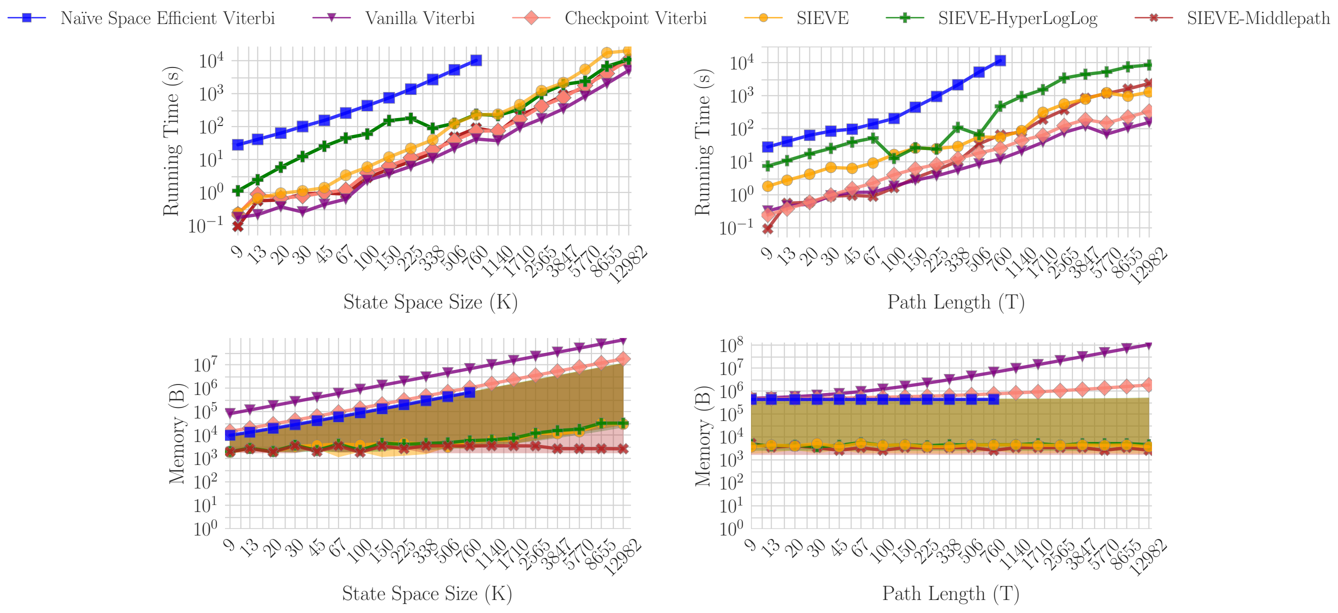

Figure 6: Synthetic data: runtime in seconds, memory usage in bytes vs. K and T (log-log scale).

