BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

MODIFYING LARGE LANGUAGE MODEL DECODING USING VITERBI ALGORITHM

by

JIVESH JAIN

B.S., Boston University, 2025

Submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

2025

Approved by

First Reader

Bobak Nazer, PhD
Associate Professor of Electrical and Computer Engineering

Second Reader

Alan Pisano, PhD
Associate Professor of The Practice of Electrical and Computer Engineering

Acknowledgments

I would like to firstly acknowledge my advisor, Professor Bobak Nazer who suggested this thesis topic and helped me throughout my research journey with any questions that I had.

I also would like to acknowledge my parents for supporting me and motivating me throughout my academic and research journey.

MODIFYING LARGE LANGUAGE MODEL DECODING USING VITERBI ALGORITHM

JIVESH JAIN

ABSTRACT

Large Language Models (LLMs) rely on an encoding-decoding process to generate text. During encoding, an input sentence is transformed into a sequence of numerical representations, which the model processes to understand context. In decoding, the model generates output tokens sequentially, predicting the next word based on previously generated words. However, standard autoregressive decoding algorithms limit LLMs by using only past context, often leading to locally optimal but globally suboptimal predictions.

This project aims to propose another transformer decoding algorithm by integrating the Viterbi algorithm, a dynamic programming approach used in sequence optimization. Instead of selecting each token greedily, our method incorporates both past and future tokens to determine more probable sequences, treating text generation as an optimization problem. By refining token selection based on structured constraints, our approach aims to find more probable text, on average, than standard decoding algorithms such as greedy search. The final deliverable is a modular decoding framework that can be integrated into existing transformer architectures and be used for generating high probability text.

Contents

List of Tables

List of Figures

List of Abbreviations

V	.....	Vocabulary space of the Large Language Model
LLM	.....	Large Language Model

Chapter 1

Introduction

1.1 Transformer/Large Language Model Architecture

Transformers are a type of neural network that uses self-attention, a mechanism that determines the importance of different parts of sequential data by mapping each component to every other component in an n-dimensional space. This mechanism allows the model to learn context from data and capture dependencies and relationships between tokens, regardless of their position in the sequence (Wolfe, 2023). Transformers operate through a neural sequential transductive architecture that utilizes stacked encoder and decoder blocks (Vaswani et al., 2017). These blocks apply multi-head self-attention and feedforward operations in parallel, which enables the model to learn context-aware representations of sequential input data. The attention mechanism allows each token to selectively focus on other relevant tokens in the sequence, enabling informed predictions that reflect the structure and correlation between sequential data.

Large Language Models (LLMs) are a type of transformer architecture that specializes in large-scale natural language processing tasks such as text generation, summarization, and question answering. LLMs, such as the GPT (Generative Pretrained Transformer) family, rely on a decoder-only transformer for autoregressive text generation. In these models, the input text is first tokenized into a sequence of token IDs, which are then embedded into an n-dimensional vector space.

Specifically, let the input token ID vector be denoted by $\mathbf{x} = (x_1, \ldots, x_N)$, where

each $x_i \in \mathbb{R}$ represents a token ID, and there are a total of N tokens. Each token ID is then mapped to its initial vector representation $\mathbf{e} = (\mathbf{e}_1, \ldots, \mathbf{e}_N)$, where $\mathbf{e}_i \in \mathbb{R}^n$, via an embedding matrix $\mathbf{W} \in \mathbb{R}^{|V| \times n}$, with $|V|$ denoting the vocabulary size of the LLM.

This process is performed by representing each token ID x_i as a one-hot vector $\mathbf{o}_i \in \mathbb{R}^{|V|}$ and computing the embedding as $\mathbf{e}_i = \mathbf{W}^\top \mathbf{o}_i$. These embedded vectors are then passed into the decoder, which applies multi-headed masked self-attention followed by position-wise feedforward layers (Radford et al., 2019). As the token representations propagate through the decoder sublayers, they are iteratively transformed to encode both local and global contextual information.

After processing through the final decoder layer, the model produces a contextual representation for each input token, as well as an embedding for the next predicted token i.e. decoder modifies the input representation to the output representation in the following mapping: $R^{N \times n} \rightarrow R^{(N+1) \times n}$ where N is the length of the input token-ids and the additional vector corresponds to the predicted continuation.

The final vector is projected back into the vocabulary space using a linear transformation to produce a distribution over the next token. This is typically implemented using the transpose of the original embedding matrix $\mathbf{W}^\top$ to obtain the vector of scores or logits(unnormalized scores) over all the tokens in the vocabulary space for the next word prediction given the input. At the core of this decoder-only transformer model is the softmax layer, which converts these raw scores over the vocabulary space V into a probability distribution for the next predicted word using

$$\text{Softmax}(\text{logit}(k_i)) = \frac{e^{\text{logit}(k_i)}}{\sum_{z=1}^{|v|} e^{\text{logit}(k_z)}} \tag{1.1}$$

where $\text{logit}(k_i)$ denotes the unnormalized score of the i^{th} token in the vocabulary. This function, by normalizing, ensures that the probabilities are within $[0, 1]$.

1.2 Decoding Schemes

Once the distribution over the vocabulary for the next token is obtained, the LLM uses a decoding scheme to decide which token to generate as its output, given the input.

1.2.1 Greedy Decoding

Greedy decoding is one of the most computationally efficient decoding schemes, as well as one of the simplest to implement. The objective of Greedy decoding is to autoregressively predict the next token given the input and the output generated so far. Mathematically,

$$(y_{greedy})_t = \arg\max_{y_t \in V} P(y_t|x, y_{<t}) \tag{1.2}$$

where y_t is a token to be predicted at the t^{th} iteration, x is the input and $y_{<t}$ is the entire output string up till the $(t-1)^{th}$ step (Gu et al., 2017). The greedy decoder aims to find the token and output it, which gives the maximum probability conditioned on the input and the outputs up till that point. Therefore, the greedy search only traverses a single path during the decoding procedure, i.e., it keeps on producing tokens by considering just the previous tokens but not how the probability distribution might change as the tokens are outputted. Thus, greedy decoding could be viewed as a locally optimal solution to text generation. This is because, since it only goes forward along a single path in the directed paths of all possible token paths, it fails to account for information changes as the LLM builds up its response. Thus, the results obtained from the greedy search are globally suboptimal as it does not necessarily find the most probable text sequence to output as the iterations are increased.

1.2.2 Beam Search

Beam search works on a similar principle to greedy decoding. Like greedy decoding, it follows an autoregressive approach, predicting one token at a time based on previous outputs, but instead of committing to a single most probable token at each step, it maintains the top-k most probable sequences (called beams) at each time step (Freitag and Al-Onaizan, 2017). This allows it to explore multiple candidate paths through the output space in parallel.

In particular, at each time step t, the decoder expands every sequence in the current beam by one token selected from the top k tokens in a distribution over the next token given a path, computing the total probability for all possible next token paths. Then it selects the top-k sequences with the highest cumulative probabilities to carry forward. This process repeats until a stopping condition is met, such as generating an end-of-sequence token or reaching a maximum length defined by the user. At the end, the most probable sequence among the final beams is selected as the output. Algorithmically, this can be written as:

Algorithm 1 Beam Search Decoding

Require: Input sequence x, model $P(y_t \mid x, y_{<t})$, beam size k, max length T
Ensure: Most probable output sequence `beam_path`

Initialize `paths` with an empty sequence
for $t = 1$ to $T - 1$ **do**
  Initialize `new_paths` as empty list
  for m $= 1 \ldots k$ **do**
    `extensions` $\leftarrow$ TopKPaths$(P(y_t \mid x, y^m_{<t}))$
    Append `extensions` to `new_paths`
  end for
  `paths` $\leftarrow$ TopK(`new_paths`)
end for
`beam_path` $\leftarrow \arg\max_{y \in \texttt{paths}} P(y \mid x)$
return `beam_path`

Here, the function TopKPaths finds the top-k tokens given a path, and then

concatenates those tokens back to the given path to obtain k different paths, and the function TopK returns the top-k most probable paths so far in terms of overall probability of the sequence.

