TreePiece: Faster Semantic Parsing via Tree Tokenization

Sid Wang
Meta Inc. USA
yuwang2020@meta.com

Akshat Shrivastava
Meta Inc. USA
akshats@meta.com

Aleksandr Livshits
Meta Inc. USA
alll@meta.com

Abstract

Autoregressive (AR) encoder-decoder neural networks have proved successful in many NLP problems, including Semantic Parsing – a task that translates natural language to machine-readable parse trees. However, the sequential prediction process of AR models can be slow. To accelerate AR for semantic parsing, we introduce a new technique called TreePiece that tokenizes a parse tree into subtrees and generates one subtree per decoding step. On TOPv2 benchmark, TreePiece shows 6.1 times faster decoding speed than standard AR, and comparable speed but significantly higher accuracy compared to Non-Autoregressive (NAR).

1 Introduction

Autoregressive (AR) modeling (Sutskever et al., 2014) is a commonly adopted framework in NLP where the next prediction is conditioned on the previously generated tokens. This paper focuses on AR approach for Semantic Parsing (Wong, 2005), an NLP task that converts a natural language utterance to a machine-interpretable symbolic representation called logical form. The sequence of actions to derive a logical form is isomorphic to a directed tree and often referred to as a parse tree (Zettlemoyer and Collins, 2005).

The runtime latency of AR linearly correlates to the output length and could result in low inference speed (Gu et al., 2017; Wang et al., 2018). Non-Autoregressive (NAR) modeling (Gu et al., 2017; Wei et al., 2019; Ma et al., 2019), on the other hand, is able to produce outputs in parallel and reduce latency by an order of magnitude (Ghazvininejad et al., 2019). However, NAR performs considerably worse than its AR counterparts without extra training recipes (Wang et al., 2019; Zhou and Keung, 2020; Su et al., 2021). The quality benefits of AR models therefore motivates us to improve their speed, rather than exploring NAR.

Our contributions

• We propose a novel approach of tokenizing parse trees into large units called TreePiece units, and then building an AR model that predicts one TreePiece unit at a time, thus reducing the number of steps needed to generate a full parse tree. To the best of our knowledge, we are the first to extend subword-tokenizer algorithm to semantic trees such that each token is a subtree.

• We validate our approach on TOPv2 benchmark and show that TreePiece decoding is 6.1 times faster than standard AR with less than 0.15% accuracy degradation, and nearly as fast as NAR with up to 0.7% accuracy gains.

• We provide theoretical proofs to support our main algorithms and their variants.

2 Methods

2.1 Parse tree

In this paper, we utilize the hierarchical semantic representations based on intent and slot (Gupta et al., 2018), allowing for modeling complex compositional queries in task-oriented dialog systems. See Figure 2 (LHS) for an example. Now let us
define a few recurring notions in the paper:

**Definition 2.1 (Ontology).** A parse tree node is called an ontology iff it represents an intent/slot, prefixed by in: and sl: respectively.

**Definition 2.2 (Skeleton).** The skeleton of a parse tree is the subtree that consists of all ontologies.

**Definition 2.3 (Utterance leaf).** A text-span node is called an utterance leaf iff its parent is a slot\(^1\).

### 2.2 TreePiece tokenizer

#### 2.2.1 Tokenizer algorithm

**Definition 2.4.** TreePiece tokenizer is an algorithm that segments any skeleton into subtrees of an open vocabulary. The minimal open vocabulary of a TreePiece tokenizer is called a TreePiece vocabulary, where an element is called a TreePiece unit.

**Definition 2.5 (TreePiece simplex).** Let \( \mathcal{V} \) be a TreePiece vocabulary and \( t \) be any TreePiece unit. A TreePiece simplex \( p \) is a mapping from \( \mathcal{V} \) to the unit interval \([0, 1]\) such that \( \sum_{t \in \mathcal{V}} p(t) = 1 \).

We propose Algorithm 1 as our TreePiece tokenizer, a Viterbi-type (Viterbi, 1967) forward-backward (Nagata, 1994a) algorithm which computes the optimal tokenization and probability for given skeleton \( S \).

