Enhancing Constraint Beam Decoding in Transformers library

Junyao Chen
jc6385
Computer Science
Columbia University
junyao.chen@columbia.edu

Pranitha Natarajen
pn2435
Computer Science
Columbia University
pn2435@columbia.edu

Raavi Gupta
rg3637
Computer Science
Columbia University
rg3637@columbia.edu

***Abstract*—This paper presents an extension of constrained beam decoding capabilities in the Hugging Face Transformers library to support structured generation requirements. While existing implementations enforce basic lexical constraints, real-world applications often demand precise control over token order and positional requirements. We introduce two novel constraint types: (1) *template constraints* that enforce fixed token positions within generated text, and (2) *ordered constraints* that require specified tokens to appear in sequence without position restrictions. Our implementation combines dynamic beam allocation with logit masking strategies to maintain decoding efficiency while satisfying structural constraints. Through systematic evaluation across GPT-2, BERT, and LLaMA models on a curated dataset of 2,700 samples from CausalBank, we demonstrate 98% template constraint accuracy with LLaMA-8B while maintaining reasonable generation speeds (5.4s/sentence). The ordered constraint implementation achieves 75% accuracy under similar conditions. Our analysis reveals a fundamental trade-off between constraint satisfaction rates and computational overhead, particularly for larger models. The proposed extensions address longstanding community requests for structured generation control and have been integrated into the Transformers library. The code can be found at github.com/jychen630/constrained_decoding while the pull request can be found at github.com/huggingface/transformers/pull/38030.**

***Index Terms*—beam search, constrained decoding, ordered constraint, template constraint**

I. Introduction

A. Background and Motivation

Constrained beam decoding is a technique in language generation that guides model outputs by enforcing constraints during decoding. In the Hugging Face Transformers library, current support is limited to simple lexical constraints that force the inclusion or exclusion of specific words without regard to their order or structure.

However, real-world applications, such as legal drafting or code generation, require more structured control over generated text. These use cases call for structured constraints, including ordered constraints (enforcing the sequence of target words) and template constraints (ensuring adherence to a specified output format).

This work responds to long-standing feature requests for structured constraints in the Hugging Face community, as evidenced by multiple GitHub issues. We extend the library's constrained beam decoding functionality to support richer constraints, contributing directly to the open-source codebase to enhance user-defined control over output generation.

B. Problem Statement

The constrained decoding module in the Hugging Face Transformers library enforces only position-agnostic lexical constraints without any mechanisms for managing word order or structural formats. This restricts its utility in applications where output must follow specific sequences or templates. Despite clear demand from the user community, there is no native support for ordered or template-based constraints. As a result, developers are unable to use constrained decoding for tasks requiring predictable structure, such as form generation or systems with formatting requirements.

C. Objectives and Scope

This work aims to extend the constrained beam decoding module in the Hugging Face Transformers library by introducing support for two new constraint types:

- Template constraints – to enforce adherence to predefined output structures.
- Ordered constraints – to ensure that specified tokens appear in a given sequence.

We focus on augmenting the existing dynamic beam allocation algorithm to efficiently support the new constraint types. The implementation is designed for integration into the upstream Transformers codebase, with the goal of maintaining decoding performance while improving constraint flexibility. Semantic contraints, and encoder-decoder models are beyond the scope of this work.

II. Literature Review

A. Review of Relevant Literature

Constrained decoding has been an important area of research in natural language generation (NLG), particularly when specific words need to be included or excluded from the generated text. One key paper, "Guided Generation of Cause and Effect" [1], introduced constraint-based techniques that focused on ensuring the logical flow of text, particularly for cause and effect relationships. Another paper, "Improved Lexically Constrained Decoding for Translation and Monolingual Rewriting" [5], tackled the issue of word inclusion, allowing

specific words to be forced into generated text, which is useful for tasks like machine translation or rewriting.

In the field of image captioning, "Guided Open Vocabulary Image Captioning with Constrained Beam Search" [4] took a similar approach but applied it to visual contexts. This work allowed the model to generate captions with certain predefined vocabulary items based on the image content, demonstrating how lexical constraints can guide generation in different domains.