While beam search offers a better approximation than greedy decoding by considering multiple options at each step, it is still locally optimal—that is, it only considers future options within the limited scope of its beam width. Instead, it compromises by prioritizing computational efficiency over a broader search space, striking a balance between performance and resource constraints.

1.2.3 Sampling based Decoding

This is another class of decoding methods that, unlike previous methods, introduce randomness for text generation. These decoding schemes output the next token by sampling from the distribution obtained from the softmax layer. The two primary techniques are:

- Top-K Sampling: This method restricts sampling to the K most likely candidates at each step (Fan et al., 2018):

$$V_K = \text{Top-K}(P(y_t|x, y_{<t}))$$
$$y_t \sim U(V_K)$$

  Here, V_K represents the K tokens with the highest conditional probability, and $U(V_K)$ is the uniform distribution over the finite set V_K. Instead of sampling uniformly from V_K, the algorithm could also alternatively sample tokens proportionally to their original conditional probabilities within V_K. The parameter K controls the diversity-quality tradeoff: smaller values produce more focused outputs, while larger values increase diversity.

- Nucleus Sampling (Top-p): Instead of using a fixed k to sample from, nucleus

sampling selects from tokens whose cumulative conditional probability exceeds the threshold p (Holtzman et al., 2020). Mathematically,

$$V_p = \arg \min_{\substack{V' \subseteq V \\ \sum_{y' \in V'} P(y'|x, y_{<t}) \geq p}} |V'|$$

$$y_t \sim U(V_p)$$

Here $U(V_p)$ represents the uniform distribution over the set V_p. More tokens are considered, on average, when the threshold p is set higher and thus, there is a greater subset to sample the next token from.

These decoding methods, instead of picking the most probable sequence, determine the output sequence by sampling from the distribution obtained from the softmax layer.

1.3 Problem Statement

The decoding strategies, such as greedy and beam decoding, discussed above are employed in LLMs for text generation. While effective to varying degrees, these methods often aim to prefer computational efficiency over high probability text generation.

The goal of this work is to develop a decoding algorithm that aims to consistently identify and return more probable output sequences, given an initial context. To achieve this, we propose a novel integration of the Viterbi algorithm into the decoding pipeline of LLMs. Our approach aims to leverage the global optimality properties of Viterbi decoding in order to discover higher-likelihood output sequences compared to the aforementioned methods.

Chapter 2

Viterbi and Implementation

2.1 Viterbi Algorithm

The Viterbi algorithm, as introduced in (Viterbi, 1967), is a dynamic programming-based algorithm primarily used with Hidden Markov Models (HMMs) to find the most probable sequence of hidden states given a sequence of observations. It operates in two phases: a forward mode and a backward mode (Jurafsky and Martin, 2025). In the forward mode, a directed acyclic graph (DAG), also known as the trellis, serves as the primary data structure. This trellis stores intermediate results, similar to the memoization step in dynamic programming. The trellis is built before the forward mode is started. In forward mode specifically, at each time step, for each possible hidden state, the algorithm, with the help of the trellis, records:

- The maximum probability of reaching that state.
- The parent state that led to that maximum probability.

This construction is based on the law of total probability and leverages the Markov property, which is that the probability of transitioning to a state depends only on the immediately previous state. Mathematically, if we denote x as the initial state and state_i as the hidden state at iteration i, then:

$$P(\text{state}_t) = (P(x)) \times \Pi_{i=1}^{t} P(\text{state}_i | \{\text{states}_j\}_{j=1}^{i-1}, x) \tag{2.1}$$

$$\implies P(\text{state}_t) = (P(x)) \times \Pi_{i=1}^{t} P(\text{state}_i | \text{state}_{i-1}) \tag{2.2}$$

The last result follows directly from applying the Markov property. Thus, at each iteration t, for each possible state, the Viterbi algorithm computes the maximum probability of reaching that state from all previous states and then assigns as the "parent" the state from which that maximum was achieved. This assignment is based on Bellman's principle of optimality (Light, 2023), which is that an optimal path must be made up entirely of optimal subpaths. It cannot contain a suboptimal path in its intermediate stages. After completing the forward phase, at the final time step, we identify the state with the highest score. This marks the end of the most probable path. In the backward mode, we reconstruct the most probable path by starting at the state with the highest probability in the final time step. We then recursively backtrack using the parent pointers stored in the trellis back to the initial state. By assembling the sequence of states visited during this backtracking, we recover the most likely sequence of hidden states corresponding to the observed data.

In our text generation case, this Viterbi algorithm will essentially replace each state with tokens from the LLM's vocabulary space. The algorithm will aim to find the most probable string to output given some initial context string.

2.2 Viterbi Decoding

This section forms the core of our decoding algorithm, detailing its principles, implementation, and how it was integrated with large language models (LLMs) to develop our custom decoding scheme.

2.2.1 Implementation

Our decoding algorithm, like any other decoding method, interfaces with an LLM to generate output. It operates in three main phases:

1. Constructing a trellis to store candidate token information (token IDs, token

names as introduced in Section 1.1) at each iteration.

2. Using the backward mode of the Viterbi algorithm, we identify the most probable path through the trellis.

3. Producing the final output by assembling the tokens corresponding to the best path.

In short, any autoregressive LLM can be integrated with our decoding algorithm. We begin by using the LLM tokenizer to convert the input text into a sequence of input token IDs. These token IDs are then passed through the LLM, which applies attention mechanisms across its multiple layers to produce a probability distribution over the vocabulary for the next token. Our decoding algorithm then selects a sequence of output token IDs by iteratively using information from these distributions. Finally, these token IDs are mapped back into human-readable text again by using the LLM's tokenizer.

Trellis Building

The trellis is constructed by creating a directed acyclic complete graph (DAG) that encodes all necessary information about the output states. Let us denote this trellis by G. The vertices $V(G)$ represent the candidate tokens, while the directed edges between two vertices $u, v \in V(G)$ correspond to the transition probabilities from u to v.

The algorithm receives as input the desired output length, which determines the depth of the trellis. At each layer t of the trellis, we maintain the set of candidate tokens that could be selected at that time step. To populate these candidates in the trellis, we iteratively build input strings up to a current token y_t at layer t i.e. we find the input IDs by iteratively calling the parent of the current token and building a context input ID list, which we use to query the transformer model to obtain

a probability distribution over the next tokens. Repeating this procedure for each token at layer t generates the set of candidates for layer $t+1$.

Problems in Trellis Construction

During the trellis construction phase, we are not yet applying the forward mode of the Viterbi algorithm, as described in Section 2.1. As a result, parent assignments between tokens, which are crucial for determining optimal paths, have not yet been finalized. However, to generate the next layer of the trellis, we must supply the transformer with input context up to a given token, query it, and obtain a distribution over possible next tokens. This process necessitates the establishment of a provisional parent-child relationship during construction: each token generated at a given step t can be tentatively assigned as the child of a token in step $t-1$ that produced it first.

The structure appears to follow a trellis design similar to classical Viterbi implementations, but with a crucial modification. Unlike traditional Viterbi, our approach explicitly establishes parent-child relationships between nodes even during the trellis construction phase. This distinction is necessary because transformers require the full sequence of token IDs up to the current position, not just the immediately preceding token, as in Markovian models. This represents a fundamental departure from traditional Viterbi implementations. The classical algorithm relies on the Markov property, and as a result the trellis construction and the execution of Viterbi algorithm are greatly simplified, as the transition probabilities between states can be determined solely based on the current state without considering the full history. The transformers inherently violate this assumption by maintaining attention across the entire sequence history.