**Notations in Algorithm 1:**

1. \( \mathcal{T} \) is the set of all subtrees of \( S \) that share the same root as \( S \) denoted by \( \mathcal{T}_0 \); \( \mathcal{T}_i \) admits a natural filtration \( \mathcal{T}_0 \subseteq \cdots \subseteq \mathcal{T}_{d-1} \subseteq \mathcal{T}_d \subseteq \cdots \subseteq \mathcal{T} \), where \( \mathcal{T}_d \) is the set of all depth-\( d \)-subtrees.
2. \( \mathcal{L} \) is the log probability on \( \mathcal{T} \) as follows:

\[
\mathcal{L}(t) = \begin{cases} 
\log p(t) & \text{if } t \in \mathcal{V}, \\
-\infty & \text{otherwise}
\end{cases}
\]

where \( \mathcal{V} \) is the TreePiece vocabulary and \( p \) the TreePiece simplex.

3. for efficiency we apply Filter(\( t, t' \)) to restrict to subtrees \( t' \) such that (a) \( t' \) is a subtree of \( t \), (b) the set difference \( t \Delta t \) has exactly one connected component and it is a TreePiece unit.

In summary, the forward step uses dynamic programming inductive on tree-depth to update all subtrees’ log-probabilities and eventually obtain \( \mathbb{P}(S; p) \) – the probability of the skeleton \( S \). The forward step also returns a map \( \mathcal{P} \) that stores for each \( t \in \mathcal{T} \) the optimal position of its previous partition. Then in the backward step we can backtrack along the path \( S, \mathcal{P}(S), \mathcal{P}(\mathcal{P}(S)), \cdots \) to recover the optimal partition \( \pi_S(p) \).

---

\(^1\)We adopt the decoupled form proposed in (Aghajanyan et al., 2020), which simplifies compositional representations by ignoring text spans that are not utterance leaves.
Algorithm 1 Forward-backward algorithm

**Input:** TreePiece vocabulary $V$, TreePiece simplex $p$, and skeleton $S$.

**Output:** Partition $π_S(p)$ and probability $P(S; p)$.

1. Initialize $L ← \log(p)$
2. $p' ←$ Constant map from $T$ to BOS token
3. $d_{\text{max}} ← $ Depth of $S$ 
4. for $d = 1, 2, \cdots, d_{\text{max}}$ do \hspace{1em} // Forward begins
   1. for $t \in T_d$ do
      1. for $d' = 1, 2, \cdots, d$ do
         1. for $t' \in \text{Filter}(T_{d'}, t) \text{ do}$
            1. $L^* ← L'(t') + \sum_{τ \in Δ^*} \log p(τ)$
            2. if $L^* > L(t)$ then
               1. $L(t) ← L^*$, $p' ← t'$
               2. $P(S; p) ← \exp(L(S))$ \hspace{1em} // Forward ends
5. while $t_{\text{curr}} \neq$ BOS token, do
   1. $t_{\text{prev}} ← \text{Filter}(t_{\text{curr}}, t)$ \hspace{1em} // Backward begins
   2. $Δ^* ← t_{\text{prev}} \cdot Δ_{t_{\text{curr}}}$
   3. $π_S(p) ← π_S(p) \cup \{t_{\text{curr}}\}$ \hspace{1em} // Backward ends
6. return $π_S(p), P(S; p)$

2.2.2 Tokenizer training

The performance of Algorithm 1 relies on the quality of the TreePiece vocabulary $V$ and TreePiece simplex $p$. To improve the quality of $V$ and $p$, we propose a two-stage training procedure:

**Stage 1. Generate $V$:** Let $\mathcal{S}$ represent all skeletons of a given training corpus. Similar to Byte Pair Encoding (BPE) (Gage, 1994; Sennrich et al., 2015), we obtain the TreePiece vocabulary $V$ and map $F_0$ between TreePiece units and their frequencies in $\mathcal{S}$. For details see Appendix A.

**Stage 2. Update $p$:** initialize the TreePiece simplex $p_0$ as the normalized frequency $p_0(t) =: F_0(t)/\sum_{τ \in V} F_0(τ)$ for all $τ \in V$, and then solve for $p_{t+1}$ iteratively as follows:

$$p_{t+1} = \arg\max_{\bar{p}} \sum_{S \in \mathcal{S}} \mathbb{E}_{π_S} [\log P(S, π; p)|S; p_t].$$

(1)

In general, problem (1) is NP-hard as it involves summing over $\Pi_S$, the set of all possible partitions $π$ of a skeleton $S$:

$$\mathbb{E}_{π_S} [\log P(S, π; p)|S; p_t] = \sum_{π \in \Pi_S} \log P(S, π; p) \cdot \frac{P(S, π; p_t)}{P(S; p_t)}.$$  

(2)

Algorithm 2 EM algorithm

Choose $N_0 ∈ \mathbb{N}^+, ε_0 > 0$; initialize $i ← 0$, $Δ ← +\infty$, $L_{\text{prev}} ← −\infty$.

