LLMs for Language Learning: Difficulty Constrained Decoding

Arthur Chansel

Data Science Master's Student
Research Project Report

September 2024 to January 2025

Supervisors
Lars Henning Klein, Valentin Hartmann
EPFL dlab

Table of Contents

1. Abstract

Regardless of their outstanding performances, Large Language Models (LLMs) still struggle with producing reliable complex output structures. While common training-focused methods for specializing LLMs are often data-heavy and computationally expensive, other architectural approaches can address all these issues.

This project introduces a method to design LLMs for language learning by constraining their output to a predefined vocabulary set, such as B1 proficiency words, and achieving backtracking for input dependence. The decoding function is modified using dynamic masking to ensure that the model generates only permissible words. A word graph is pre-constructed offline from the vocabulary dataset with a trie-based structure and employed as a finite-state machine to provide valid tokens for masking (i.e., state transitions). Additionally, the beam search method is tested to improve the output quality.

This approach explores new avenues for specifying LLMs, and provides a scalable and efficient solution for restricting the output, offering potential for integrating them into language learning tools, enhancing personalized education.

2. Introduction

Large Language Models (LLMs) are increasingly ubiquitous, finding applications across a wide range of tasks. However, it is still a challenge for them to consistently produce reliable complex output structures[1]. Furthermore, common training-focused methods for specializing LLMs are often data-heavy and computationally expensive[2], lacking rigorous control over the model's behavior. Training approaches adjust the model's parameters to minimize a target function, but do not fully constrain its output from an architectural perspective. To address these limitations, alternative methods[3] focus on explicitly modifying the internal structure of the model, offering greater control over LLM behavior.

This project aims to leverage LLMs as tools for language learning. Specifically, we intend to restrict the model's outputs to a predefined whitelist, such as A1 proficiency level, enabling a conversation in a precise and targeted vocabulary. The primary objective of this project is to operate on the internal structure of the model by modifying its decoding function. It includes the design of appropriate tools and algorithms that can achieve backtracking to dynamically restrict the output space according to a predefined vocabulary. It excludes traditional training-focused approaches and advanced syntax constraints for modelling specific sentence structures. We seek to offer a scalable and efficient solution for integrating LLMs into language learning tools, potentially enhancing personalized language education.

To achieve this, we drew inspiration from the paper A General-Purpose Algorithm for Constrained Sequential Inference (Deutsch et al., 2019). We constrain the output by dynamically masking the output space, based on the input. The dynamic mask corresponds

[1] https://www.instill.tech/blog/llm-structured-outputs
[2] https://www.ml6.eu/blogpost/fine-tuning-large-language-models
[3] https://towardsdatascience.com/different-ways-of-training-llms-c57885f388ed

to the transitions of a finite-state machine (FSM). We implemented this FSM as a word graph, with a trie-based structure, built from the whitelist of permissible words. The word graph encodes all possible token combinations applicable to construct words in the vocabulary. Analogously, this can be thought as assembling different combinations of LEGO bricks to form the same final object. By matching the input with a state of the graph, we only allow valid tokens, and achieve backtracking with input-dependent generations.

Additionally, we tried to improve the quality of the generated outputs with beam search, without success. However, no specific benchmarks were used for quality check, since we can directly tell when a sentence is not phrased correctly in our generations. Beam search algorithm considers multiple alternative sequences before selecting the most optimal one, it aims to provide more robust and accurate generations, ensuring backtracking. We choose to employ beam search as it is a well-established method, further work on this task could employ other traditional or more experimental methods such as length penalization or top-p sampling[4].

This report is organized as follows: a background study of related work on constrained LLM decoding methods ; methodology, including data processing, model, word graph, constrained decoding and beam search ; results ; discussion on limitations, and suggestions of future improvement. The source code of this project is provided in a GitHub repository.[5]

3. Related Work

Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning (Geng et al., 2023)

This paper introduces a method for constrained decoding on specific tasks with complex structures by leveraging input-dependent grammars. The output space is specified using formal grammar, which is designed for various important NLP tasks, such as closed information extraction, entity disambiguation, and constituency parsing.

The decoding process produces tokens one by one, with the method intervening during decoding by pruning the probability distribution to include only the subset of tokens allowed by the formal grammar. This subset of allowed tokens is returned by an incremental parser (Angelov, 2009), abstracted as a completion engine. The paper highlights that this process is compatible with any decoding algorithm.

This work inspired our approach as input-dependent grammar and token pruning to restrict the output space aligns with our objectives. However, the formal grammar designed in this paper was tailor-made for the specific NLP tasks it covers. Consequently, it is overly specialized and incompatible with our objective of dynamically restricting the output to a predefined vocabulary. We aim to develop an analogous method, with appropriately designed grammar constraints.

---

[4] https://medium.com/@developer.yasir.pk/understanding-the-controllable-parameters-to-run-inference-your-large-language-model-30643bb46434

[5] https://github.com/ledondodo/difficulty_constrained_decoding

A General-Purpose Algorithm for Constrained Sequential Inference (Deutsch et al., 2019)

This paper introduces a general framework for expressing output structure constraints, using token pruning to restrict the output space during inference. It enables the application of constraints for fast and efficient inference.

The constraints are expressed as a set of finite-state automata (FSA), defined as (weighted) directed graphs. The transitions in the automata formulate constraints to prune the model's probability distribution. This framework supports both greedy decoding and beam search. The authors have made their implementation available as the gcd package on GitHub[6], which relies on Pynini and OpenFst libraries.

These methods are highly relevant in our project, and we implemented our constraints using this framework. However, as the approach is very general, we will also compare results with a more basic structure, implemented from scratch as a trie, to evaluate performance differences.

4. Methodology

The methods were implemented with Python 3.11.

4.1 Dataset

The datasets used in this project consist of English words annotated with their respective Common European Framework of Reference for Languages (CEFR) levels. CEFR is a language learning standard for categorizing language proficiency into six levels (A1, A2, B1, B2, C1, C2).

- The CEFR-J Word List[7]: Developed by Japanese researchers provides a CEFR-labeled word list. English words have been extracted from primary and secondary school textbooks (Years 3 to 10 of the UK school system) used in China, Korea, and Taiwan.
- Kaggle Dataset[8]: A collection of 10,000 English words labeled according to CEFR levels.

To ensure the datasets are suitable for the task, we expanded the word lists to include common variations, addressing the limitations of datasets that primarily list simple word forms. The expanded dataset should contain any form of word that our model allows to generate. These variations include:

- Case Variations: Title case, uppercase, and lowercase representations of words.
- Space-Prefixed Tokens: From a tokenizer perspective, we observed that some tokens can encode a space prefix to the word. We expand words to include these variations. For example, the word "cat" is expanded to "_cat" and tokenized as combinations like "_ca"-"t". Spaces are denoted by "_" to ease readability.

[6] https://github.com/CogComp/gcd
[7] http://cefr-j.org/download.html
[8] https://www.kaggle.com/datasets/nezahatkk/10-000-english-words-cerf-labelled

Combining these variations increases the data by a factor 6.

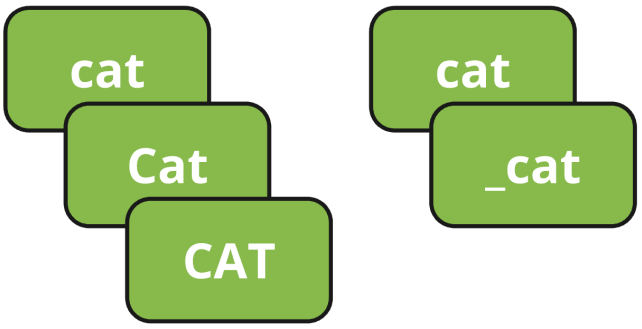

Fig. 1: Word variations handled in this project

The preprocessing excludes more complex variations such as verb conjugations, plural forms, and function words. These cases require non-trivial rules, as not all words undergo the same transformations (e.g., “cat” → “cats”, “tooth” → “teeth”) and some variations can not be applied to all words because function words or verbs do not have plural forms, and non-verbs do not have conjugated forms. Consequently, these expansions were left for potential future work.