Transformer-based models are inherently *non-Markovian*. The probability of generating a token at time $t+1$ depends on the entire input sequence up to time t. That

is,

$$P(y_{t+1} \mid y_t, y_{t-1}, \ldots, y_1) \neq P(y_{t+1} \mid y_t, x)$$

where x is the input prompt. A slight modification to any token earlier in the sequence, even tokens far removed from y_t, can significantly alter the output distribution over the next token. This violation of the Markov property creates serious challenges for trellis construction.

Consequences for Trellis Construction Because of this dependency on full context, assigning parents without accounting for the entire prior path can lead to a mismatch between the transition probabilities recorded during trellis construction and those used during decoding. If the context used to generate a token differs from the context assumed when selecting a parent later, then the transition probability between two nodes no longer correctly reflects the real model behavior.

In particular, if the parent of a token changes during the final Viterbi decoding pass (because the decoding finds a more probable path through a different prior context), the associated transition probability recorded during trellis construction becomes invalid. This misalignment can propagate errors forward, ultimately causing the decoded sequence to deviate from the true most probable sequence according to the model.

Thus, without careful handling, the traditional separation between trellis construction and decoding, which works under the Markov assumption, breaks down for transformers. A naive Viterbi application would risk severely underestimating or overestimating path probabilities, resulting in suboptimal decoding outcomes. The fundamental non-Markovian nature of transformers means that the construction of the trellis cannot simply replicate the methods developed for classical Markovian scenarios. Instead, trellis construction and parent assignment must be tightly coupled

with respect to full input contexts, taking into account the entire token history at each step. This motivates the **Viterbi-on-the-go** strategy described in the next section, which ensures that parent assignment and probability estimation remain consistent during the construction of the trellis itself.

Viterbi-On-The-Go

To correctly represent transition probabilities as edge weights in the trellis, it is essential to take into account not only the current token but also the specific path that led to it. In particular, the context up to a given state directly influences the transformer's output distribution, making it critical that the parent of each token reflects the most probable path according to the model's scoring.

A natural solution to this issue is to integrate the forward step of the Viterbi algorithm into the trellis construction process itself. Rather than building the entire trellis first and then assigning parents retrospectively, we assign parents as we build each layer. Specifically, at each layer t, for any token y_t, we select its parent based on the token at layer $t-1$ that maximizes the conditional probability:

$$\text{parent}(y_t) = \arg\max_{y_{t-1}} P(y_t \mid x, y_1, \ldots, y_{t-1}).$$

This strategy, which we refer to as the Viterbi-On-The-Go routine, ensures that the principle of optimality is enforced during trellis construction: at each step, we follow the path that yields the highest probability of reaching the current token.

Practically, to populate the next layer of the trellis, we generate new candidate tokens by querying the transformer with the current token context, along with the full input sequence that led to the current token—that is, the string with the highest probability of reaching this point in the trellis. In doing so, the transition probabilities recorded between states correctly correspond to the actual transformer outputs under

the intended contexts, thus making sure that the transition probabilities used for finding the best path (the most probable path in the trellis) would not have an incorrect transition probability anywhere along the path.

If, instead, we were to assign each node its parent based on the first token that produced it, rather than the one leading to the highest probability, then the input context associated and transition probabilities with later tokens would differ from the ones used during trellis construction. This inconsistency between the trellis edge weights and the actual model behavior would compromise the integrity of the structure and could result in suboptimal path selections during the final Viterbi decoding.

Therefore, by incorporating a backward step at each trellis layer to select the maximizing parent before generating new candidate tokens, we maintain coherence between parent assignment and transition probability estimation. This alignment ensures that the forward decoding pass operates over a trellis whose structure accurately reflects the model's behavior, preserving both correctness and optimality. The visualization below can help in showing this:

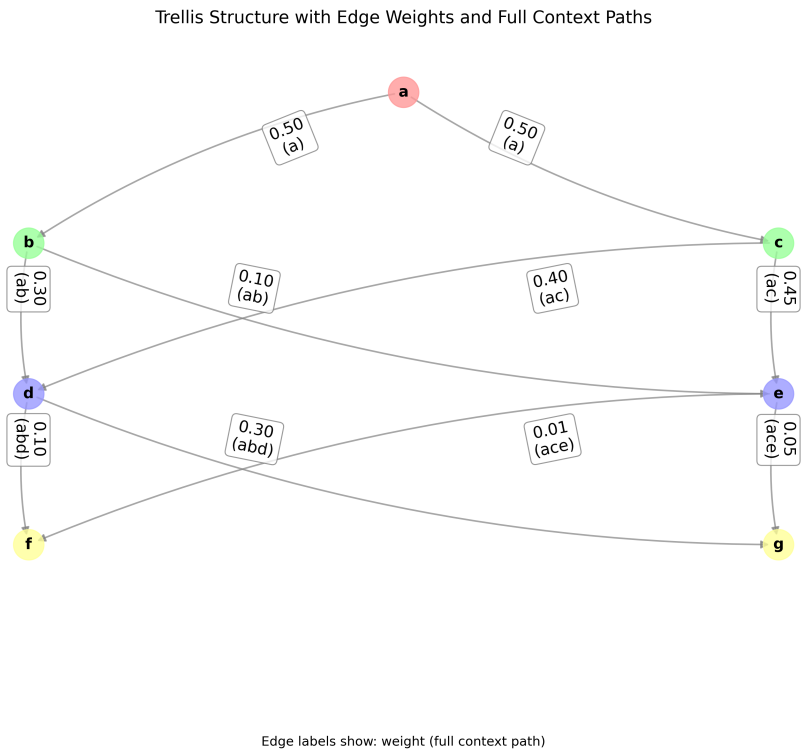

Figure 2·1: A trellis consisting of multiple states with context/custom token ID labeled on the nodes

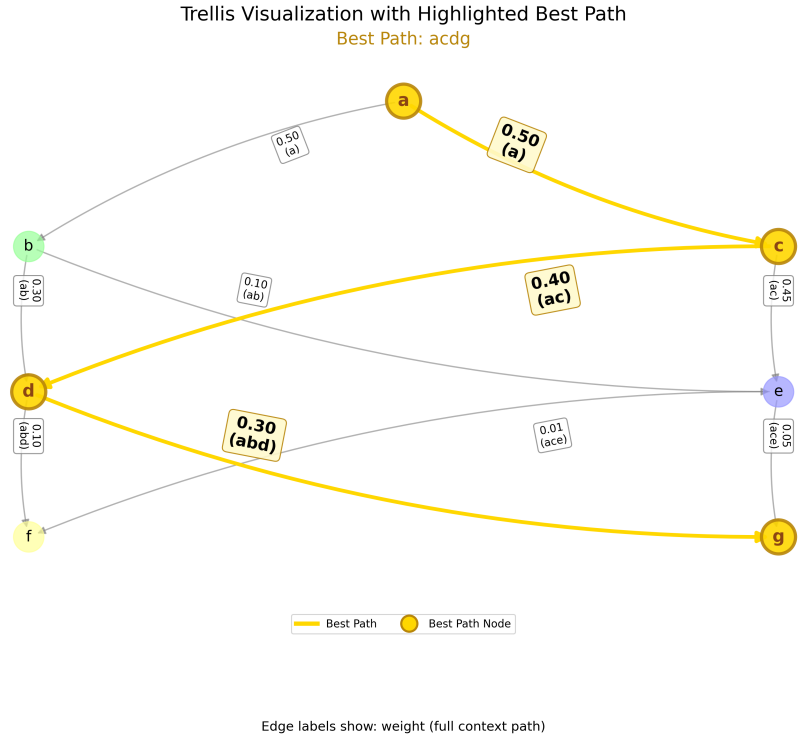

Figure 2·2: Showing the best path that would be returned by classical trellis generation procedure