while $i < N$ and $Δ > ε_0$ do
   1. $L_{\text{curr}} ← 0$, $F^* ←$ Zero function on $V$
   2. for $S ∈ \mathcal{S}$ do
      1. Compute $π_S(p_i) \text{ and } P(S; p_i) \Rightarrow \text{E-step}$
      2. $L_{\text{curr}} ← L_{\text{curr}} + \log P(S; p_i)$
   3. for $t ∈ π_S(p_i)$ do
      1. $F^*(t) ← F^*(t) + 1$
   4. for $t ∈ V$ do
      1. $p_{i+1}(t) ← \frac{F^*(t)}{\sum_{τ \in V} F^*(τ)} \Rightarrow \text{M-step}$
      2. $i ← i + 1$, $Δ ← L_{\text{curr}} − L_{\text{prev}}$
      3. $L_{\text{prev}} ← L_{\text{curr}}$

To solve (1) in polynomial time, we propose Algorithm 2 (whose E-step uses Algorithm 1) and impose the following assumption on the joint distribution of $S$ and $π$:

$$P(S, π; p) \propto \left\{ \begin{array}{ll} \prod_{τ \in π} p(τ), & \text{if } π = π_S(p) \\ 0, & \text{otherwise,} \end{array} \right.$$  

(3)

where $π_S(p) = \arg\max_{π \in π_S} \prod_{τ \in π} p(τ)$. Applying (3), we see that all but one summand in (2) vanish. The following Theorem claims that Algorithm 2 solves for (1) under assumption (3):

**Theorem 2.6.** Let $p_0, p_1, \cdots$ be TreePiece simplices obtained from Algorithm 2. If (3) is true, then (1) holds. Moreover, $\sum_{S \in \mathcal{S}} \log P(S; p_t)$ is monotonically non-decreasing in $i$.

For proof of Theorem 2.6, see Appendix B.

2.3 Modeling

We describe the model that generates parse tree components and the method to piece them together.

2.3.1 Modeling mechanism

As illustrated in Figure 1, an encoder computes the hidden states of a given utterance, then an AR decoder consumes the encoder hidden states and generate TreePiece units autoregressively. The technique in Subsection 2.3.2 will allow us to put these units together and obtain a full skeleton. The skeleton then uniquely determines the number (denoted by $N$) and positions of all utterance leaves (see Figure 2), which offers us the convenience to use an NAR decoder to generate all utterance leaves within one step. For NAR utterance decoder, we closely follow (Ghazvininejad et al., 2019): first
| Baselines | \(\text{EM} \quad \text{EM-S} \quad \text{CPU Decoding time (ms)} \quad \text{CPU Inference time (ms)} \quad \text{GPU Decoding time (ms)} \quad \text{GPU Inference time (ms)} | |
|---|---|---|---|---|---|---|
| AR | 86.99 | 89.13 | 45.53 | 63.81 | 44.87 | 55.77 |
| NAR | 86.29 | 88.56 | 7.00 | 25.21 | 6.42 | 17.56 |

| Our method | \(\text{EM} \quad \text{EM-S} \quad \text{CPU Decoding time (ms)} \quad \text{CPU Inference time (ms)} \quad \text{GPU Decoding time (ms)} \quad \text{GPU Inference time (ms)} | |
|---|---|---|---|---|---|---|
| TreePiece-1200 | 86.51 | 89.05 | 7.61 | 25.89 | 6.66 | 17.75 |
| TreePiece-800 | 86.73 | 89.13 | 7.79 | 26.04 | 7.09 | 18.12 |
| TreePiece-600 | 86.86 | 89.26 | 7.85 | 26.14 | 7.34 | 18.45 |
| TreePiece-400 | 86.65 | 89.10 | 7.94 | 26.53 | 7.81 | 19.84 |
| TreePiece-300 | 86.47 | 89.04 | 8.09 | 26.79 | 7.94 | 19.83 |
| TreePiece-200 | 86.51 | 89.02 | 8.09 | 26.57 | 8.19 | 20.18 |
| TreePiece-100 | 86.57 | 89.05 | 8.51 | 27.19 | 8.65 | 20.60 |
| TreePiece-0 | 86.31 | 88.56 | 11.57 | 29.73 | 12.01 | 24.02 |

Table 1: Quality and latency of all models on TOPv2. We train each model with 3 random seeds, and report the averaged EM/EM-S scores and latency on test split of TOPv2 dataset. We measure the decoding and overall inference latency of all models on both CPU and NVIDIA V100 32 GB GPU, and report the averaged milliseconds over all test samples. The number suffix for a TreePiece model represents the expansion size when creating TreePiece vocabulary. The best entry for each metric is bolded.

prepare the NAR decoder’s input by concatenating the embeddings of the predicted TreePieces and \(N\) mask tokens. Then each decoder layer performs self-attention as well as cross attention with encoder hidden states. Lastly the decoder generates utterance predictions at these \(N\) masked positions.