A more recent development, "Fast Lexically Constrained Decoding with Dynamic Beam Allocation" [3], improved the efficiency of lexically constrained decoding by dynamically adjusting the beam search process. This method speeds up the generation process while still ensuring that the required words are included in the output. However, this approach focused mainly on lexical constraints and did not address more complex structural constraints like word order or output format. The Hugging Face Transformers library uses this implementation for enforcing lexical constraints.

B. Identification of Gaps in Existing Research

Previous research has focused on lexical constraints which works for tasks like translation or captioning. But these methods don't support the kind of structured control we want. Furthermore, the open-source community highlights the need for template-style decoding constraints. We fill this gap by extending Hugging Face's implementation, adopting dynamic beam allocation to support richer, template and ordered constraints.

III. Methodology

A. Data Collection and Preprocessing

We constructed evaluation dataset from **CausalBank**, a large-scale, open-domain, sentence-level parallel causal corpus. The original corpus is annotated with *different causal types* (e.g., cause-effect, enablement, reason-result). From each causal type, we sampled a subset of sentences, resulting in a curated pool of **2.7k samples**.

For each sampled sentence, we randomly selected **5–10 tokens** to serve as *constraint tokens*. All other token positions were replaced with blanks (`""`), forming the basis of our two constraint evaluation tasks:

- **Template Constraint Dataset:** Each entry contains a `starting_text` and a `template`, which is a list of fixed constraint tokens interleaved with blank slots. The model must generate a continuation that fills in the blanks while preserving the fixed tokens at their specified positions. An example entry is shown below:

```
{ "starting_text": "But first",
  "template": ["", "", "", " have", "", "",
       " that", "", "", "", " imbalance",
      "", " calories", "", "", " the", "",
      "", "", ""] }
```

- **Ordered Constraint Dataset:** Using the same sentences and `starting_text` values as the template set, we removed all blank slots from the `template` field. The result is a sequence of constraint tokens that must appear *in order*—but not necessarily contiguously—in the generated output. An example entry is shown below:

```
{ "starting_text": "is a",
  "template": [" of", " much", " than", "
      the"] }
```

These datasets allow systematic evaluation of language models under both position-specific (template) and order-specific (ordered) generation constraints.

B. Model Selection

We evaluate our constrained decoding methods across three LLMs that vary in architecture, scale, and training paradigm:

- **GPT-2 (small)**: An autoregressive transformer with 117M parameters, GPT-2 is selected for its compatibility with causal decoding and widespread use in constrained generation baselines. It serves as an accessible reference point for assessing decoding behavior in smaller-scale models.
- **BERT-base-cased**: Although BERT is a masked language model (MLM) rather than an autoregressive decoder, we include it to probe the adaptability of constraint logic in masked token prediction and to illustrate that template- or order-aware decoding is feasible in bidirectional contexts. BERT-base has 110M parameters.
- **Meta-Llama-3-8B-Instruct**: Instruction-tuned LLaMA 3 model is an autoregressive model with 8 billion parameters. As of the time, it's writing one of the most popular baseline model utilized in academics setting.

C. Optimization Procedure(s)

Our method is threefold. Two algorithms were applied to TemplateConstraint, resulting one is faster than the other. The idea of fast algorithm for TemplateConstraint is extended to the algorithm of OrderedConstraint.

TemplateConstraint - Dynamic Programming We adopt a dynamic programming-based decoding strategy for template-constrained generation, extending the standard beam search to a three-dimensional grid over time steps and constraint progress. At each time step, the algorithm maintains partial hypotheses in a beam for every possible number of constraints completed. For each cell `Grid[t][c]`, candidate hypotheses are formed by combining three sources:

- **Open generation (g):** If a hypothesis is not currently satisfying any constraint, it generates freely using the model's next-token distribution.
- **Constraint initiation (n):** If a new constraint is to be started, the model enforces the next token as specified in the constraint sequence.
- **Constraint continuation or blank (s):** If a constraint is already in progress, the next action depends on whether the upcoming token is a blank (`""`) or a fixed word.

All candidate hypotheses from these sources are scored, and the top-k are retained for each cell. Constraint fulfillment is tracked through an internal index, and constraint satisfaction

is deferred until final hypothesis selection at the end of generation. The algorithm 1 describes the pseudocode.