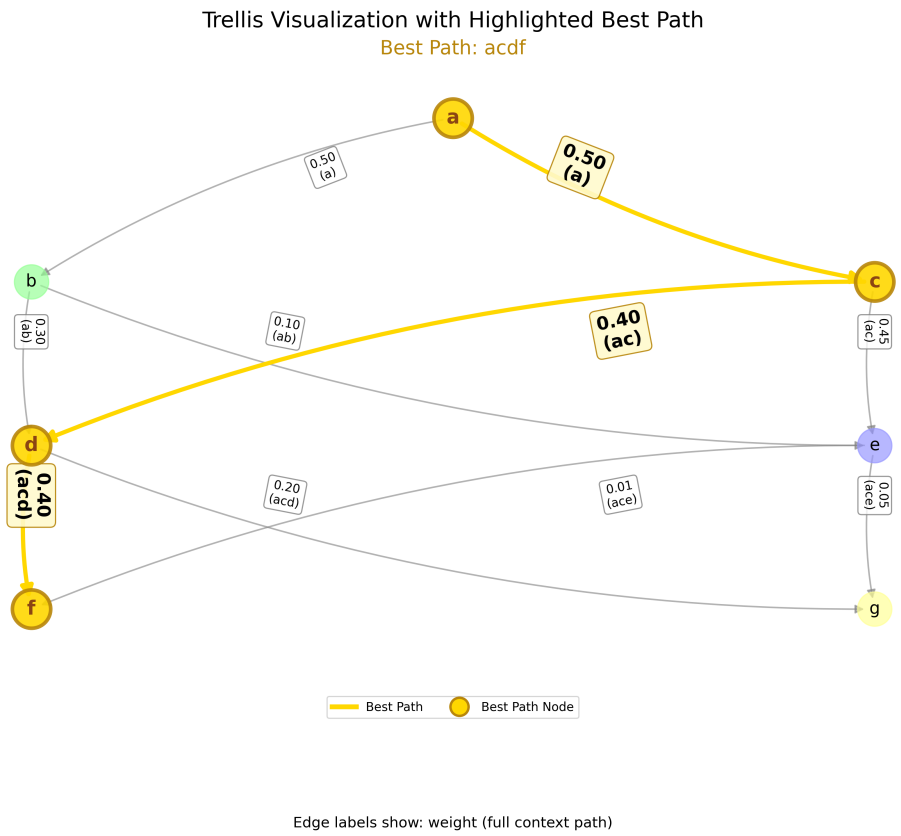

Figure 2·3: Showing the best path that would be returned by our Viterbi-On-The-Go trellis generation procedure

The figure 2·1 illustrates a trellis consisting of two states in each layer. The transition probabilities are annotated on the edges, along with the context strings used to query the transformer once a particular token is reached, to generate tokens for the next layer. This figure adopts a parent assignment scheme where the first token that generates a token in the subsequent layer is designated as the parent of that token, as would be done in the traditional Viterbi trellis generation.

In Figure 2·2, we demonstrate the best path computation using the traditional trellis generation process. At layer 3, which contains tokens (d, e), token b generated token d first and was thus assigned as its parent, while token c was assigned as the parent of token e. The token with the highest total score at the final iteration is g,

making it the endpoint for the best path. During backtracking, we follow the path to node d, which provided the maximum transition probability leading to g.

However, an important issue arises: the transition probability $P(d|c,a) > P(d|b,a)$. Ideally, this means that during forward processing, token c should have been assigned as the parent of token d. Consequently, during the backward step of Viterbi decoding, node c would correctly appear in the best path. The problem is that, during trellis construction, the transition probability from d to g was obtained by querying the transformer with the context string abd, not acd. In the corrected best path, the context leading to d should have been ac, and the appropriate transition probability for reaching g would require querying with acd, not abd. Thus, an inconsistency occurs: the forward context used to calculate transition probabilities does not match the context assumed during backtracking.

To address this, the Viterbi-On-The-Go strategy performs forward mode computations simultaneously while building the trellis. As a result, the context strings used to generate transition probabilities correctly match the contexts during the backward mode, ensuring that the returned string reflects the true probability conditioned on the initial context.

Finally, figure 2·3 illustrates that when we reach token d, it is correctly assigned token c as its parent. Consequently, the next layer of tokens is generated using the updated context associated with token d. In this example, due to the new context, $P(f|d,c,a) > P(g|d,c,a)$, meaning token f is now selected as the best token in that layer. Therefore, the final best path corresponds to the string $acdf$.

Summary of Phase 2 and Phase 3 Procedures

Once the trellis generation is complete, the final step is to backtrack from the most probable token in the last layer to the first layer by following the parent pointers. After identifying the best path, we extract the token IDs associated with the tokens

along this path and pass them through the LLM encoder to decode the token IDs back into text. The resulting text is then output as the final prediction of the LLM, conditioned on the initial context string. This concludes Phase 3 of our algorithm, as mentioned in Section 2.2.1. The algorithm terminates once the number of words specified by the user has been generated.

2.2.2 Optimizations

To make our algorithm more computationally feasible and scalable, we incorporate several key optimizations:

- Pruning with a Top-c Strategy: From the algorithm description, it is evident that trellis growth can scale exponentially (in the vocabulary size) with the number of tokens generated at each layer. A simple solution to control this rapid expansion is to define a pruning constant c. At each iteration, instead of expanding every possible token, we restrict the expansion of each token in the current layer to only its Top-c most probable next tokens. Thus, for every token in a given layer, we generate only its Top-c successor tokens for the next layer, significantly reducing the branching factor and therefore the breadth of the trellis. The pruning constant c is specified by the user as an input to the algorithm, and by default, is set to 3.

- Batch Processing and Parallelism: Since querying the transformer model constitutes the most computationally expensive operation in the algorithm, we implement a batching strategy to improve efficiency. Instead of querying the model separately for each individual context string, we batch multiple context strings together and query the transformer in parallel. This approach enables the transformer to simultaneously generate the Top-c probable next tokens for all tokens at a given layer, substantially reducing the overall number of model

calls and increasing the inference speed of the algorithm.

- Caching of Transition Probabilities during Viterbi-on-the-Go: As part of the Viterbi-on-the-go trellis construction strategy, we compute transition probabilities between all tokens in layer $t-1$ and the newly generated tokens in layer t. To improve computational efficiency during the backward step of parent assignment, we use caching to store each token in layer $t-1$ to store the transition probabilities to all tokens in layer t. This caching enables constant-time ($O(1)$) lookup when determining the transition probability from a token in layer $t-1$ to a token in layer t. Although this approach introduces additional memory usage, it remains tractable, as the caches are localized within function calls and are created only once for each token in layer $t-1$ during the trellis construction.

- Unique Token Storage per Layer: Although the branching factor c (as defined previously) helps control trellis growth, the number of nodes can still grow exponentially with depth, albeit at a slower rate compared to full vocabulary expansion if ($c \ll |V|$, where V is the vocabulary size of the LLM). Empirical observations during testing revealed that as the trellis depth increases, there is substantial overlap among tokens generated across different paths. To exploit this redundancy, we enforce unique token storage at each layer: each token is stored only once per layer, regardless of how many parent paths generate it. This strategy significantly reduces memory usage and minimizes the number of transition probabilities that must be computed and cached. By maintaining a more compact trellis, we improve traversal efficiency by reducing both the number of paths to explore and the backtracking operations required during trellis construction and decoding.

- Top-j Filtering during Forward Mode (Vocabulary Cutoff): Another critical

optimization addresses the computational cost of handling the transformer's full vocabulary, which typically includes over $50,000$ tokens. During transition probability calculation, when determining the probability of transitioning from a token in layer $t-1$ to a token in layer t, we introduce a cutoff threshold j (e.g., $j = 500$). Specifically, during next-token prediction, any token not ranked among the top-j tokens in the transformer's output distribution is assigned a transition probability of zero. This strategy drastically reduces the number of transitions that must be considered at each step, lightens the output processing load, and significantly improves runtime. Importantly, this optimization has minimal impact on performance, as low-probability tokens are unlikely to contribute to the best path through the trellis, as will be further discussed in Section 3.2.