The data is processed from the raw datasets with the help of the tokenizer. Initially, words are filtered according to the desired CEFR level (A1 for example) and stored in a list. These words are then expanded to the aforementioned variations, using basic string operations.

Finally, the data is tokenized ; for each word we gather every token of the tokenizer that is a subunit of the word, comparing them at the character level. These tokens are stored in the whitelist, while punctuation and separation tokens are stored in breaklists and spacelists.

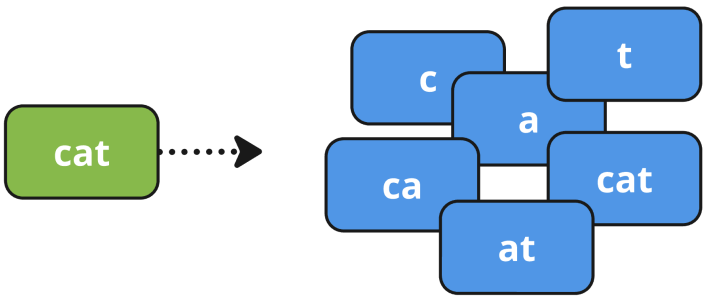

Fig. 2: Data tokenization

To optimize graph construction, a word whitelist is stored — a nested dictionary mapping each word to its constituent tokens.

4.2 Model

All the results in this project are computed with the SmolLM-135M-Instruct model[9], smaller compared to more common models used in today's NLP research which often have billions of parameters[10] (e.g., Llama3 with 70B parameters[11]).

4.3 Word Graph

To implement vocabulary constraints, we address the main challenge of dynamic behavior with a word graph. Language models tokenize words into subword units, leading to multiple possible token sequences for the same word. Inspired by the framework described in Deutsch et al., 2019, we formulate this structure as a finite-state machine (FSM), namely word graph. The FSM begins at a ROOT state and ends at a FINISHED state, with transitions between states representing tokens.

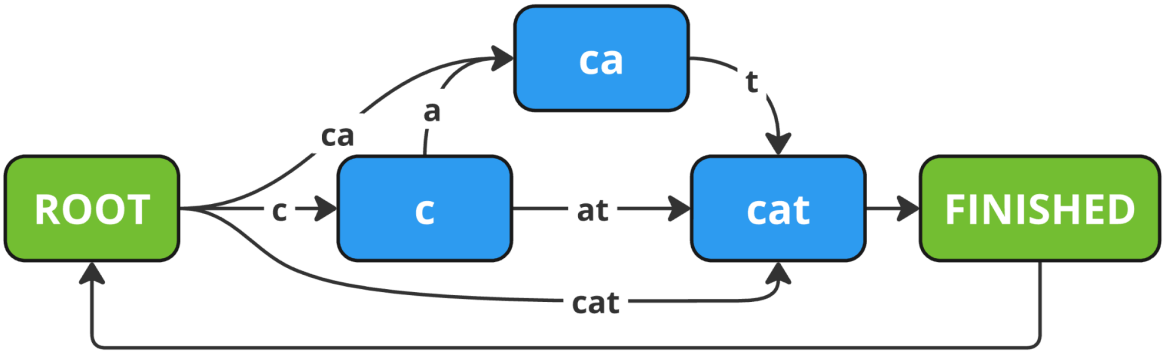

Fig. 3: Word graph for "cat"

This FSM can be implemented using libraries such as Pynini or a custom trie structure. In the trie implementation, states, arcs, and tokens are tracked in a dictionary. Initially, we define ROOT and FINISHED states. Then, for each word of the list, we call a recursive construction function from the ROOT node.

The recursive function attempts to form every combination of tokens (as strings) with the current state name — the ROOT state is considered as an empty string. If a combination matches the beginning of the word currently being constructed, a new state and transition are added to the graph, and the function is called recursively. The recursion ends when the state name entirely matches the word. A transition to the FINISHED state is finally added, and the graph depth updated. For loop efficiency, we benefit from the word whitelist to retrieve only tokens involved in the current word.

The word graph only needs to be computed once, offline, and stored. During inference, the graph is traversed to dynamically update constraints.

Three key functions support the graph's dynamic constraints:

---

[9] https://huggingface.co/HuggingFaceTB/SmolLM-135M
[10] https://www.techtarget.com/whatis/feature/12-of-the-best-large-language-models
[11] https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct

1. Match Function: Identifies the state of the graph corresponding to a given token sequence. First, it clips the sequence to the graph depth to reduce computational overhead — depth is the highest number of transitions (i.e. tokens) to reach the FINISHED state from the ROOT. It then iteratively calls the recursive match function from the root, for each starting point of the sequence.
2. Recursive Match Function: It compares the transitions leaving the given state with the first token of the sequence. When a match is found, the function calls itself recursively, using the remaining sequence (with the first token removed) and the state corresponding to the matching transition. Consequently, we are advancing through the graph, according to the sequence of tokens. The matching state is returned, or an error is raised if no valid transitions are found.
3. Transitions Function: Returns valid token transitions from a given reference state (i.e., token IDs of outgoing edges). Additionally to explicit transitions, when the reference state is the ROOT, the function also returns special tokens such as punctuations and separations, allowing them only between two words of the vocabulary.

By dynamically constraining the output space based on the token sequence, the FSM ensures backtracking capabilities, and enforces input-dependent vocabulary constraints during generation.

4.4 Constrained Decoding

We apply vocabulary constraints at inference time, within the decoding function of the language model. At each generation step, the whitelist of permissible tokens is retrieved from the word graph based on the current token sequence. This whitelist is applied to the model output using custom logits processors, pruning invalid tokens and restricting the output space. This ensures that the generated text always adheres to the predefined vocabulary constraints.

The detailed implementation of constrained decoding dynamic constraints is detailed as follows:

1. Logit Calculation: The model call produces logits for the next tokens based on the current token sequence (model input).
2. Whitelist Generation: The token sequence is processed by the match and transition functions of the word graph to retrieve the whitelist — the output space of the next token.
3. Logits Processing: Invalid tokens are pruned to restrict the output space. A custom logits processor masks out tokens not in the whitelist by setting their logits to: $-\infty$.
4. Token Selection: The modified logits are transformed into scores using log-softmax. The token with the best score is appended to the current sequence for the next generation step.

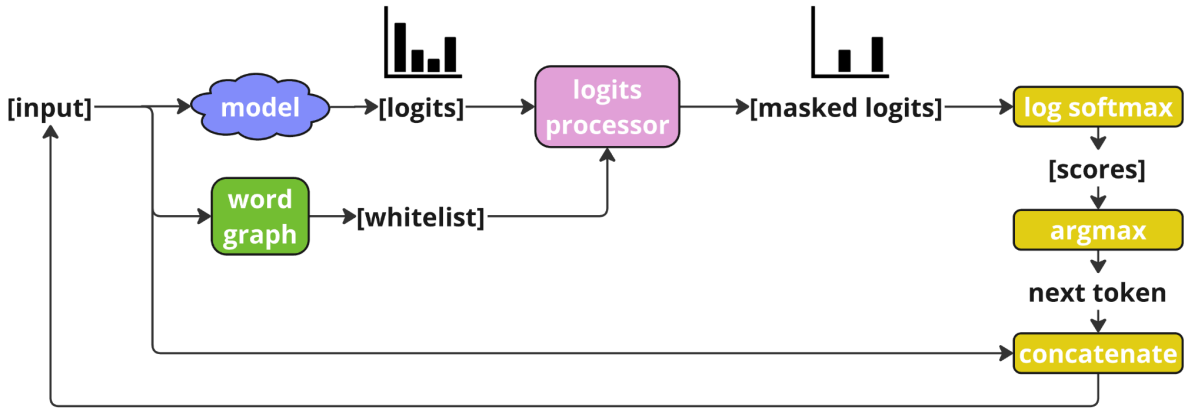

Fig. 4: Constrained decoding illustration. Input sequence of tokens serve to produce next tokens logits (from the model) and whitelist (from the word graph). Logits not included in the whitelist are masked by the logits processor. The masked logits are finally converted into scores and the top score token is elected to be the next token. The latter is concatenated with the input for the next step of generation.