2.3.2 Assemble TreePiece units

Unlike subword-tokenization, where original sentence can be trivially recovered from subword units via string concatenation, there is no canonical way to reassemble TreePiece units. To overcome this issue, we allow TreePiece units to have placeholders\(^2\), and require that two units can only be joined at a placeholder node. This design provides a unique way to glue a sequence of ordered (e.g. pre/level-ordered) TreePiece units, as shown in Figure 2.

3 Experiments

3.1 Datasets

We train, validate, and test our approach on the publicly available benchmark TOPv2 (Chen et al., 2020), a multi-domain task-oriented semantic parsing dataset. The dataset provides a training/validation/test split. Throughout our experiments, we use the training split to train the models, the validation split for earlystopping, model checkpointing, and hyperparameter tuning, and the test split to report the best model’s performance.

3.2 Metrics

We evaluate the model performance on two metrics: Exact Match (EM) respectively Exact Match of Skeleton (EM-S), defined to be the percentage of utterances whose logical forms respectively skeletons are correctly predicted (Shrivastava et al., 2022).

\[^2\]Finding all possible placeholder patterns is NP-hard and unnecessary. In Appendix D we provide a practical solution.
3.3 Baselines
We compare our approach against 2 baselines: AR and NAR. Both baselines are sequence-to-sequence (seq2seq) that produces subword units of serialized logical forms. Their output space consists of ontologies (prefixed by left bracket “[”), utterance leaves\(^3\), and right bracket “]”.

**AR baseline** admits a standard AR structure. It has an autoregressive decoder that generates serialized logical forms by producing one token at a time.

**NAR baseline** adopts mask-predict (Ghazvininejad et al., 2019) with beam size 1, which predicts the output length first and then generates all tokens in one step using a non-autoregressive decoder.

3.4 Experiment setup
For TreePiece models, we experiment with 9 different expansion sizes (used in Stage 1, Subsection 2.2.1) varying from 0 to 1200. We optimize the hyperparameters for both baselines and TreePiece-600, and apply the same hyperparameters from TreePiece-600 to all other TreePiece models. We defer the model configurations, training details, hyperparameter choices to Appendix E.

4 Results
4.1 Quality
As shown in Table 1, TreePiece model sees up to 0.7% relative improvements over NAR and less than 0.15% degradation from AR in terms of EM, while achieving the best EM-S score among all approaches, especially showing 0.8% relative improvement over NAR. We attribute TreePiece’s high quality on skeleton predictions to its ability to respect the tree structure of logical forms and generating 100% valid outputs by design so that the model can better focus on utterance-understanding without being distracted by any structure issue.

Table 2 further shows TreePiece’s privilege over NAR in handling tasks of higher complexity, achieving > 2% improvements for frames with more than 1 intents.

4.2 Latency
Table 1 indicates that TreePiece makes decoding 6.1\times 5.8 times faster and overall inference 3.0\times 2.5 times faster than AR on GPU\textbackslash CPU, with only 5% inference latency growth compared to NAR. In

\[^3\text{We represent utterance leaves in span-pointers (Shrivastava et al., 2021) form to simplify the parsing task.}\]

<table>
<thead>
<tr>
<th>#Intents</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>EM</td>
<td>88.10</td>
<td>83.21</td>
<td>69.46</td>
<td>46.51</td>
</tr>
<tr>
<td>EM-S</td>
<td>90.01</td>
<td>87.23</td>
<td>72.73</td>
<td>50.00</td>
</tr>
<tr>
<td>NAR EM</td>
<td>87.72</td>
<td>80.95</td>
<td>67.60</td>
<td>23.26</td>
</tr>
<tr>
<td>NAR EM-S</td>
<td>89.65</td>
<td>84.81</td>
<td>69.93</td>
<td>24.42</td>
</tr>
</tbody>
</table>

Table 2: Comparison on TOPv2 tasks with different level of complexity in terms of number of intents.