The dynamic programming approach takes quadratic computation time with respect to the vocabulary size, resulting 20 minutes per output generation on a GPU-enabled hardware Tesla V100-SXM2-32GB with GPT2. In the below section, the logit masking algorithm significantly reduces the computation.

TemplateConstraint - Logit Masking We implement logit processor that enforces constraint generation at each decoding step through token-level masking and scoring adjustments. We implement a linear-time logit processor that enforces the token template during decoding by dynamically masking invalid outputs, leaving the only-valid constrained token with non-zero score. The only non-zero token is then guaranteed to be selected by the regular beam search algorithm. The processor tracks the current slot index (`template_pos`) in the template sequence, where each slot may be:

- A fixed token (e.g., `" have"`), which must be generated exactly.
- A wildcard (`""`), which permits any token to be generated.

At each step, the `TemplateConstraintLogitsProcessor` performs the following:

1) If the current slot is a fixed token, leave the score of that token unchanged while all other tokens are masked logits set to $-\infty$.
2) If the current slot is a wildcard, no masking is applied.
3) Upon generating, a token that satisfies the current slot, is incremented to advance the constraint. After all constraints are satisfied, the traditional beam search algorithm continues to finish the generation until the stopping criteria is met.

OrderedConstraint - Soft Logit Masking

The **OrderedConstraint** relaxes the constraint to require only that specified tokens appear in a particular order, not at fixed positions. The `OrderedConstraintLogitsProcessor` starts by maintaining a pointer to track the next expected token in the constraint list. When the expected token is generated, the pointer is incremented to move to the next constraint. Instead of masking to an infinite score, we apply a **soft guidance** where the logit for the next expected token is boosted by a constant score. Other tokens are optionally penalized by subtracting a constant score.

D. Profiling tools and methods

We utilize Weights & Biases (WandB) to systematically track and visualize our experiments. Comprehensive reports detailing our findings and evaluations can be accessed at the following links: Template Constraint and Ordered Constraints.

E. Evaluation Metrics

Our primary objective was to evaluate whether a language model, when provided with a starting text and a corresponding template, successfully generates outputs that adhere to the specified template constraints. Accordingly, accuracy, defined as the proportion of outputs satisfying the template, served as a central evaluation metric. In addition, we investigated the computational efficiency of the generation process, with particular attention to how model size impacts generation time. To this end, we measured both time per sentence and time per token under varying conditions. These included changes in beam size, maximum sequence length, and the underlying model architecture. Our evaluation thus focused on three key metrics across these scenarios: template satisfaction accuracy, average generation time per sentence, and average generation time per token.

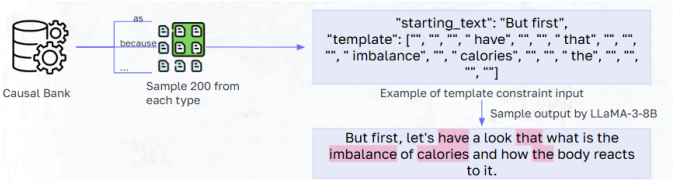


Fig. 1: The pipeline for evaluating the templates on the CausalBank dataset.

IV. Experimental Results

In this section, we describe the results of various experiments conducted.

A. Experimental Setup

Once the dataset is prepared, we perform inference using multiple models under varying hyperparameter settings to generate outputs conditioned on different templates and corresponding starting texts. The details of the dataset utilized for this evaluation are provided in Section III-A.

B. Performance Comparison of Different Models

TABLE I: Template Constraint Evaluations

Model	Beams	Returns	MaxLen	Accuracy	Time/Sentence	Time/Token
gpt2	3	3	100	0.94	1.25	0.01
gpt2	5	5	100	0.96	1.42	0.01
gpt2	3	3	20	0.13	0.21	0.01
gpt2	5	5	20	0.13	0.23	0.01
bert-base-cased	3	3	100	0.75	1.10	0.01
bert-base-cased	5	5	100	0.80	1.35	0.01
bert-base-cased	3	3	20	0.06	0.16	0.01
bert-base-cased	5	5	20	0.06	0.17	0.01
Meta-Llama-3-8B-Instruct	3	3	100	0.98	5.41	0.06
Meta-Llama-3-8B-Instruct	5	5	100	0.99	5.98	0.06
Meta-Llama-3-8B-Instruct	3	3	20	0.10	1.02	0.05
Meta-Llama-3-8B-Instruct	5	5	20	0.10	1.17	0.06