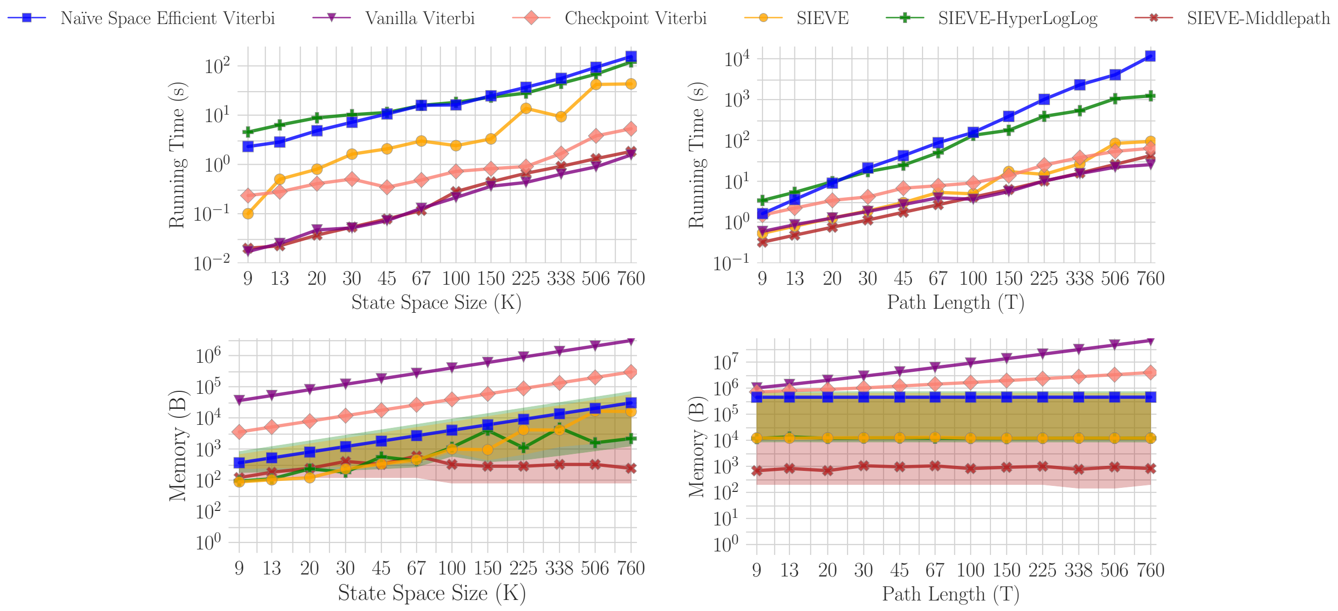

Figure 7: Forced Alignment: runtime in seconds, memory usage in bytes vs. K and T (log-log scale).

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,

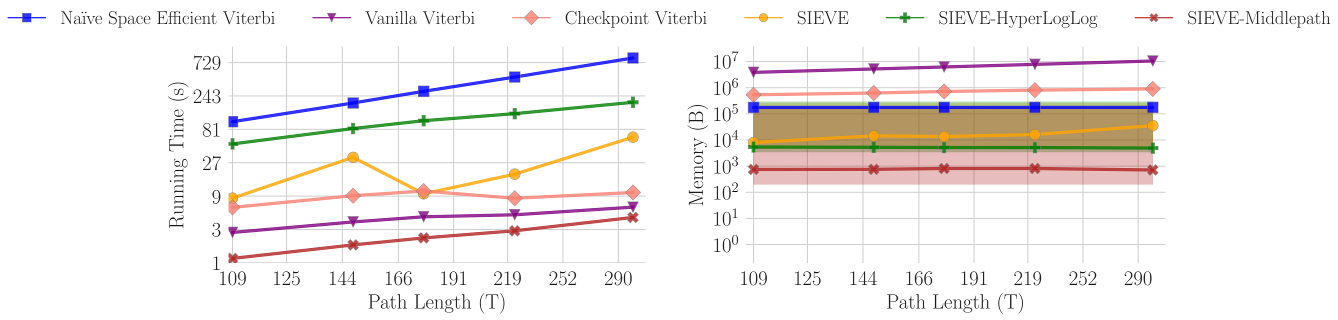

Figure 8: Forced Alignment with varying speech rate: runtime in seconds, memory usage in bytes vs. T (log-log scale).