Figure 3, we compare the decoding steps needed to generate full skeletons between TreePiece decoder (of 7 different expansion sizes) and AR decoder. The plot illustrates the acceleration effects of our approach, showing that TreePiece with just 200 expansion-size can already reduce the averaged decoding steps by 83.3% compared to AR.

Related work Autoregressive modeling have been used in a range of Semantic Parsing works (Tai et al., 2015; Cheng et al., 2017b; Dong and Lapata, 2018). Especially, the Sequence-to-Tree scheme was adopted by (Dong and Lapata, 2016). To speed up the inference time, Non-autoregressive modeling were introduced to the field of Machine Translation (Gu et al., 2017; Lee et al., 2018; Libovický and Helcl, 2018), and later become popular in Semantic Parsing as well (Ghazvininejad et al., 2019; Babu et al., 2021; Shrivastava et al., 2021). However, to match the quality of AR, extra training stages are necessary such as Knowledge Distillation from AR models (Gu et al., 2017; Lee et al., 2018; Wei et al., 2019; Stern et al., 2019). On the other hand, (Rubin and Berant, 2020) improves AR decoding’s efficiency via Bottom-Up Parsing (Cheng et al., 2017a). Our paper takes a completely different path from all previous work by extending the subword tokenization algorithms (Nagata, 1994b; Scott, 2002; Sennrich et al., 2015; Kudo, 2018; Kudo and Richardson, 2018) to trees.

Conclusion
This paper proposes a novel way to model and speed up Semantic Parsing via tokenizing parse trees into subtrees. We provide thorough elucidations and theoretical supports for our technique, and demonstrate significant improvements in terms of speed and quality over common AR and NAR baselines on the TOPv2 benchmark.
Limitations

The proposed TreePiece technique, while evaluated on TOPv2 dataset, is not intrinsically bound to it. Indeed, our approach requires only two conditions on a dataset for applicability:

- offers a closed vocabulary of ontologies;
- logical forms inherently carry tree structures.

As a matter of fact, TreePiece can seamlessly adapt to a broad range of datasets, including WikiSQL (Zhong et al., 2017), WEBQUESTIONS (Berant et al., 2013), SequentialQA (Iyyer et al., 2017), GEOquery (Davis and Meltzer, 2007), Spider (Yu et al., 2018), ATIS (Hemphill et al., 1990), etc. Despite this, we solely focused on showcasing its effectiveness in the specific case of "task-oriented natural language understanding based on intent and slots". Additionally, our approach employs standard autoregressive decoding for subtree generation, neglecting the exploration of even more efficient decoding techniques. Lastly, our current tokenization algorithm may introduce out-of-vocabulary (OOV) tokens; while we proposed effective ways to reduce OOV rates, it however cannot fully eliminate the OOV phenomena.

Ethics Statement

Our proposed method presents a novel tokenization operation for tree-like data, yielding substantial practical implications in semantic parsing domains such as natural language understanding, SQL generation, code generation, etc. However, it is crucial to acknowledge that, similar to many other tokenization algorithms, our approach may introduce biases from the training data into vocabulary and tokenization patterns. Consequently, practitioners need to be mindful of this when curating the training corpus before utilizing our method.

References


Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. ArXiv, abs/1601.01280.


Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015. Neural machine translation of rare words with subword units. ArXiv, abs/1508.07909.


Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In NIPS.


Tao Yu, Rui Zhang, Kai-Chou Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Z Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. ArXiv, abs/1809.08887.


A Appendix: Vocabulary generation

This stage resembles the merging operation in Byte Pair Encoding (BPE) (Gage, 1994; Sennrich et al., 2015). Given a training corpus, denote its skeletons by $\mathcal{S}$. We initialize the TreePiece vocabulary $V$ as the set of ontologies extracted from $\mathcal{S}$ and $\mathcal{F}_0$ as the map between ontologies and their frequencies in $\mathcal{S}$. Now repeat the steps below until $V$ reaches a pre-determined size:

- Count the frequencies of all adjacent but unmerged TreePiece unit pairs in $\mathcal{S}$. Find the most frequent pair $p^*$ and its frequency $n^*$.
- Merge $p^*$ in every $S \in \mathcal{S}$ that contains $p^*$, add $p^*$ to $V$, and update $\mathcal{F}_0$ with $\mathcal{F}_0(p^*) = n^*$.

B Appendix: Proof of Theorem 2.6

For convenience we adopt the following notations.

**Notation B.1.** Let $p$ be a TreePiece simplex and $\pi =: [\tau_1, \ldots, \tau_k]$ be a partition where each $\tau_i$ is a TreePiece unit. Define $\mathbf{p}(\pi) := \prod_{\tau \in \pi} \mathbf{p}(\tau)$.