TABLE II: Ordered Constraint Evaluations

Model	Beams	Returns	MaxLen	Accuracy	Time/Sentence	Time/Token
gpt2	3	3	100	0.78	1.06	0.01
gpt2	5	5	100	0.57	1.20	0.01
bert-base-cased	3	3	100	0.65	1.03	0.01
bert-base-cased	5	5	100	0.40	1.21	0.01
Meta-Llama-3-8B-Instruct	3	3	100	0.75	5.30	0.05
Meta-Llama-3-8B-Instruct	5	5	100	0.51	5.86	0.06

Algorithm 1 Template Constraints - Dynamic programming

procedure TEMPLATECONSTRAINTSEARCH(model, input, constraints, maxLen, numC, k)
startHyp ← model.getStartHyp(input, constraints)
Grid ← initGrid(maxLen, numC, k) {initialize beams in grid}
Grid[0][0] ← startHyp {constraints now also store the id till which constraint c has been fulfilled; id starts from -1}
for $t = 1$ **to** $maxLen - 1$ **do**
  for $c = 0$ **to** $\min(t, numC)$ **do**
    $n, s, g \leftarrow \{\}$
    for all hyp in Grid[t-1][c] **do**
      if hyp.isOpen() **then**
        output ← model.generate(hyp, input, constraints) {generate new open hyps}
        $g \leftarrow g \cup output$
      end if
    end for
    if $c > 0$ **then**
      for all hyp in Grid[t-1][c-1] **do**
        if hyp.isOpen() **then**
          $n \leftarrow n \cup model.start(hyp, input, constraints)$ {start new constrained hyps}
        else
          if constraints[c][id + 1] == "" **then**
            $g \leftarrow g \cup model.generate(hyp, input, constraints)$ {generate blank tokens}
          else
            $s \leftarrow s \cup model.continue(hyp, input, constraints)$ {continue unfinished}
          end if
        end if
      end for
    end if
    Grid[t][c] ← top-k from $(n \cup s \cup g)$ by model.score() {k-best scoring hypotheses}
  end for
end for
topLevelHyps ← Grid[:][numC] {get hyps in top-level beams}
finishedHyps ← hasEOS(topLevelHyps) {finished hyps have generated the EOS token}
bestHyp ← argmax_h model.score(h) {select highest-scoring finished hyp}
return bestHyp
end procedure=0

C. Analysis of Results

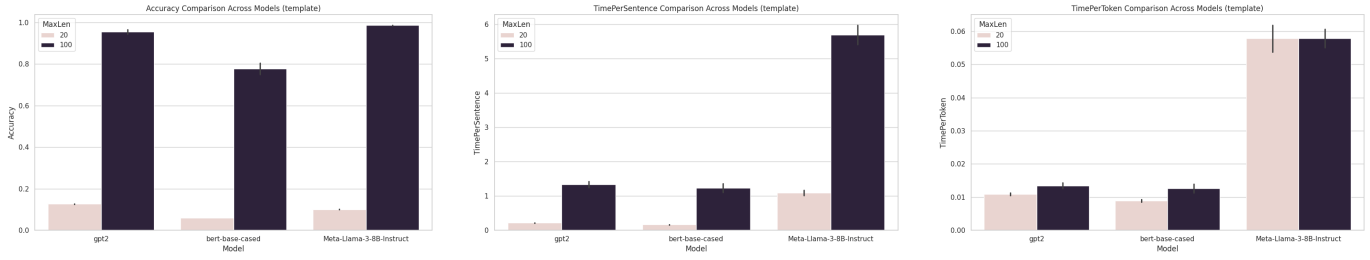
We evaluated three models—`gpt2`, `bert-base-cased`, and `Meta-Llama-3-8B-Instruct`—under two generation settings: **template constraint** and **ordered constraint**, with variations in beam width and sequence length. The results, present in Tables I and II, reveal several key trends:

a) Impact of Generation Mode: Overall, model accuracies were significantly higher in the template constraint across all configurations. For instance, `gpt2` (beam=3, MaxLen=100) achieved an accuracy of 0.9448 in template constraint mode compared to only 0.7791 in ordered constraint mode. A similar drop is observed for `Meta-Llama-3-8B-Instruct`, where the accuracy fell from 0.9837 to 0.7455 under the same settings. This stems from the fact that ordered constraints incorporates any number of tokens between the fixed tokens which require them to have a larger seqlen than template constraint in order to satisfy the same constraint.