Collectively, these optimizations enable the Viterbi-On-The-Go trellis generation procedure to operate efficiently, ensuring that both computation and memory usage don't scale very quickly as the number of tokens generated by the decoding scheme increases.

The pseudocode for the final algorithm could be found below:

Algorithm 2 Viterbi-On-The-Go Decoding Algorithm

Input: Target number of tokens T, pruning constant c, input string x, and Transformer model.
Output: Decoded output string and associated path probability.
Phase 1: Trellis Construction
Initialize the Transformer with input string x and obtain the probability distribution $P_X(x)$ over the vocabulary $\mathbf{V} = \{x_1, \ldots, x_K\}$.
Select the top-c candidate tokens according to $P_X(x)$ and store them in an array *Unique*, along with their transition probabilities.
Trellis.states[1] $\leftarrow$ *Unique*
Trellis.probs[1] $\leftarrow$ initial transition probabilities
Set $s = c$.
for $t = 2$ to T **do**
    Construct batch inputs: `batch_inputs` = [*Unique*[m].build_context for $m = 1, \ldots, s$]
    Query the Transformer with `batch_inputs` to obtain `batch_predictions`.
    Let l = length(`batch_predictions`).
    Update *Unique* = $\bigcup_{i=1}^{l}$ `batch_predictions`[i].
    for $m = 1$ to s **do**
        For each token in *Unique*, compute the transition probability from Trellis.states[$t-1$][m].
    end for
    for each token in *Unique* **do**
        Assign parent(token) = $\arg\max_{m_{t-1} \in \text{Trellis.states}[t-1]} P\left(\text{token} \mid x, \bigcap_{d=1}^{t-2} m_d, m_{t-1}\right)$
    end for
    Update s = length(*Unique*).
    Trellis.states[t] $\leftarrow$ *Unique*
    Trellis.probs[t] $\leftarrow$ corresponding transition probabilities
end for
Phase 2: Viterbi Backtracking
Identify the most probable state in the final layer Trellis.states[T].
Backtrack along the parent pointers to reconstruct the best path through the trellis.
return best path nodes, best path probability
Phase 3: Output Decoding
Decode the sequence of best path nodes into a string using the decoding function.

return Decoded string and best path probability.

2.3 Comparison with other Decoding Methods

Our Viterbi decoding algorithm differs fundamentally from other search strategies, such as the greedy method and beam search, because the structure of traversal paths is different for each method. The greedy search follows a single most likely path at each step and produces output tokens sequentially without any exploration of alternatives. Beam search explores multiple paths simultaneously, maintaining a fixed number k of active candidates at each step (known as the beam width). Our pruning constant c plays a role analogous to the beam width k, in that it controls the number of paths expanded at each layer of the trellis. However, our algorithm departs from beam search in that beam search expands each path independently based solely on the tokens produced along that path, i.e., no crosslinks are allowed between different paths.

In contrast, our modified Viterbi decoding allows crosslinks between paths. This means that when deciding the next token, we consider the possibility of transitions from any token in the previous layer to any token in the current layer, not just descendants of a specific beam. This richer connectivity structure intuitively allows us to explore a broader set of paths within the same branching factor c, and thus our algorithm may discover different and potentially better decoding sequences than beam search.

For instance, let i denote the trellis layer (or time step) and j, the index of the paths in the trellis. Then, $y_{i,j}$ represents the token at layer i along path j. We define a path at step s as

$$\text{path}(j) = \{y_{1,j}, y_{2,j}, \ldots, y_{s,j}\},$$

where s is the current depth of the trellis. In the next iteration $s+1$, during the parent assignment step (i.e., identifying the best predecessor for a token in layer $s+1$), our

algorithm considers all transitions of the form:

$$y_{s,j} \to z_{s+1,k}, \quad \forall j \in \{1, \ldots, u\}, \quad \forall z_{s+1,k} \in \text{Unique candidates at iteration } s+1,$$

where u is the number of tokens in the trellis at step s, and k ranges over the number of unique candidate tokens selected at step $s+1$.

Each $z_{s+1,k}$ then selects its best predecessor $y_{s,j}$ by maximizing the transition probability:

$$\text{parent}(z_{s+1,k}) = \arg\max_{y_{s,j}} P(z_{s+1,k} \mid x, \text{history ending at } y_{s,j}).$$

This allows crosslinks between different paths at each step. Thus, it is possible to have $\text{path}(i) : y_{1,i} \ldots y_{s,i} y_{s+1,i}$ where $y_{s+1,i} = z_{s+1,d}$ where d is different from i, that is the token $z_{s+1,d}$ was produced as the Top-c token from the path d and stored initially as the unique token produced from path d. Thus, these crosslinks allow links between tokens on different paths to potentially find output strings better than what could be found without considering transitions from other paths (i.e., Beam Search).

Sampling-based methods differ fundamentally from our algorithm in that they are only focused on generating text by probabilistically sampling from the output distribution at each step, without guaranteeing that the resulting sequence has high overall probability.

Although we are attempting to discover more paths within the same beam width c, it comes at the cost of additional computation, as is also shown below:

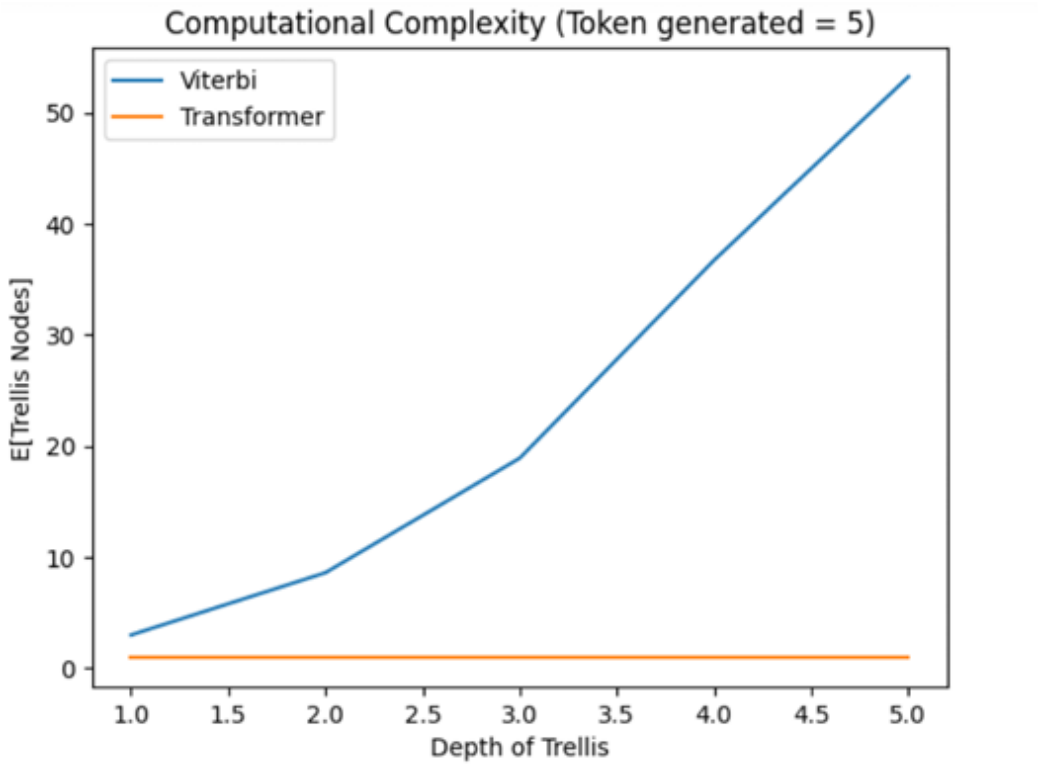

Figure 2·4: Computational Complexity of Our Viterbi Decoding with greedy search (represented as Transformer)

This graph presents the expected number of unique trellis nodes (i.e., the effective width of the search space) as a function of the trellis layer number, averaged over approximately 30 examples from the TruthfulQA dataset. All examples share a fixed output length of 5 tokens, while varying in input prompt length. The use of a uniqueness constraint on tokens at each layer effectively curbs exponential growth in the number of nodes, yet the overall computational cost of building the trellis still increases with depth. In contrast, the greedy decoding algorithm operates with constant-time complexity per output token, as it avoids trellis construction altogether. While this makes greedy decoding substantially faster, it limits the algorithm to a single path and consequently reduces its capacity to explore alternative candidate sequences.