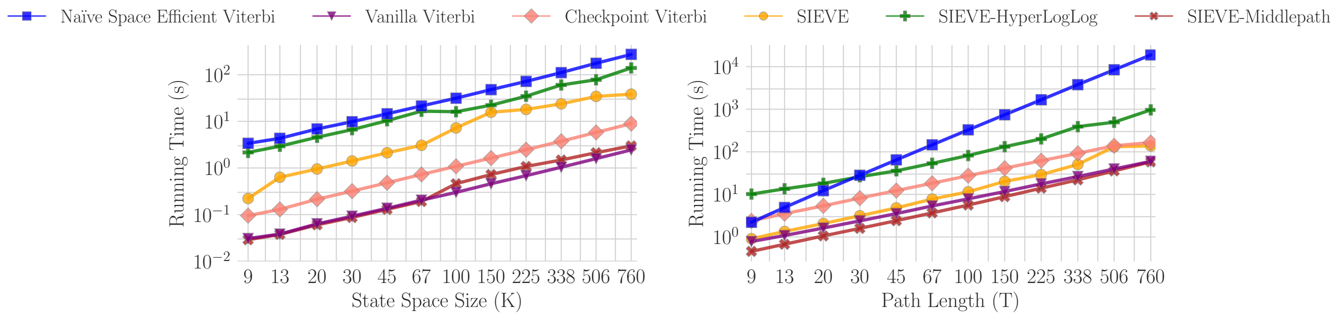

Figure 9: Forced Alignment on smartphone device: runtime in seconds vs. K and T (log-log scale).

its median memory usage is constantly lower, due to the fact that the state space size gradually falls. To illustrate this effect, Figure 10 shows the distribution of the state space size in SIEVE over recursive calls to predecessors and successors of median pairs, as well as the aggregate distribution. The state space size K' in subproblems is never larger than half the initial size K. The performance of the three SIEVE variants mirrors that observed with synthetic data.

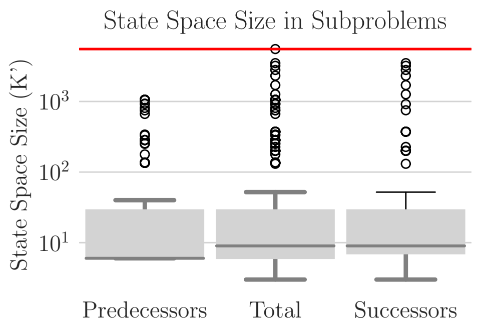

Figure 10: Forced Alignment: distribution of the state space sizes in recursive calls; path length $T = 730$; y-axis in log scale; horizontal red line corresponds to K.

Lastly, we try forced alignment while altering the speech tempo, whereby each observation sequence represents the same sentences uttered at a different pace. This way we obtain realistic observation sequences of varying length, while considering the entire state space with each sequence length. The HMM now includes 2164 states of which 1257 are emitting. Figure 8 shows our runtime and memory usage results vs. path lengths T, whereas results vs. state space size are now inapplicable. These results corroborate the conclusions drawn in previous experiments regarding the time- and space-scalability of SIEVE in comparison to the three baselines.

Low-memory Hardware. To assess the performance of SIEVE in low-performance, low-memory devices, we carry out the forced alignment experiments on a common ANDROID SMARTPHONE with a RAM of 6 GB and 1.80 GHz CPU. Figure 9 shows the results. We observe similar trends to those in preceding experiments, while the quadratic time complexity dependence of Naïve Space Efficient Viterbi on path length T is now more pronounced. This finding suggests that the advantages of SIEVE variants are even more crucial to obtain on low-performance small devices, on which the runtime overhead of the naïve space-efficient solution is prohibitive.

5 CONCLUSIONS

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 naïve 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].

REFERENCES

[1] Håkan Ardö, Kalle Åström, and Rikard Berthilsson. 2007. Real-time Viterbi optimization of Hidden Markov Models for multi target tracking. In *IEEE Workshop on Motion and Video Computing (WMVC)*. 2–2.

[2] Mohamed Benzeghiba, Renato de Mori, Olivier Deroo, Stéphane Dupont, Teodora Erbes, Denis Jouvet, Luciano Fissore, Pietro Laface, Alfred Mertins, Christophe Ris, Richard Rose, Vivek Tyagi, and Christian Wellekens. 2007. Automatic speech recognition and speech variability: A review. *Speech Communication* 49, 10-11 (2007), 763–786.