4.5 Beam Search

Beam search serves as an additional mechanism to enhance decoding robustness. Unlike greedy decoding, which selects the most likely token at each step, beam search maintains multiple sequences (beams) simultaneously. By tracking the top B most probable sequences, the model can recover from earlier suboptimal token choices and explore alternative paths.

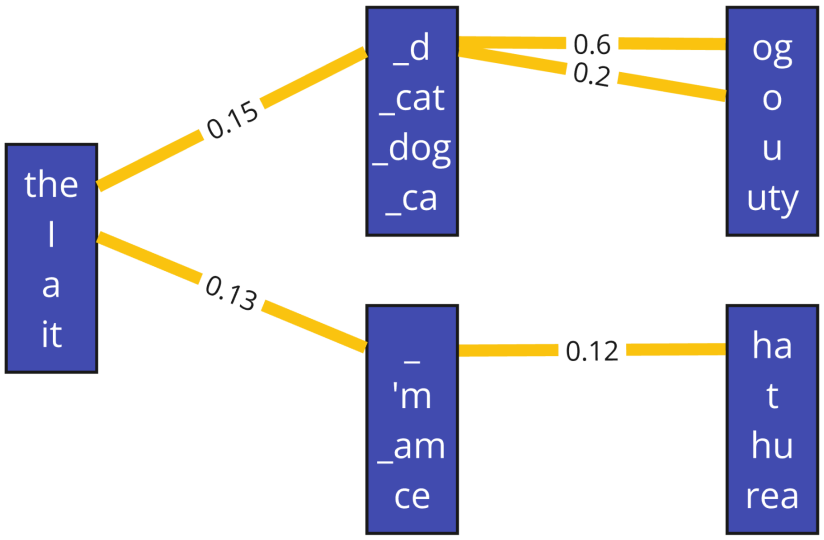

Fig. 5: Beam search with 3 beams (hypothetical example for illustration purpose).[12] Blue boxes represent top tokens sorted by scores, at different steps of generation. Yellow beams represent 3 hypothetical paths, combining the highest score tokens. Selected next token probabilities are denoted on each beam.

[12] A beam search visualizer is offered by Hugging Face: https://huggingface.co/spaces/m-ric/beam_search_visualizer

The whitelist of permissible tokens needs to be computed for each beam independently, as different beams may correspond to different states in the FSM.

To illustrate, consider the following hypothetical example:

- The sequence generated so far is “Trigonometry is related to ”.
- Assume the tokenizations: “geometry” → “geo”-“metry”, “geology” → “geo”-“logy”, and “maths” → “ma”-“ths”.
- Suppose “geometry” is disallowed by the whitelist, while “geology” and “maths” are contained in the whitelist.
- At the first step, “geo” has a higher probability than “ma.” However, at the second step, “geo” cannot form “metry” (whitelisted constraint), so the model backtracks to “ma”-“ths”, which is a valid sequence, and has a higher probability than sequence “geo”-”logy”.

Without beam search, the model would generate “Trigonometry is related to geology”. With beam search, the output would correctly be “Trigonometry is related to maths”. Beam search, therefore, enhances the robustness of generations by enabling error recovery and promoting adherence to vocabulary constraints.

In preliminary work, we explored beam search extensively to deepen understanding by developing custom implementations. The most important custom algorithm covered was a dynamic beam search, which maintained K beams at each generation step, ensuring their probabilities summed up to a specified threshold. Various strategies for the threshold value were tested:

- A. Constant decay
- B. Probability mass ratio (summing all beams prob together)
- C. Constant value
- D. Scores flattening approach:
    - D1. Constant value, softmax on flatten scores
    - D2. Constant value, softmax before flattening scores

Their implementations are detailed in the appendix.

Additionally, a normalization factor alpha helped stability (e.g., 0.05) when multiplying probabilities of N generation steps, such that:

$$p = \sqrt[N^{\alpha}]{p_1 p_2 \dots p_N}$$

Eq. 1: Sequential probabilities normalization, inspired from geometric mean and perplexity[13].

This exploration provided insights into customizing the algorithm, particularly the representation and manipulation of probabilities (e.g., differences between softmax and log-softmax outputs). Ultimately, a standard beam search was used for constrained decoding. However, the preliminary work highlighted key customization points, making it easier to integrate constrained decoding efficiently.

---

13 https://medium.com/nlplanet/two-minutes-nlp-perplexity-explained-with-simple-probabilities-6cdc46884584

5. Results

All the results were computed using an Apple M1 CPU with 8 cores.

5.1 Dataset

We measured the data processing time for different subset sizes.

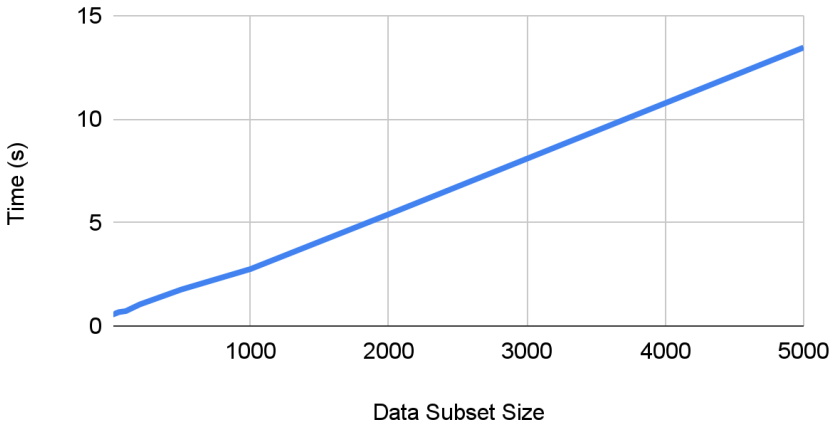

Fig. 6: Data processing computational cost

5.2 Word Graph

At a preliminary stage of the implementation, we compared two different structures for graph construction: Pynini and trie-based.

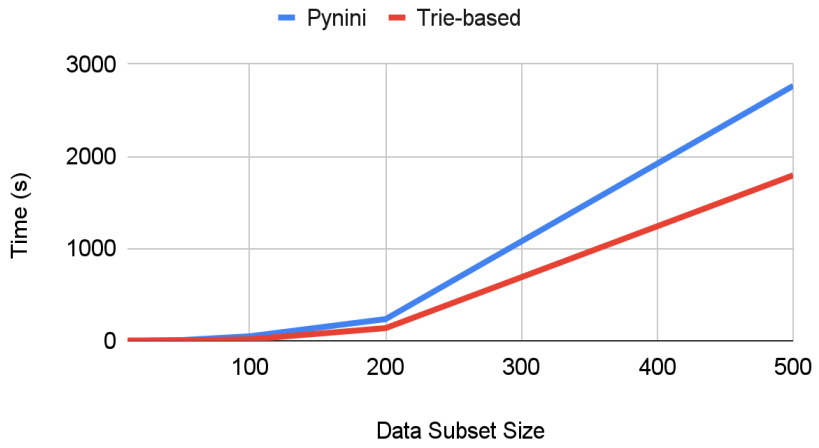

Fig. 7: Preliminary methods cost comparison

Revising the algorithms led to a significant improvement in graph construction time. We measured the graph building time for the final trie-structure implementation.