Chapter 3

Results

Now, that we have seen that the major difference between our algorithm and the others lies in the number of paths it is considering at each stage, we can use a metric called perplexity to actually find out whether our algorithm is able to find more probable text sequences than greedy search and beam search as well as the other decoding methods.

3.1 Perplexity

Perplexity is a metric that is used in language generation for quantifying how uncertain the model is during generating the next word in the sequence string. Mathematically, perplexity denoted by PPl below can be expressed as:

$$PPl(X) = 2^{H(X)} \implies PPl(X) = 2^{-\sum_{x \in V} p_X(x) \log_2(p_X(x))}$$

The entropy of a random variable X is given by $H(X) = -\sum_{x \in V} p_X(x) \log_2 p_X(x)$. The entropy of X is the logarithm of the perplexity of X. The perplexity can also be calculated using base e or any other base other than base 2. Now, the perplexity of the string can be thought of as inversely proportional to the probability of the string. Less perplexity implies less uncertainty in predicting the next word and hence, higher probability of the overall text sequence, whereas higher perplexity corresponds to lower probability of the output sequence. Thus, perplexity can help serve as a metric to judge the output string from our decoder with the output strings from

other decoders mentioned in section 1.2.

3.2 Perplexity of different decoders

This section details some of the experiments we performed to compare the strings returned by our decoder as compared to the strings returned by other decoders in section 1.2. We used perplexity as a metric to compare the different strings, as perplexity information will let us know more about the total probability of the computed string given the user's input. The experiments were performed using the wikitext dataset. The graphs 3·1 and 3·2 show the results.

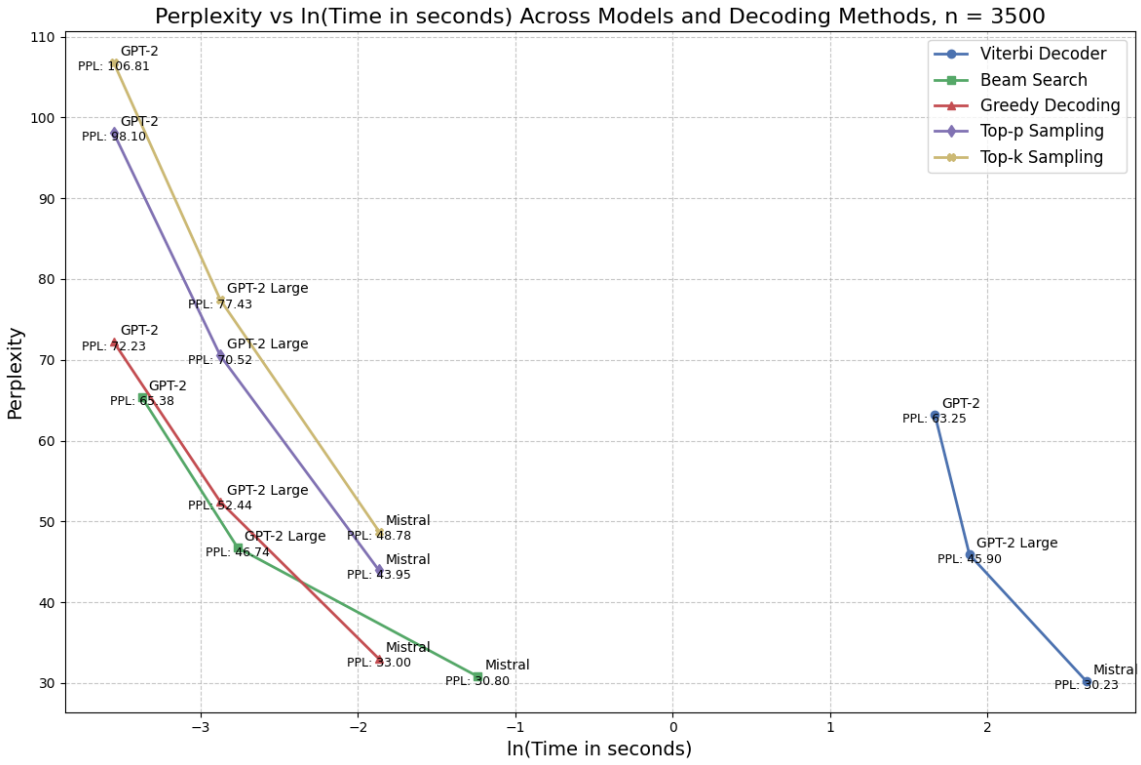

Figure 3·1: Showing the Perplexity vs ln(Time in seconds) for different decoders for a generation of 7 tokens and 3500 examples

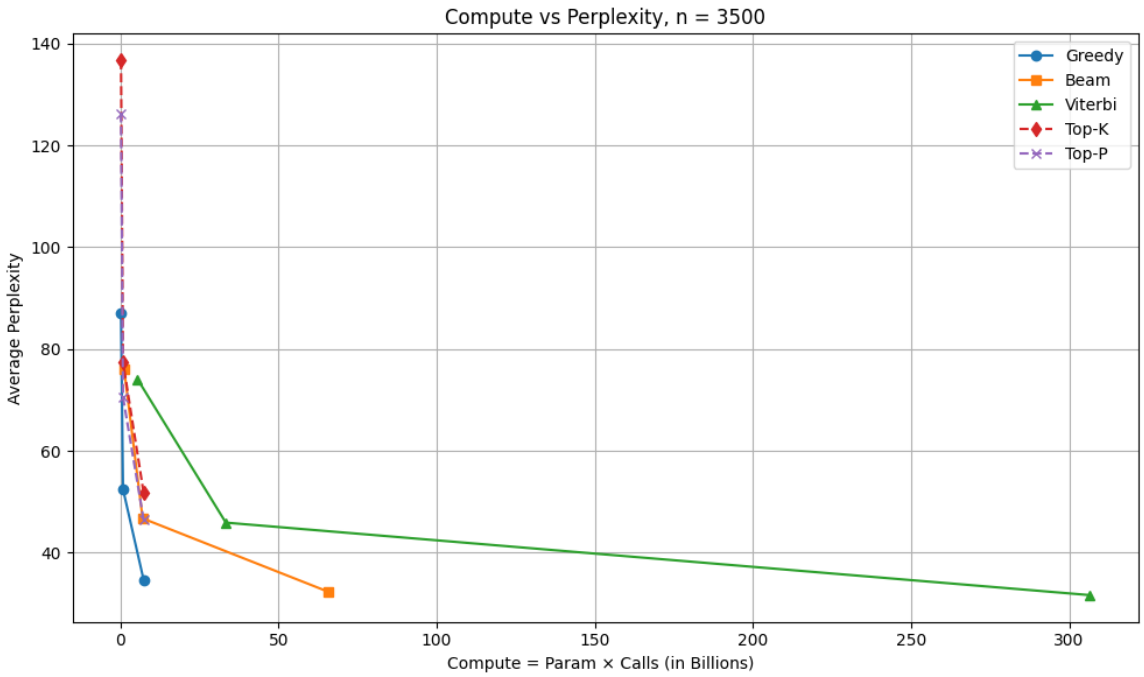

Figure 3·2: Showing the Perplexity vs compute for different decoders for a generation of 7 tokens and 3500 examples

We conducted the experiments on hardware consisting of two GPUs with compute capability 8.6 paired with a dual-core CPU. Figure 3·1 presents a comparative analysis of perplexity versus ln(Time in seconds) across different decoders. To comprehensively evaluate our algorithm's scalability and effectiveness, we tested it against three models of increasing complexity: GPT-2 ($124M$ parameters), GPT-2 Large($774M$ parameters), and Mistral ($7B$ parameters).