**Notation B.2.** Let $\pi =: [\tau_1, \ldots, \tau_k]$ be a partition and $\tau$ be any TreePiece unit. Define $n(\pi, \tau) := \sum_{\tau_i \in \pi} \mathbbm{1}_{\tau = \tau_i}$. In other words, $n(\pi, \tau)$ is the number of appearances of $\tau$ in $\pi$.

Now we introduce a general hypothesis and will prove a key lemma under this hypothesis.

**Hypothesis B.1.** The joint distribution of skeleton $S$ and partition $\pi$ satisfies the following rule,

$$\mathbb{P}(S, \pi; \mathbf{p}) \propto \prod_{\tau \in \pi} \mathbf{p}(\tau) \cdot \chi(\pi, \mathbf{p}) \text{ if } \pi \in \Pi_S$$

$$0 \text{ otherwise,}$$

(4)

where $\chi : \Pi_S \times [0, 1]^{|V|} \to \{0, 1\}$ is locally smooth almost everywhere (under Lebesgue measure on $[0, 1]$). In other words, for a.e. $\mathbf{p} \in [0, 1]^{|V|}$ and every $\pi \in \Pi_S$ there exists a neighborhood $B_\epsilon(\mathbf{p})$ where $\chi(\pi, \cdot)$ is constant.

**Remark B.1.** Assumption (3) is a special case of hypothesis B.1, where $\chi(\pi, \mathbf{p}) = 1$ if $\pi = \pi_S(\mathbf{p})$ and 0 otherwise.

**Remark B.2.** Without loss of generality, in equations (3) and (4) we replace the symbol “$\propto$” with “$=$”, which otherwise complicates all formulae expressions with a non-essential scalar constant.

**Lemma B.1.** Under Hypothesis B.1, $\mathbf{p}_{k+1}$ is a solution to (1) if the following holds $\forall \tau^* \in V$:

$$\mathbf{p}_{k+1}(\tau^*) = \frac{\sum_{S \in \mathcal{S}} \mathbb{E}_{\Pi_S}[n(\pi, \tau^*)|S; \mathbf{p}_k]}{\sum_{\tau \in V} \sum_{S \in \mathcal{S}} \mathbb{E}_{\Pi_S}[n(\pi, \tau)|S; \mathbf{p}_k]}.$$  

(5)

proof of Theorem 2.6 assuming Lemma B.1 holds. By following the $E$-step of Algorithm 2, we can express the frequency $F^*(t)$ as

$$\sum_{S \in \mathcal{S}} \sum_{\pi \in \pi_S(p_t)} 1_{\tau = \pi} = \sum_{S \in \mathcal{S}} n(\pi_S(p_t), t).$$  

(6)

Assumption 3 says that the probability measure $\mathbb{P}(\pi|S; \mathbf{p}_k)$ is supported on the singleton $\pi_S(p_t)$, therefore the following holds for all $\tau \in V$:

$$n(\pi_S(p_t), \tau) = \mathbb{E}_{\Pi_S}[n(\pi, \tau)|S; \mathbf{p}_k].$$  

(7)

Now inserting the identity (7) to the right hand side of equation (6) we obtain

$$F^*(t) = \sum_{S \in \mathcal{S}} \mathbb{E}_{\Pi_S}[n(\pi, \tau)|S; \mathbf{p}_k].$$  

(8)

Next, Inserting (8) to the $M$-step in Algorithm 2, we have

$$\mathbf{p}_{k+1}(t) = \frac{\sum_{S \in \mathcal{S}} \mathbb{E}_{\Pi_S}[n(\pi, t)|S; \mathbf{p}_k]}{\sum_{\tau \in V} \sum_{S \in \mathcal{S}} \mathbb{E}_{\Pi_S}[n(\pi, \tau)|S; \mathbf{p}_k]}.$$  

(9)

Invoking Lemma B.1, we see that $\mathbf{p}_{k+1}$ is the solution to problem (1), which proves the first conclusion in Theorem 2.6.