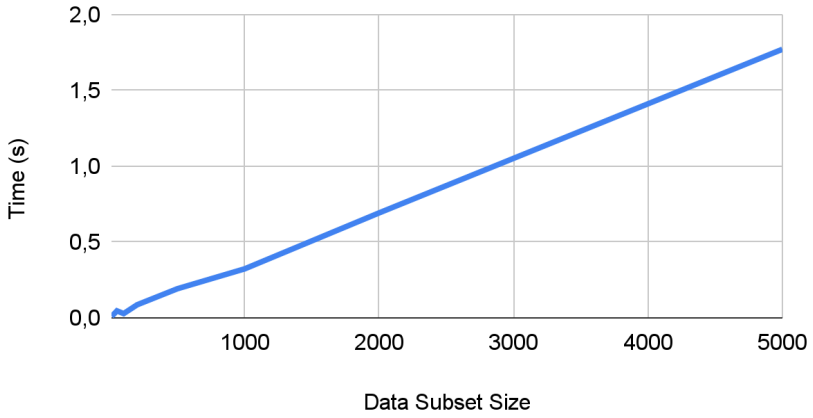

Fig. 8: Trie-based word graph construction cost

Figures 9 and 10 present graph anatomy features and file storage size for different subset sizes.

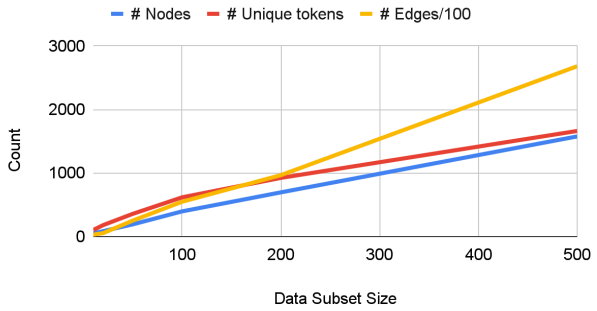

Fig. 9: Graph properties

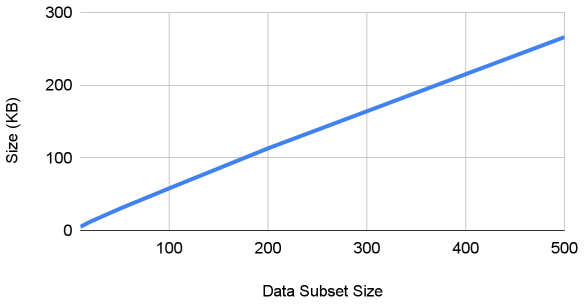

Fig. 10: Graph file size

Small graph examples are provided in the appendix. Their internal composition can be observed to understand the organization of states and token transitions.

Additionally, we developed predictive models to estimate the time complexity from empirical measures, primarily used to estimate computational cost of preliminary methods.

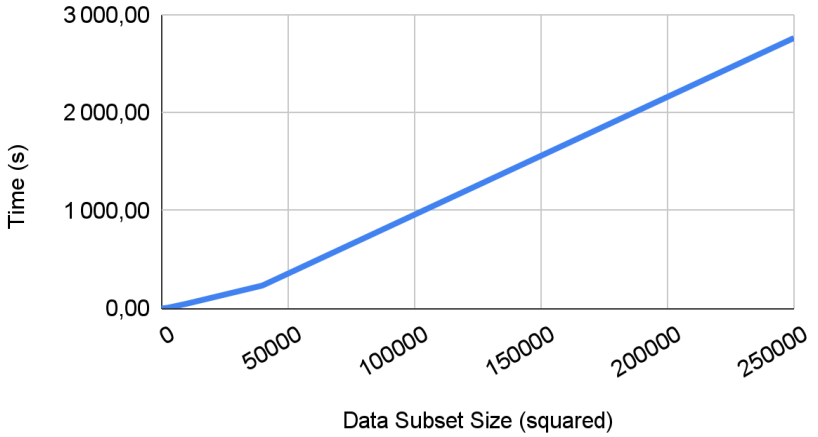

Fig. 11: Quadratic cost of preliminary methods

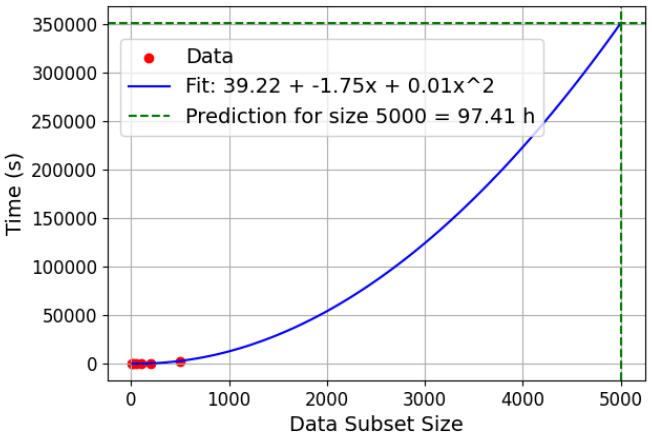

Fig. 12: Time predictions of preliminary methods, fitting a quadratic model MSE=578.74, R2=0.999

Dataset	Size	Predicted Time
CEFR-J A1	931	3 hours
CEFR-J A1 expanded	4655	3.5 days
CEFR-J	6709	1 week
CEFR-J expanded	33545	6 months

Table 1: Time predictions of preliminary methods

5.3 Constrained Decoding

We measured the computational cost for constrained decoding, in comparison with unconstrained decoding. To understand the time overhead, we measured specific sections of the code, isolating the impact of the constraints at each stage of the process.

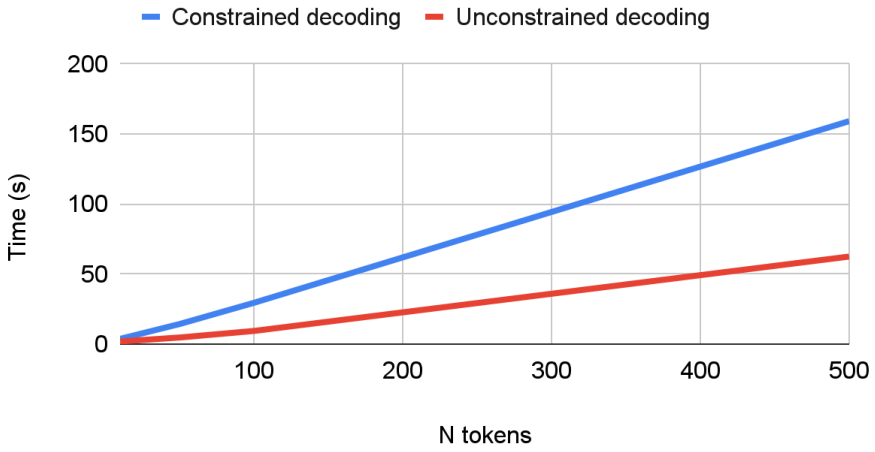

Fig. 13: Generation time, with 2 beams

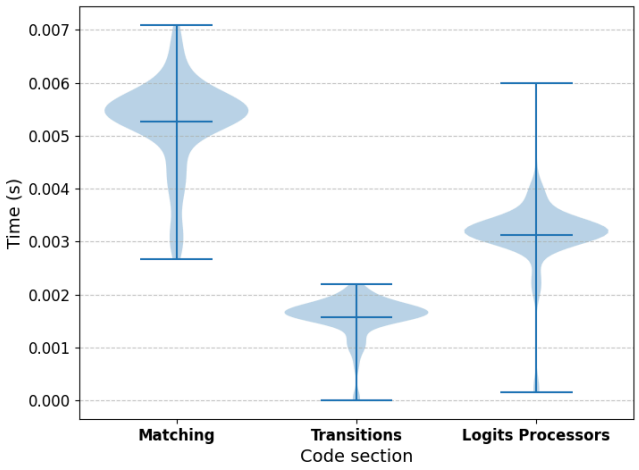

Fig. 14: Time overhead, generating N=100 tokens and 2 beams with Kaggle dataset, and prompt “Do you like dogs?”

We compared outputs for different prompts and datasets.