b) Model-wise Performance: `Meta-Llama-3-8B-Instruct` consistently achieved the highest accuracies, especially in the template constraint mode with long sequences (MaxLen=100), reaching 0.9870. However, this came at the cost of significantly longer inference time per sentence and token. In contrast, `bert-base-cased` was the fastest model but also had the lowest accuracies in most configurations, especially with short sequences.

c) Influence of Sequence Length: All models showed a large drop in accuracy when MaxLen was reduced from 100 to 20. For example, `gpt2` (beam=3) dropped from 0.9448 to 0.1254 in template constraint mode. This suggests that the models struggle to satisfy constraints when enough tokens are not given for generation.

d) Trade-offs: There is a clear trade-off between accuracy and decoding time. While `Meta-Llama-3-8B-Instruct` offers superior generation



(a) Template Satisfaction accuracy. (b) Average time taken per sentence. (c) Time taken per token.

Fig. 2: LLaMA-8B-Instruct achieves a template satisfaction accuracy of 98%. This is expected to increase as the maximum sequence length is raised. However, LLaMA also takes the most time for inference.

(a) Template Satisfaction accuracy. (b) Average time taken per sentence. (c) Time taken per token.

Fig. 3: LLaMA-8B-Instruct achieves a template satisfaction accuracy for ordered constraits of 70%. This is expected to increase as the maximum sequence length is raised. However, LLaMA also takes the most time for inference.

quality, its inference time is several times that of `gpt2` or `bert-base-cased`. In time-sensitive applications, this may favor smaller models, especially if moderate accuracy is acceptable.

V. Discussion

A. Interpretation of Results

The results overview can be found in Figures 2 and 3.

B. Comparison with Previous Studies

Previous studies introduced the concepts of Template Constraints and Ordered Constraints. However, they lacked integration with the Transformers library which is the most widely used framework for text generation with language models.

C. Challenges and Limitations

The `OrderedConstraintLogitProcessor` relies on a constant score added to or subtracted from the tokens scores in order to encourage or penalize the soft generation. The constant score is therefore a hyperparameter. A future work is to develop reasonable heuristics to determine such constant scores. Finding a suitable score may consider several factors, including but not limited to the number of constrained tokens have been satisfied at the current position of generation, the remaining length of the output relative to the `max_len` limit, and the current scores of the constrained tokens.

D. Future Directions

One important aspect to evaluate further is fluency. Since we're working with constrained generation, there's always a tension between enforcing structure and preserving natural, creative output. So far, we've only done human evaluations, which helped us catch major issues, but we haven't tried automatic metrics like BLEU or BERTScore yet. These could give us a more standardized way to assess quality, especially when comparing across different decoding strategies. We did implement support for multiple constraints, but the original GitHub issue was focused on single constraints, so we stuck to that for most of our testing and analysis. Extending our setup to handle multiple or even nested constraints would be a natural next step. It would also be interesting to try enforcing other types of constraints beyond lexical, for instance, syntactic patterns or semantic roles, which would push the model further in terms of structure and control. Another idea we had but didn't fully pursue was using a trie to manage multiple constraints more efficiently during decoding. This could help prune the search space early when multiple constraints share common prefixes, and make decoding more scalable as we increase constraint complexity.

VI. Conclusion

A. Summary of Findings

We introduced support for template and ordered constraints in Hugging Face's constrained decoding framework, addressing long-standing community requests for structured gener-

ation control. As part of this, we adapted dynamic beam allocation, that was earlier used only for lexical constraints, to work with our new constraint types. Experiments show that while larger models like LLaMA-8B-Instruct achieve higher constraint satisfaction, they incur longer inference times, and all models benefit significantly from increased sequence length. Future work will explore fluency evaluation, support for multiple or nested constraints, and trie-based decoding to scale constraint handling efficiently.