Secondly, the monotonicity of $\log \mathbb{P}(S; \mathbf{p})$ can
be achieved as follows:
\[
\sum_{S \in \mathcal{S}} \log \mathbb{P}(S; p_{k+1}) = \sum_{S \in \mathcal{S}} \log \mathbb{P}(S, \pi_S(p_{k+1}); p_{k+1}) \\
\geq \sum_{S \in \mathcal{S}} p_{k+1}(\pi_S(p_{k+1})) = \sum_{S \in \mathcal{S}} \log \mathbb{P}(S, \pi_S(p_{k+1}); p_{k+1}) \\
= \sum_{S \in \mathcal{S}} \log \mathbb{P}(S, \pi_S(p_{k+1}); p_{k+1}) \\
= \sum_{S \in \mathcal{S}} \mathbb{E}[\log \mathbb{P}(S, \pi_S(p_k); p_k)|S; p_k] \\
= \sum_{S \in \mathcal{S}} \mathbb{E}[\log \mathbb{P}(S, \pi_S(p_k); p_k)] \\
= \sum_{S \in \mathcal{S}} \log \mathbb{P}(S; p_k).
\]

Here, all equalities are consequences of Assumption 3, the first inequality follows from the definition of \( \pi_S(p_{k+1}) \), and the second inequality uses the maximization property of \( p_{k+1} \):
\[
p_{k+1} = \arg \max_p \sum_{S \in \mathcal{S}} \mathbb{E}[\log \mathbb{P}(S, \pi_S(p_k); p_k)|S; p_k].
\]

(11)

Thus concludes Theorem 2.6.

To complete the proof of Theorem 2.6 it suffices to prove Lemma B.1.

**proof of Lemma B.1.** Consider the following Lagrange multiplier of problem (1):
\[
\mathcal{L}(p, \lambda) = \sum_{S \in \mathcal{S}} \mathbb{E}[\log \mathbb{P}(S, \pi; p)|S; p_k] \\
+ \lambda(\sum_{\tau \in \mathcal{V}} p(\tau) - 1).
\]

(12)

Plugging (4) into the above equation, we get
\[
\sum_{S \in \mathcal{S}} \sum_{\tau \in \mathcal{S}} \log p(\pi) \cdot \mathbb{P}(\pi|S; p_k) \\
+ \sum_{S \in \mathcal{S}} \sum_{\tau \in \mathcal{S}} \log \chi(\pi, p) \cdot \mathbb{P}(\pi|S; p_k) \\
+ \lambda(\sum_{\tau \in \mathcal{V}} p(\tau) - 1) \\
:= \text{I} + \text{II} + \text{III}.
\]

Inserting equation (13) to the following identity:
\[
\nabla_{p, \lambda} \mathcal{L} = 0,
\]

(14)

we obtain for each \( \tau^* \in \mathcal{V} \) that
\[
\sum_{S \in \mathcal{S}} \sum_{\pi \in \mathcal{S}_S} \frac{n(\pi, \tau^*)}{p(\tau^*)} \cdot \mathbb{P}(\pi|S; p_k) + \lambda = 0.
\]

(15)

Note the locally constant assumption in Hypothesis B.1 makes the derivative of term II vanishes a.e.. Identity (15) then allows us to solve for \( p(\tau^*) \):
\[
p(\tau^*) = - \frac{1}{\lambda} \sum_{S \in \mathcal{S}} \sum_{\pi \in \mathcal{S}_S} n(\pi, \tau^*) \cdot \mathbb{P}(\pi|S; p_k).
\]

(16)

Next, using the simplex property \( \sum_{\tau \in \mathcal{V}} p(\tau) = 1 \) and summing up (16) over \( \mathcal{V} \), we find:
\[
\frac{1}{\sum_{\tau \in \mathcal{V}} \sum_{S \in \mathcal{S}} \sum_{\pi \in \mathcal{S}_S} n(\pi, \tau) \cdot \mathbb{P}(\pi|S; p_k)} \sum_{\tau \in \mathcal{V}} \sum_{S \in \mathcal{S}} \sum_{\pi \in \mathcal{S}_S} n(\pi, \tau) \cdot \mathbb{P}(\pi|S; p_k).
\]

(17)

Plugging the above value of \( \lambda \) back to (16), we obtain the final expression of \( p(\tau^*) \):
\[
\sum_{\tau \in \mathcal{V}} \sum_{S \in \mathcal{S}} \sum_{\pi \in \mathcal{S}_S} n(\pi, \tau^*) \cdot \mathbb{P}(\pi|S; p_k) \\
= \sum_{\tau \in \mathcal{V}} \sum_{S \in \mathcal{S}} \mathbb{E}[n(\pi, \tau^*)|S; p_k]
\]

(18)

which is precisely (5). This proves the if direction of the Lemma. Indeed, a maximizer \( p \) must be a critical point of the Lagrange multiplier and satisfy (14), therefore identity (18) holds. Conversely, identity (18) for arbitrary \( \tau^* \in \mathcal{V} \) fully characterize \( p \), and by the if direction it can only be the unique maximum. This proves the opposite direction, and completes the proof of Lemma B.1.

\[ \square \]

C Appendix: An FFBS algorithm

We propose Algorithm 3, a Forward-Filtering Backward-Sampling (FFBS) (Scott, 2002; Kudo, 2018; Kudo and Richardson, 2018) algorithm under the setting of TreePiece. We highlight those lines in Algorithm 3 that differ from Algorithm 1. Their main distinctions lie in (1) update formula for probabilities, (2) backward strategy.