Prompt 1		Describe your house
Dataset	**Output**	
CERF-J	String	A house island that island that island that island that island that island
	Tokens	A/ house/ is/l/and/ that/ is/l/and/ that/ is/l/and/ that/ is/l/and/ that/ is/l/and/ that/ is/l/and/
Kaggle	String	A house is a place where we live, work, and play. It's a place where we can restaurant,
	Tokens	A/ house/ is/ a/ place/ where/ we/ live/,/ work/,/ and/ play/./ It/'s/ a/ place/ where/ we/ can/ rest/aur/ant/,/
Prompt 2		Do you like dogs?
Dataset	**Output**	
CEFR-J	String	Addanswer Ice, Ice! Ice! Ice! Ice
	Tokens	Ad/d/ans/we/r/ /I/c/e/,/ I/c/e/!/ I/c/e/!/ I/c/e/!/ I/c/e/
Kaggle	String	Addanswer I love them! They bring so much love and care into our life. They're a great
	Tokens	Ad/d/ans/we/r/ /I/ love/ them/!/ They/ bring/ so/ much/ love/ and/ care/ into/ our/ life/./ They/'re/ a/ great/

Table 2: Outputs comparison for different prompts and datasets, generating N=25 tokens. Outputs are given as string, as well as a sequence of tokens separated by a slash character.

5.4 Beam Search

We measured computational cost at generation for different numbers of beams, overall, as well as in specific sections of the code.

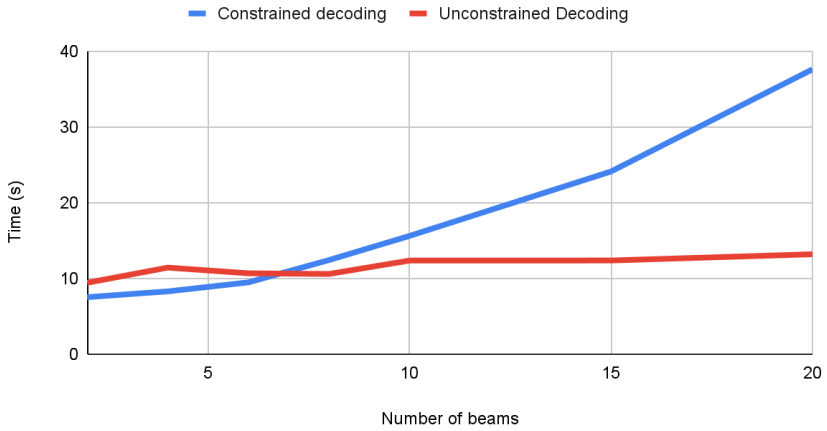

Fig. 15: Generation time, with N=100

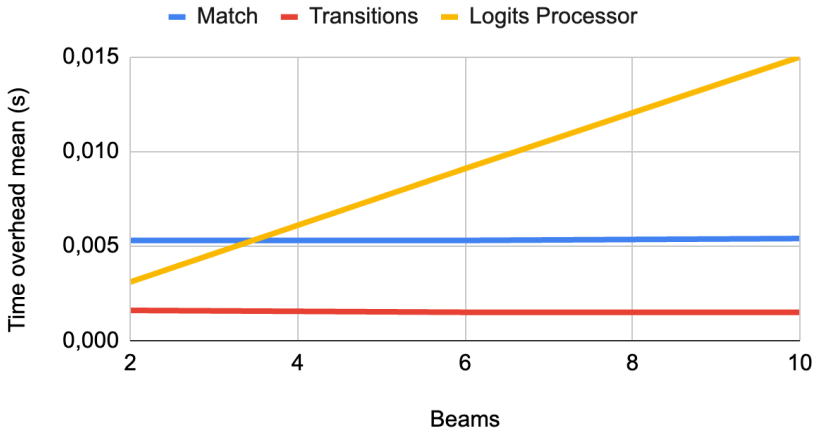

Fig. 16: Time overhead means for different code sections, generating N=100 tokens with Kaggle dataset, and prompt “Do you like dogs?”

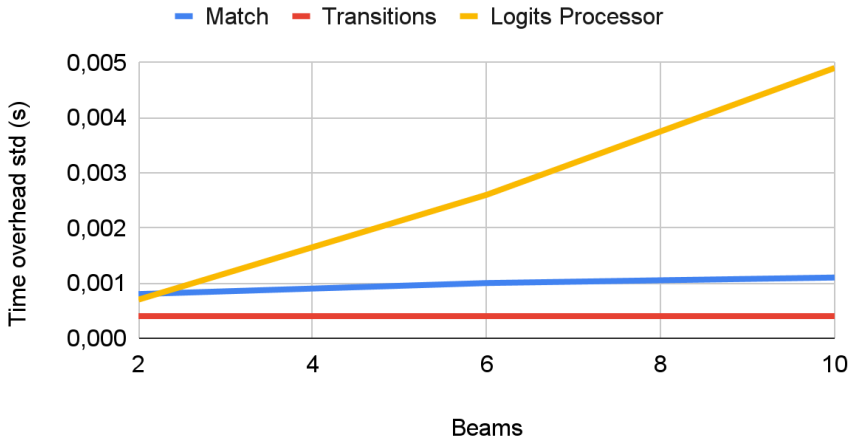

Fig. 17: Time overhead standard deviations for different code sections, generating N=100 tokens with Kaggle dataset, and prompt “Do you like dogs?”

We compared outputs with the same prompt and dataset over different numbers of beams.

Prompt		Reply only with the base forms of words, don’t use any inflections.
Beams	**Output**	
2	String	AUTUMN What a great question! I'm glad you're interested in this topic. I'm glad you're
	Tokens	A/UT/UM/N/ /What/ a/ great/ question/!/ I/'m/ glad/ you/'re/ interested/ in/ this/ topic/./ I/'m/ glad/ you/'re/
10	String	AUTUMN What a great question! I'm glad you're interested in this topic. I'm glad you're
	Tokens	A/UT/UM/N/ /What/ a/ great/ question/!/ I/'m/ glad/ you/'re/ interested/ in/ this/ topic/./ I/'m/ glad/ you/'re/

Table 3: Outputs comparison, varying beams, generating N=25 tokens with Kaggle dataset. Outputs are given as string, as well as a sequence of tokens separated by a slash character.

Results of preliminary work on beam search exploration are provided in the appendix.

6. Discussion

6.1 Dataset

The measurements of data processing exhibited a linear computational cost (Fig. 6), demonstrating efficiency and scalability. This ensures the method can be extended to more datasets, as long as they are correctly formatted.

However, the data expansion could be improved to handle non-trivial cases, such as function words, verb conjugations, or plural forms. On the other hand, more complete datasets could mitigate these challenges, avoiding spending time on elaborated expansion algorithms design.

Additionally, while the handling of punctuation and separation tokens is functional, it is limited to predefined lists (e.g., breaklist and spacelist). Expanding them to the wide variety of special tokens found in tokenizer vocabularies — SmolLM spread to 50K token vocabulary — would require more advanced and elaborated methods to ensure robustness.

6.2 Model

The choice of SmolLM-135M-Instruct was intentional, as the focus of the work was not on comparing different models but on modifying the decoding function. This approach allows for scaling to any model with any set of weights. We selected this model for its lightweight nature, which facilitated rapid experimentation and allowed us to focus on optimizing the

decoding process without the computational overhead of larger, more resource-intensive models. However, further work could focus on enhancing output quality, leveraging appropriate evaluation metrics and larger models.

6.3 Word Graph

At the start of the project, both Pynini and trie-based implementations had suboptimal functions; however, even at this early stage, the trie structure demonstrated approximately 1.5 times faster performance than Pynini (Fig. 7). While Pynini is a versatile tool for finite-state transducers, it did not offer enough specialized advantages to justify its computational cost in this specific case. Its general-purpose design made it less efficient than a custom trie structure, which proved to be more suitable due to its simplicity and effectiveness in handling token transitions. Consequently, we focused our efforts on optimizing the trie-based approach.