The results depicted in Figure 3·1 demonstrate that our decoder can achieve lower average perplexity scores, translating to higher probability strings, across all tested models. This experiment utilized $3,500$ examples from the dataset, with each decoder generating 7 tokens per example. The graph plots the average perplexity for each decoder against the time required to produce outputs.

While our decoder achieved superior string probabilities, it did require more com-

putational time compared to alternative decoders. Despite this increased time, notably, even when our algorithm and the beam search were configured with identical beam widths, our approach consistently produced different and lower perplexity strings, as highlighted in the table 3.1 below. This finding suggests that the current implementation of beam search causes it to fail in optimally exploring its search space, thereby overlooking potentially higher-probability sequences that our algorithm successfully identifies. This indicates that beam search does not fully explore the space within its fixed beam width, potentially missing higher-probability strings, which leads to higher perplexity in its results. Our decoder, by contrast, can produce lower perplexity outputs and demonstrates that beam search can be suboptimal, even when restricted to its fixed beam width.

Our decoder not only outperforms beam search but also consistently surpasses sampling-based techniques. The probabilistic sampling methods, including Top-K and nucleus sampling (Top-P), consistently produced higher perplexity strings than our algorithm. This demonstrates that while these sampling techniques introduce diversity, they often sacrifice optimality by selecting sub-optimal tokens (in terms of lower overall probability of the output sequence) during the generation process.

Furthermore, our decoder decisively outperforms greedy search results, confirming that the locally optimal choices made by greedy decoding frequently lead to globally suboptimal sequences. Figure 3·2 examines how the perplexity of the output string scales with the size of the model while maintaining a consistent output token length between different decoders. We quantify compute as compute = number of parameters $\times$ number of transformer calls by the decoding method. This analysis reveals that for smaller models like GPT-2, our decoding algorithm maintains competitive efficiency in terms of transformer queries required, as our compute is closer to beam search's compute when we used GPT-2 than when we used all the other

remaining models. However, as the complexity of the model increases, our algorithm trades off computational complexity to find a higher-probability string output.

Table 3.1: Comparison of Our Decoder's Performance Against Beam Search and Greedy Decoding

tokens produced = 7, number of examples = 3500

Model	**vs. Beam Search**		**vs. Greedy Search**	
	Lower Perplexity	Lower or Equal Perplexity	Lower Perplexity	Lower or Equal Perplexity
GPT-2 (124M)	1649	3335	2907	3441
GPT-2 Large (774M)	1207	3291	2716	3443
Mistral 7B v0.1	1201	3272	2489	3467

Table 3.1 displays the frequency with which our algorithm generates output strings that have a lower perplexity than beam search and greedy search. It also shows the total number of instances where our algorithm produces strings that are either equal to or lower than the perplexity of beam search or greedy search.

In Section 2.2.2, we discussed using Top-j filtering—restricting each decoding step to the top j most probable tokens (e.g., $j = 500$)—to reduce computational cost. Figures 3·3, 3·4, and 3·5 support this strategy by showing that, across 3500 examples (for at least token generation processes that are relatively smaller, such as 7 for us), the average position of the best path remains near the beginning of the token range, even as the number of generated tokens increases. While some later tokens may have high conditional probabilities, the highest-probability paths overall tend to involve tokens from the earlier ranks. As decoding progresses and more nodes are added to the trellis, lower-probability paths struggle to overcome the accumulated advantage of earlier, high-probability sequences. Therefore, setting the Top-j cutoff to a smaller value significantly improves computational efficiency without substantially compromising performance.

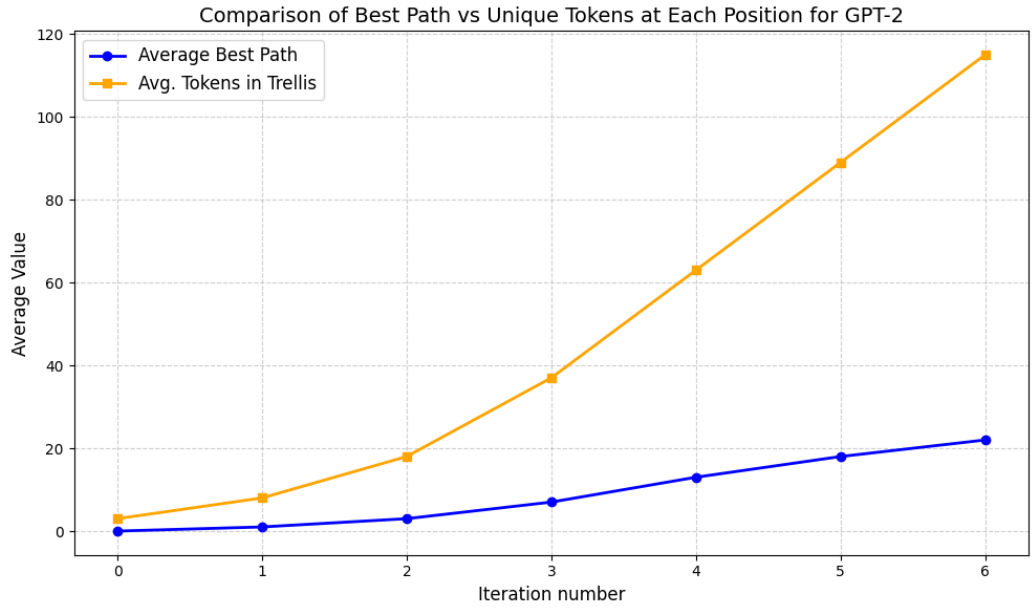

Figure 3·3: Best_path returned by Viterbi and average number of nodes in the trellis at a given iteration for GPT-2

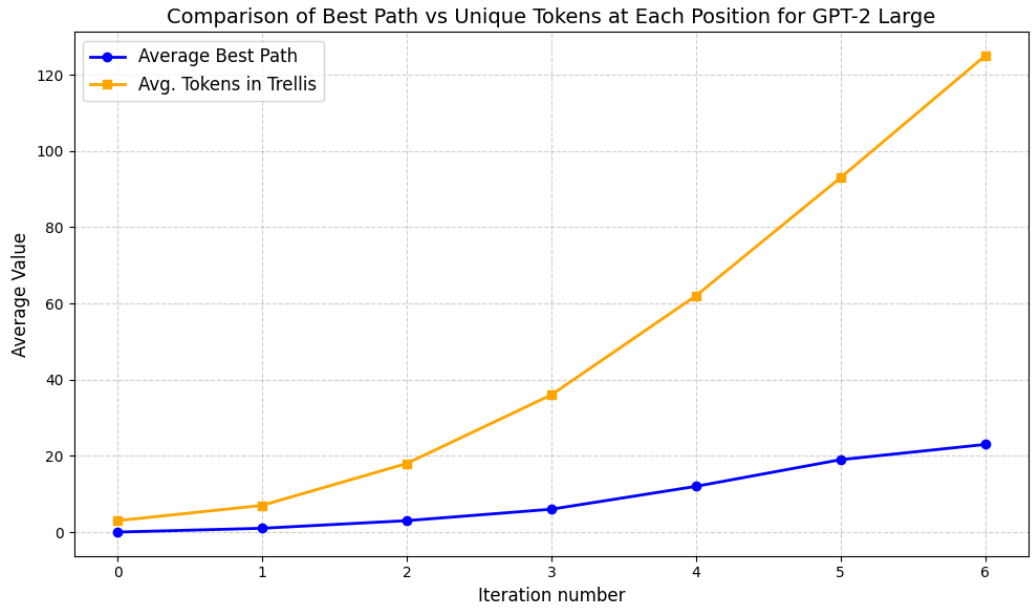

Figure 3·4: Best_path returned by Viterbi and average number of nodes in the trellis at a given iteration for GPT-2 Large