[3] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. 2011. HyperANF: approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*. 625–634.

[4] Hugo Braun, Justin Luitjens, Ryan Leary, Tim Kaldewey, and Daniel Povey. 2020. GPU-accelerated Viterbi exact lattice decoder for batched online and offline speech recognition. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 7874–7878.

[5] Diamantino Caseiro, Hugo Meinedo, António Serralheiro, Isabel Trancoso, and João Neto. 2002. Spoken book alignment using WFSTs. In *Proceedings of the 2nd International Conference on Human Language Technology Research (HLT)*. 194–196.

[6] George E. Dahl, Dong Yu, Li Deng, and Alex Acero. 2012. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on Audio, Speech, and Language Processing* 20, 1 (2012), 30–42.

[7] Jon Feldman, Ibrahim Abou-Faycal, and Matteo Frigo. 2002. A fast maximum-likelihood decoder for convolutional codes. In *Proceedings of the 56th IEEE Vehicular Technology Conference (VTC)*. 371–375.

[8] Pedro F. Felzenszwalb, Daniel P. Huttenlocher, and Jon M. Kleinberg. 2003. Fast algorithms for large-state-space HMMs with applications to web usage analysis. In *Advances in Neural Information Processing Systems 16 (NIPS)*. 409–416.

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

[10] Mark J. F. Gales and Steve J. Young. 2007. The application of Hidden Markov Models in speech recognition. *Foundations and Trends in Signal Processing* 1, 3 (2007), 195–304.

[11] John S. Garofolo, Lori F. Lamel, William M. Fisher, Jonathan G. Fiscus, David S. Pallett, and Nancy L. Dahlgren. 1993. *DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1.* National Institute of Standards and Technology.

[12] Alexandru-Lucian Georgescu, Alessandro Pappalardo, Horia Cucu, and Michaela Blott. 2021. Performance vs. hardware requirements in state-of-the-art automatic speech recognition. *EURASIP J. Audio Speech Music. Process.* 2021, 1 (2021), 28.

[13] Sudipto Guha. 2005. Space efficiency in synopsis construction algorithms. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. 409–420.

[14] Joachim Hagenauer and Peter Hoeher. 1989. A Viterbi algorithm with soft-decision outputs and its applications. In *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*. 1680–1686.

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

[16] Pedro J. Moreno and Christopher Alberti. 2009. A factor automaton approach for the forced alignment of long speech recordings. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 4869–4872.

[17] Douglas B. Paul and Janet M. Baker. 1992. The design for the Wall Street Journal-based CSR corpus. In *The 2nd International Conference on Spoken Language Processing (ICSLP)*.

[18] Jennifer Pohle, Roland Langrock, Floris M. van Beest, and Niels Martin Schmidt. 2017. Selecting the number of states in hidden Markov models: pragmatic solutions illustrated using animal movement. *Journal of Agricultural, Biological, and Environmental Statistics* 22, 3 (2017), 270–293.

[19] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely. 2011. The Kaldi speech recognition toolkit. In *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*.

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

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

[22] Keith Vertanen. 2008. Combining open vocabulary recognition and word confusion networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 4325–4328.

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

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

[25] Qi Wang, Lei Wei, and Rodney A. Kennedy. 2002. Iterative Viterbi decoding, trellis shaping, and multilevel structure for high-rate parity-concatenated TCM. *IEEE Transactions on Communications* 50, 1 (2002), 48–55.

[26] Reza Yazdani, Albert Segura, José-María Arnau, and Antonio González. 2017. Low-power automatic speech recognition through a mobile GPU and a Viterbi accelerator. *IEEE Micro* 37, 1 (2017), 22–29.

[27] Steve J. Young, Gunnar Evermann, Mark J. F. Gales, Thomas Hain, Dan Kershaw, Xunying Liu, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, et al. 2002. *The HTK Book.* University of Cambridge, Department of Engineering.

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