Through algorithmic refinements, we successfully reduced the time complexity of graph construction from quadratic to linear (Fig. 8), addressing one of the most significant and challenging aspects of the project. Key optimizations included the use of a word whitelist implemented as a nested dictionary, which effectively mitigated the combinatorial explosion within the recursive function, considerably reducing for-loop size. Furthermore, replacing lists with sets for membership checks significantly enhanced performance by enabling faster token comparisons. The graph properties and file storage size plots (Fig. 9, Fig. 10) show a linear trend (or less) and also stand for a sanity check of graph construction algorithms. These improvements ensured scalability while maintaining computational efficiency.

The time prediction model (Fig. 12) helped to identify the impracticality of quadratic-time algorithms (Fig. 11) for large datasets, and clarified the importance of optimizing the algorithm to handle real-world, larger datasets effectively (Table 1).

Ultimately, we opted for a flexible graph approach that starts at a ROOT state and terminates at a FINISHED state. This design enforces the primary constraint of restricting outputs to the permissible vocabulary, aligning with the project's objectives. However, it does not impose additional constraints on syntax or structure of the generated text, as this was beyond the scope of the project. Future work could explore more complex graph designs to enhance output quality while maintaining the scalability achieved in this implementation.

6.4 Constrained Decoding

The constrained decoding successfully achieves the primary objective of the project, restricting the output to words predefined by the dataset vocabulary.

The decoding process operates with a linear time complexity relative to the number of generation steps N (Fig. 13). However, in comparison with unconstrained decoding our approach takes approximately 3 times longer. This additional overhead is attributed to the logit masking by processors, the graph state matching, as well as getting the transitions of the matched state at each step (Fig. 14).

Testing with various prompts and datasets (Table 2) revealed that, while constrained decoding effectively limits the output space to the vocabulary, challenges remain in producing high-quality outputs. For instance, due to gaps in the vocabulary, the model

occasionally generates unexpected or nonsensical outputs, such as replacing missing verbs with unrelated nouns : “the house is” → “the house island”, “is” is missing in the CEFR-J dataset. The same behavior can be caused by expected gaps because of level filtering : “we can rest” → “we can restaurant”, “rest” is a B1 word in Kaggle dataset. These observations highlight the importance of comprehensive and well-curated datasets, or refined structure constraints for improving output quality.

6.5 Beam Search

Beam search performance exhibited distinct behaviors (Fig. 15). For constrained decoding, the generation time increased more than linearly when increasing the number of beams, whereas for unconstrained decoding, the time remained relatively constant. Further analysis revealed that the overhead for graph operations remained constant regardless of the number of beams (Fig. 16 and 17). However, the overhead associated with the logit processor increased linearly with the number of beams. As mentioned earlier, each beam must be processed individually, as they can lead to different graph states. Consequently, processing beams requires iterative handling, and even a constant time overhead (e.g., graph operations) would be repeated which adds to the computational cost.

Beam search method intends to improve output quality, by adding stability in token selection. Interestingly, our results showed that the number of beams did not significantly impact the quality of the generated outputs (Table 3). This observation suggests that maintaining a beam count as low as two is sufficient for achieving the same results, while also minimizing computational costs.

It is worth noting, however, that while our limited testing did not reveal a significant impact of beam count on output quality, the possibility of unobserved effects remains. More extensive testing and experimentation might uncover situations where beam count plays a more crucial role.

Future work could explore additional methods for improving output quality, such as length penalization or random sampling. These are well-established approaches that were not covered in this study but could complement or enhance the use of beam search.

7. Conclusion

This project successfully fulfills its primary objectives by restricting output from an architectural perspective, offering an efficient and scalable solution to adhere to a predefined vocabulary. The constrained decoding restricts the tokens output space to a predefined word list by dynamically pruning logits at inference time. It can be adapted to other datasets with appropriate formatting. It demonstrates scalability through efficient graph construction and operations. The linear computational complexity could be achieved by formulating constraints as a trie-based word graph, and optimizing algorithms with set operations and nested dictionaries. It ensures that the solution remains practical for large datasets.

However, despite these successes, the project faced challenges in generating high-quality outputs. This limitation can be attributed to several factors: the incompleteness of the datasets used ; the complexity of expanding dataset to non-trivial variations ; the flexibility of

the graph design at the cost of syntax control ; and the lack of more advanced stabilization methods beyond beam search. These issues highlight areas for improvement and provide starting points for further exploration.

The approach was focused on the decoding process, while not evaluating the model choice impact on generations. Additionally, comparing output quality can be difficult, and robust evaluation metrics would also be valuable contributions to this area of research. Future work could focus on addressing these shortcomings.

This project provides a support for constrained decoding techniques and insights into improving vocabulary-controlled text generation. We hope it inspires future research to build upon these findings, pushing the boundaries of controlled language model decoding.

References

- Geng, S., Josifoski, M., Peyrard, M., & West, R. (2024). Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. arXiv preprint. https://arxiv.org/abs/2305.13971
- Deutsch, D., Upadhyay, S., & Roth, D. (2019). A general-purpose algorithm for constrained sequential inference. In M. Bansal & A. Villavicencio (Eds.), Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL) (pp. 482–492). Hong Kong, China: Association for Computational Linguistics. https://aclanthology.org/K19-1045/ https://doi.org/10.18653/v1/K19-1045
- Gorman, K. (2016). Pynini: A Python library for weighted finite-state grammar compilation. In B. Jurish, A. Maletti, K.-M. Würzner, & U. Springmann (Eds.), Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata (pp. 75–80). Berlin, Germany: Association for Computational Linguistics. https://aclanthology.org/W16-2409/ https://doi.org/10.18653/v1/W16-2409
- Angelov, K. (2009). Incremental parsing with parallel multiple context-free grammars. In A. Lascarides, C. Gardent, & J. Nivre (Eds.), Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009) (pp. 69–76). Athens, Greece: Association for Computational Linguistics. https://aclanthology.org/E09-1009/

Appendix

Setup for Pynini

Pynini and its dependencies need to be installed with Python 3.8, but then the project was implemented with Python 3.11.

Virtual environment for installing Pynini dependencies conda create --name pynini python=3.8 conda activate pynini
Install OpenFST (10min) wget http://www.openfst.org/twiki/pub/FST/FstDownload/openfst-1.6.9.tar.gz tar -zxvf openfst-1.6.9.tar.gz cd openfst-1.6.9 ./configure --enable-grm --prefix=*/path/to/folder*/openfst make make install
Install Re2 (5min) git clone https://github.com/google/re2 cd re2 git checkout 2018-04-01; git pull make make test export CPATH=*/path/to/folder*/re2_lib/usr/local/include export LD_LIBRARY_PATH=*/path/to/folder*/re2_lib/usr/local/lib export LIBRARY_PATH=*/path/to/folder*/re2_lib/usr/local/lib export CPATH=${CPATH}:*/path/to/folder*/openfst/include export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:*/path/to/folder*/openfst/lib export LIBRARY_PATH=${LIBRARY_PATH}:*/path/to/folder*/openfst/lib make install DESTDIR=*/path/to/folder*/re2_lib make testinstall
Install Pynini (20min) wget http://www.opengrm.org/twiki/pub/GRM/PyniniDownload/pynini-2.0.0.tar.gz tar -zxvf pynini-2.0.0.tar.gz cd pynini-2.0.0 python setup.py install
Virtual environment for the project conda create --name *venvname* python=3.11 conda activate *venvname* pip install -r requirements.txt

Table 4: Command line setup for external libraries

Word Graph Examples

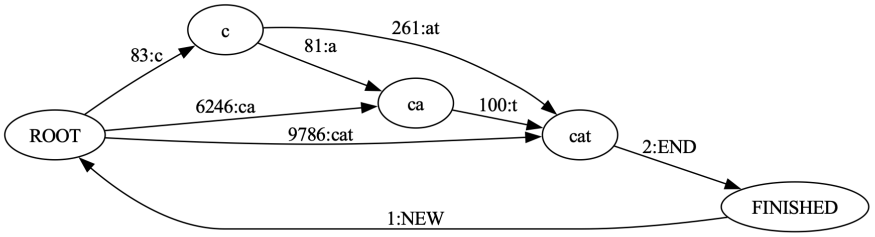

Fig. 18: Word graph for [“cat”]