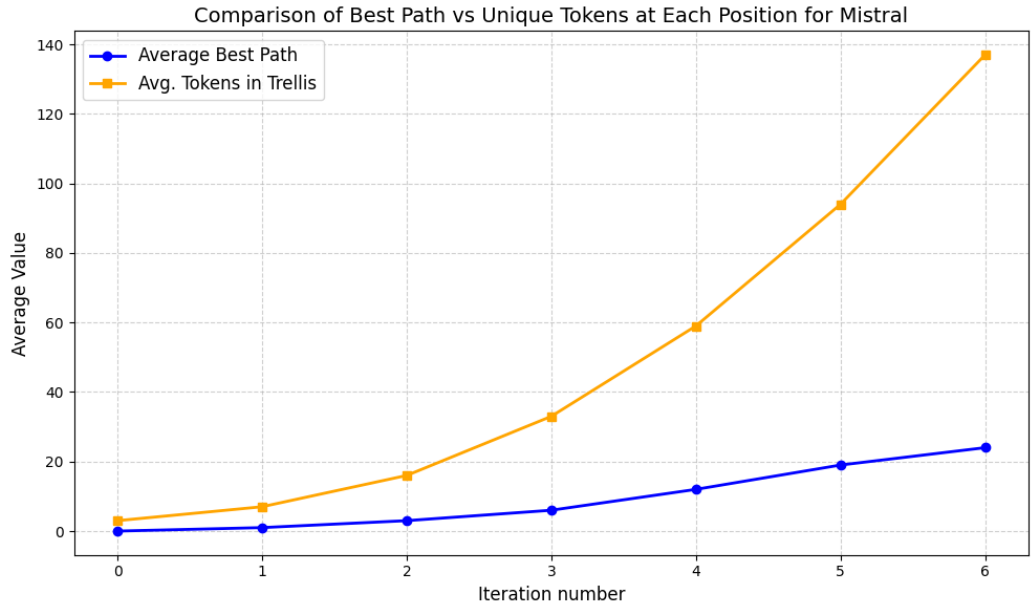

Figure 3·5: Best_path returned by Viterbi and average number of nodes in the trellis at a given iteration

3.2.1 Algorithm Optimality and Behavior Relative to Greedy and Beam Search

An important property of our algorithm is that it always returns the most probable path contained within the trellis. If the greedy decoding path exists in the trellis and is the most probable, our algorithm will return it. Likewise, if the beam search path exists and has the highest total probability among all paths in the trellis, the algorithm will return the beam path. However, the key distinction lies in the algorithm's ability to find and output a different path if a more probable sequence exists—one that neither greedy nor beam search may discover.

Empirical evidence from Table 3.1 supports this behavior. In several instances, our decoder returns the same string as greedy decoding, implying that the greedy path was preserved in the trellis and was also the most probable. A similar observation applies to beam search. As model complexity increases, the number of times our

decoder returns a different output from greedy decoding and beam search decreases. However, the number of times it returns a string with perplexity less than or equal to greedy decoding or beam search remains roughly the same.

This trend seems to suggest that as model capacity increases (e.g., with more parameters), the model becomes more confident in its predictions along dominant decoding paths such as greedy or beam search. As a result, it becomes increasingly difficult to identify alternative paths in the trellis that are more probable than these standard decoding trajectories. As a result, the greedy path or beam path(s) are more likely to remain fully intact in the trellis. Thus, the string output by our algorithm, for at least small token generation (e.g. 7 for us), was either the greedy or beam string or a better one in terms of overall probability.

When the Greedy Path Is Excluded from the Trellis

The greedy path may be excluded from the trellis if, during any iteration, the greedy token at step $t+1$ is assigned a parent that is not the greedy token from time step t. This results in a deviation from the greedy context, and the continuation of the greedy path in the trellis gets terminated.

We can formalize this exclusion condition as follows. Let $s = y_1^s, \ldots, y_k^s$ be an alternative path in the trellis, and let $g = y_1^g, \ldots, y_k^g$ be the greedy path. Define the *deficit* as the cumulative difference in log-probabilities between the greedy path and the alternative path s at the current iteration t:

$$\text{Deficit} = \sum_{k=1}^{t} \log P(y_k^g \mid x, y_{<k}^g) - \log P(y_k^s \mid x, y_{<k}^s)$$

At iteration $t+1$, if the *deficit* is less than 0, i.e., the path s is more probable than path g at iteration t, then the greedy path will not remain in the trellis for future

iterations if

$$P(y^g_{t+1} \mid x, \bigcap_{i=1}^{t} y^g_i) + \text{Deficit} < P(y^g_{t+1} \mid x, \bigcap_{i=1}^{t} y^s_i)$$

Now, if the deficit is greater than 0, again we will reach the same condition as above but in this case the transition probability to the next greedy token from path s should be higher than the transition probability from greedy path by an amount that exceeds this deficit so as not to select the greedy path. This condition captures the scenario where the Viterbi-on-the-go decoding strategy assigns the parent of the greedy token to a node from a non-greedy context due to a higher transition probability. Subsequent tokens are decoded from this new context, effectively removing the greedy path from the trellis.

This analysis can be extended to beam search as well. In that case, the exclusion of the beam path would require that all k top paths are replaced by alternative parents at some point in the decoding process.

Finally, it is worth noting that while excluding the greedy (or beam) path could theoretically lead to sequences with higher perplexity, our empirical observations show that this is rare for short sequence generation tasks. In most cases, our decoder still produces strings that match or outperform greedy (or beam) decoding in terms of probability.

Chapter 4

Conclusions

4.1 Summary of the thesis

This work introduces the integration of Viterbi decoding as a viable decoding strategy for autoregressive transformer models. To this end, we developed a fully functional algorithm that can be seamlessly incorporated with large language models (LLMs) to generate textual outputs. Empirical evaluations demonstrate that our approach frequently produces outputs with lower or equal perplexity compared to traditional greedy and beam decoding strategies. However, these improvements in perplexity, within the same search space, come at the cost of increased computational overhead. A promising direction for future work is trellis sparsification—that is, pruning low-probability token branches beyond a certain threshold—to enhance scalability and improve the algorithm's computational efficiency.

References

Fan, A., Lewis, M., and Dauphin, Y. (2018). Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 889–898. Association for Computational Linguistics.

Freitag, M. and Al-Onaizan, Y. (2017). Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, Vancouver, Canada. Association for Computational Linguistics.

Gu, J., Bradbury, J., Xiong, C., Li, V. O., and Socher, R. (2017). Trainable greedy decoding for neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 196–206. Association for Computational Linguistics.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. (2020). The curious case of neural text degeneration. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*.

Jurafsky, D. and Martin, J. H. (2025). Hidden markov models. In *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Draft, Online Manuscript, 3rd edition. Draft of January 12, 2025.

Light, B. (2023). The principle of optimality in dynamic programming: A pedagogical note. *arXiv preprint arXiv:2302.08467*.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. Technical report.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 6000–6010.

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

Wolfe, C. R. (2023). Decoder-only transformers: The workhorse of generative llms. https://cameronrwolfe.substack.com/p/decoder-only-transformers-the-workhorse. Accessed: 2025-05-02.