B. Contributions

Our implementation has been written in a pull request to `huggingface/transformers` library, with 540 lines of code, 6 files changed in three regions in `transformers/generation` module: `generation/beam_constraints.py`, `generation/logits_process.py` and artificial test cases files. Our code has passed the initial CI/CD checks and under review by huggingface community maintainers.

VII. Acknowledgment

We thank Professor Kaoutar El Maghroui and the TAs for their mentorship.

References

[1] Guided Generation of Cause and effect. Zhongyang Li, Xiao Ding, Ting Liu, J. Edward Hu, Benjamin Van Durme. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI 2020).

[2] Chris Hokamp and Qun Liu. 2017. Lexically Constrained Decoding for Sequence Generation Using Grid Beam Search. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1535–1546, Vancouver, Canada. Association for Computational Linguistics.

[3] Matt Post and David Vilar. 2018. Fast Lexically Constrained Decoding with Dynamic Beam Allocation for Neural Machine Translation. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 1314–1324, New Orleans, Louisiana. Association for Computational Linguistics.

[4] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. 2017. Guided Open Vocabulary Image Captioning with Constrained Beam Search. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 936–945, Copenhagen, Denmark. Association for Computational Linguistics.

[5] J. Edward Hu, Yuntian Deng, and Alexander M. Rush. 2019. Improved Lexically Constrained Decoding for Translation and Monolingual Rewriting. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 839–850, Minneapolis, Minnesota. Association for Computational Linguistics.

Appendix A
Dashboard

We developed a dashboard that displays the input and output of the constraints and generated output. The figure 4 and figure **??** shows the dashboard interface, which includes:

- An input text field for the starting prompt
- A constraint input area where users can specify constraints in JSON format
- A toggle between template and ordered constraint modes
- Real-time generation of up to 5 different outputs

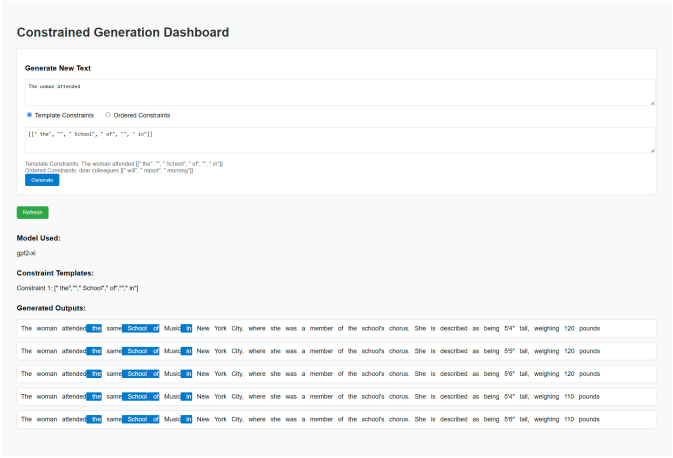


Fig. 4: The dashboard UI with a template constraint example.

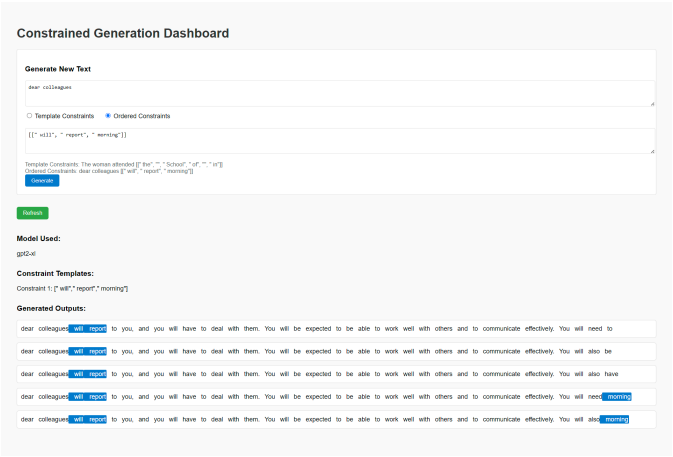


Fig. 5: The dashboard UI with an ordered constraint example.

- Visual highlighting of constrained tokens
- A refresh button to update the display with new output

All generated outputs are saved to a JSON file. Please find the code of the demo at https://github.com/jychen630/constrained_decoding/tree/master/demo.

Appendix B
An example of template constraint generation algorithm walkthrough.