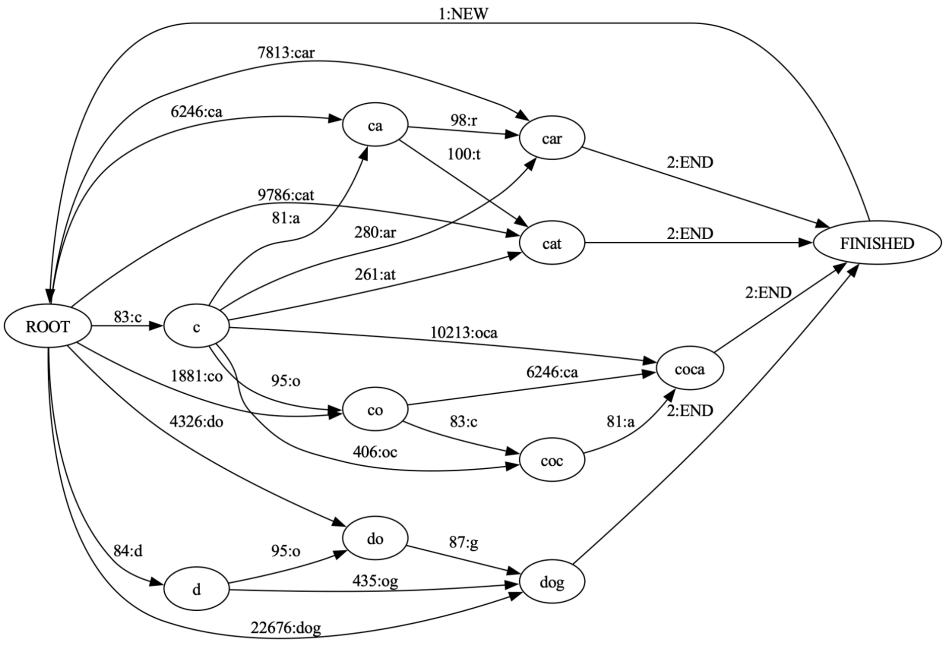

Fig. 19: Word graph for [“cat”,”car”,”coca”,”dog”]

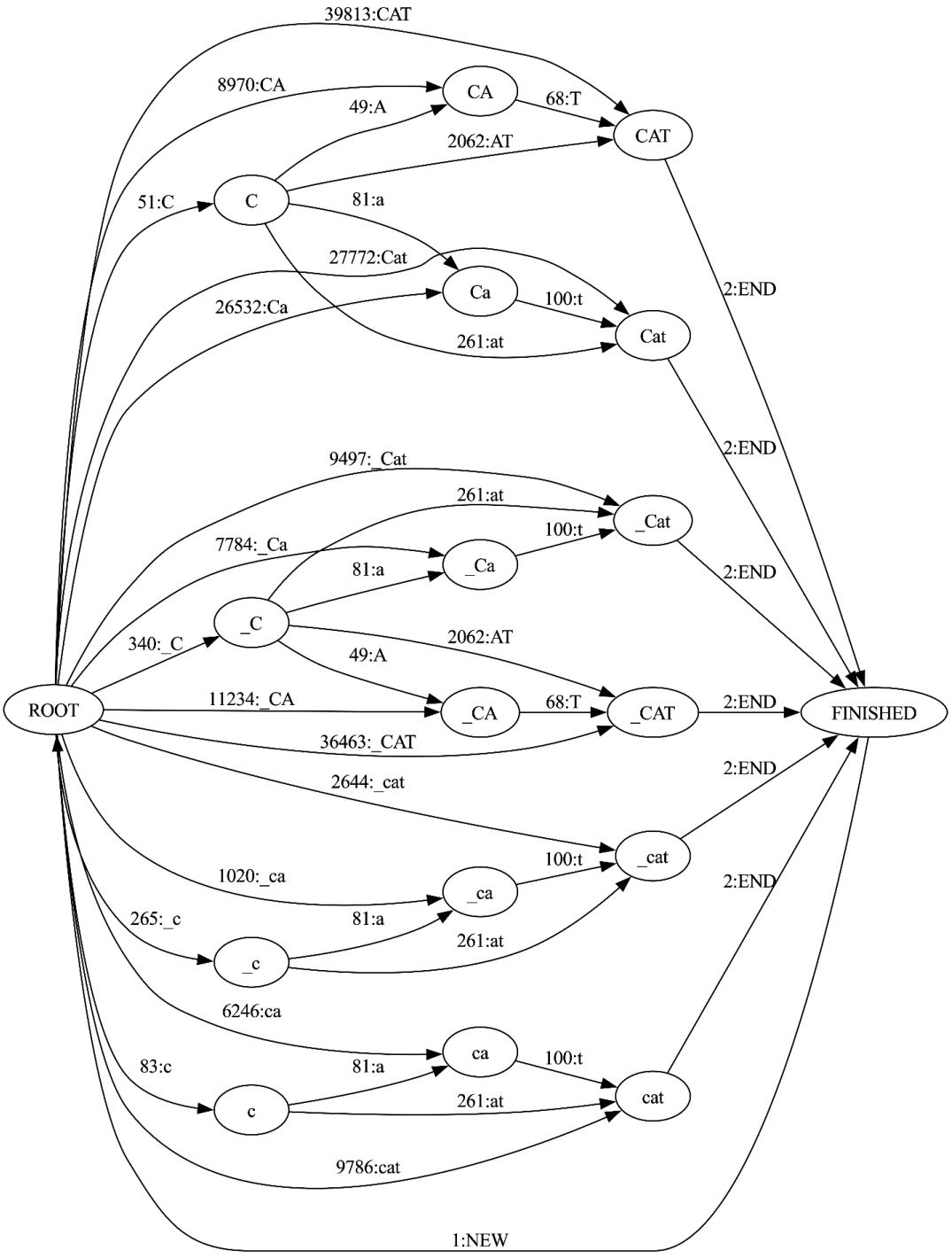

Fig. 20: Word graph for [“cat”] expanded

Dynamic Beam Search (preliminary work)

The custom dynamic beam search achieves two functions. First, it maintains K beams at each generation step, ensuring their probabilities summed up to a specified threshold (represented as an orange dotted line). The number of beams K is clipped to *bmax*. Second, it blacklists the punctuation tokens, restricting the output to generate a special token every *max_consecutive* steps, at least. Punctuations, when generated, are represented by a vertical green dotted line. Additionally, a normalization factor alpha is used for stability when multiplying probabilities of N generation steps (Eq. 1).

Strategy A: Constant decay

Threshold at step i formulated as $t_0(t_1^i)$.

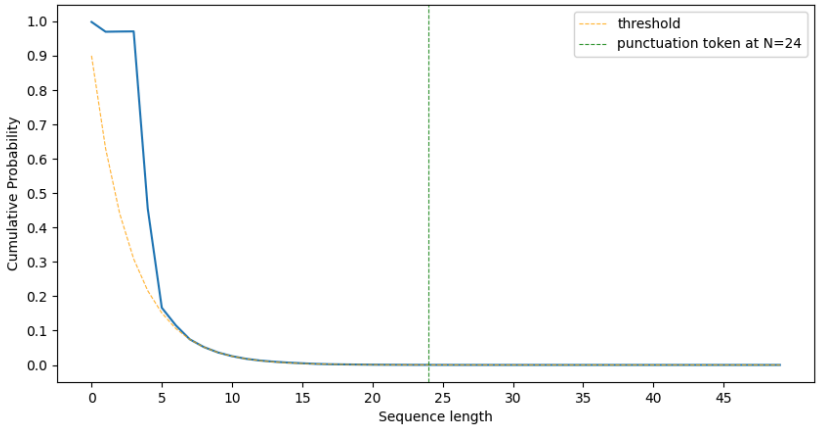

Fig. 21: Cumulative prob of dynamic K beams, with strategy A
alpha=0.05, bmax=100, max_consecutive=20, blacklist=True
t0=0.9, t1=0.7

Strategy B: Probability mass ratio

The probability mass *pmass* is obtained by summing all beam probabilities together. The threshold is then given by $pmax * pmass$, with $pmax$ as ratio.

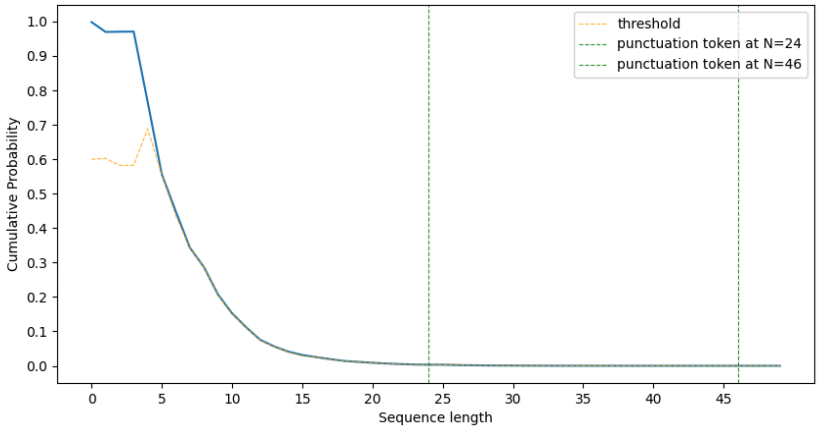

Fig. 22: Cumulative prob of dynamic K beams, with strategy B
alpha=0.05, bmax=100, max_consecutive=20, blacklist=True
pmax=0.6

Strategy C: Constant value
Threshold is $pmax$.

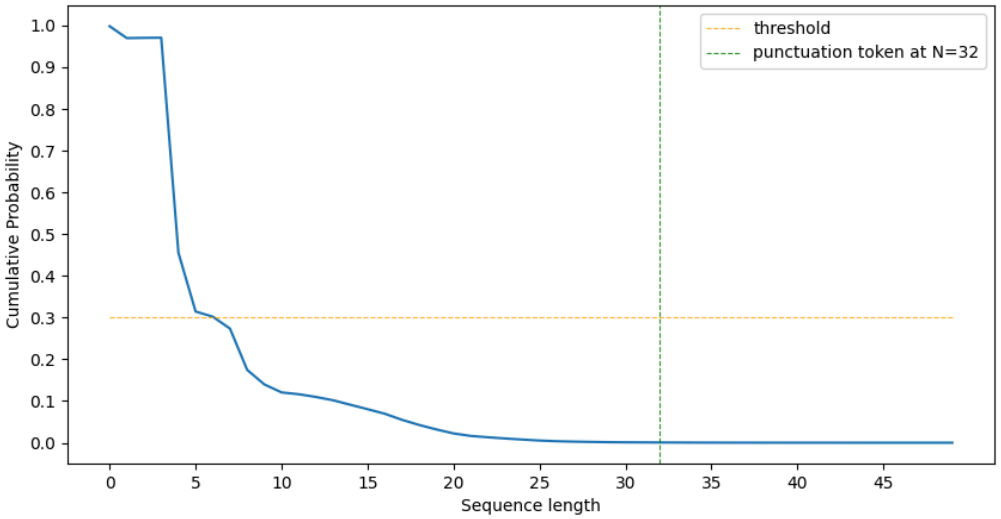

Fig. 23: Cumulative prob of dynamic K beams, with strategy C
alpha=0.05, bmax=100, max_consecutive=20, blacklist=True
pmax=0.3

Strategy D: Constant value with softmax
The scores are first processed with softmax, to obtain values between 0 and 1, considered as probabilities. However, the softmax can be applied at two specific locations. Threshold is $pmax$.

Alternative D1: softmax on flatten scores
By first flattening scores, then applying softmax, we ensure that the values sum up to 1.

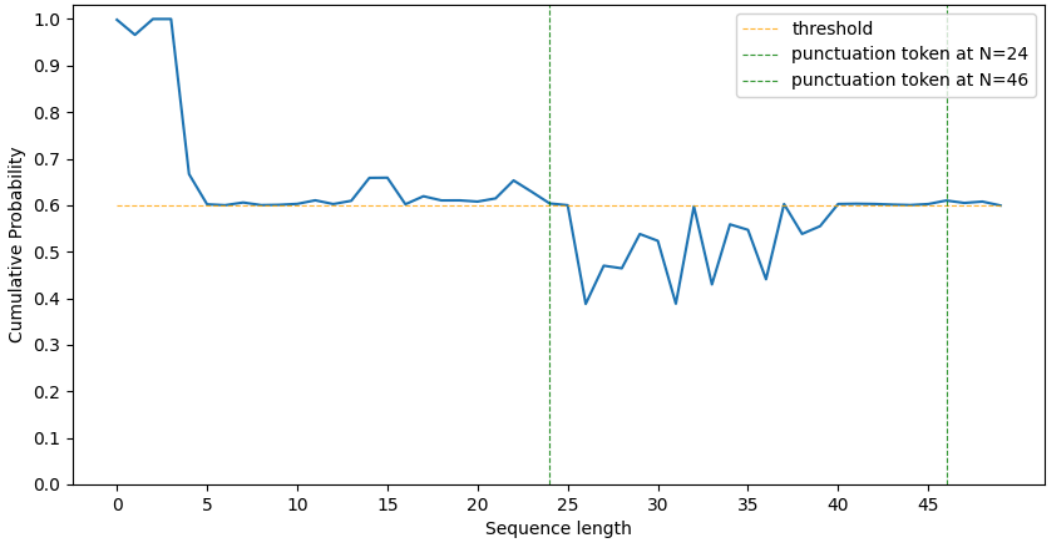

Fig. 24: Cumulative prob of dynamic K beams, with strategy D1
alpha=0.05, bmax=100, max_consecutive=20, blacklist=True
pmax=0.6

Alternative D2: softmax before flattening scores

By first applying softmax, then flattening scores, we would have a more dynamic behavior. Each beam values would sum up to 1, then by flattening them, the entire values would sum up to K (number of dynamic beams).

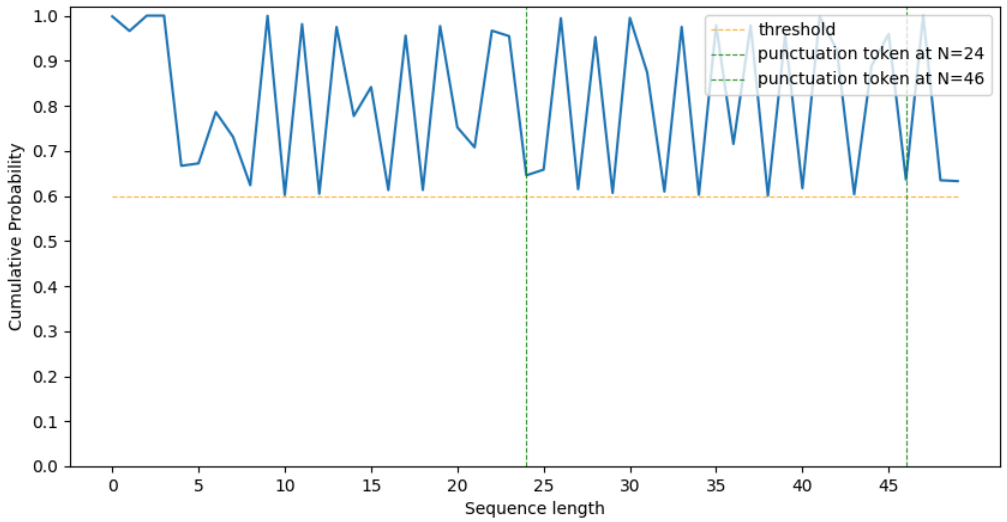

Fig. 25: Cumulative prob of dynamic K beams, with strategy D2
alpha=0.05, bmax=100, max_consecutive=20, blacklist=True
pmax=0.6