Before forward, we call GetInitPairProbs to initialize a probability function on the Cartesian product space \( T \times T \cup \{ \text{BOS} \} \) as follows:
\[
\mathcal{P}(t, t') = \begin{cases} 
    p(t) & \text{if } t \in \mathcal{V} \text{ and } t' = \text{BOS token}, \\
    0 & \text{otherwise}.
\end{cases}
\]
During **backward**, we call **Sampling** to randomly sample a previous subtree of $t$ with respect to the following distribution:

$$\left\{ \frac{\exp(\theta \cdot \log \mathcal{P}(t', t))}{\sum_{s \in \mathcal{T}(t')} \exp(\theta \cdot \log \mathcal{P}(s, t))} \right\}_{t' \in \mathcal{T}(t)}$$  \hspace{1cm} (19)

where $\mathcal{T}(t) = \{ t' \in \mathcal{T} : \mathcal{P}(t, t') > 0 \}$. Here a smaller $\theta$ leads to a more uniform sampling distribution among all partitions, while a larger $\theta$ tends to select the Viterbi partition picked by Algorithm 1 (Kudo, 2018).

**Algorithm 3** Forward-Filtering Backward Sampling (FFBS) Algorithm

**Input:** TreePiece vocabulary $\mathcal{V}$, TreePiece simplex $p$, skeleton $S$, sampling coefficient $\theta$.

**Output:** Partition $\pi_S(p)$ and probability $\mathbb{P}(S; p)$.

1. $\mathcal{T} \leftarrow$ All subtrees of $S$ with the same root.
2. $\mathcal{L} \leftarrow \log(p)$, $Q \leftarrow \exp(\mathcal{L})$
3. $d_{\max} \leftarrow$ Depth of $S$
4. for $d = 1, 2, \cdots, d_{\max}$ do // Forward begins
5.  for $t \in \mathcal{T}_d$ do
6.    for $d' = 1, 2, \cdots, d$ do
7.      for $t' \in \text{Filter}(\mathcal{T}_{d'}, t)$ do
8.        $\Delta^* \leftarrow t'\Delta t$
9.        $Q^* \leftarrow Q(t') \cdot \prod_{\tau \in \Delta^*} p(\tau)$
10.       $Q(t) \leftarrow Q(t) + Q^*$
11.     $\mathcal{P}(t, t') \leftarrow Q^*$
12.   end
13. $\mathbb{P}(S; p) \leftarrow Q(S)$ // Forward ends
14. $t_{\text{curr}} \leftarrow S$, $\pi_S(p) \leftarrow 0$ // Backward begins
15. while $t_{\text{curr}} \neq \text{BOS}$ token do
16.  $t_{\text{prev}} \leftarrow \text{Sampling}(\mathcal{P}, t_{\text{curr}}, \theta)$
17.  $\Delta^* \leftarrow t_{\text{prev}} \Delta t_{\text{curr}}$
18.  $\pi_S(p) \leftarrow \pi_S(p) \cup \Delta^*$, $t_{\text{curr}} \leftarrow t_{\text{prev}}$
19. $\pi_S(p) \leftarrow \pi_S(p) \cup \{ t_{\text{curr}} \}$ // Backward ends
20. return $\pi_S(p), \mathbb{P}(S; p)$

Algorithm 3 allows us to sample from all possible partitions rather than generating fixed patterns. In practice, this version is used in place of Algorithm 1 to reduce the OOV rates; see Appendix D for further discussions.

**Remark C.1.** Let us assume the following holds in place of Assumption (3):

$$\mathbb{P}(S, \pi; p) \propto \left\{ \prod_{\tau \in \pi} p(\tau) \right\}$$  if $\pi \in \Pi_S$

$$\left\{ 0 \right\}$$  otherwise,  \hspace{1cm} (20)

another special case of Hypothesis B.1 with $\chi(\pi, p) \equiv 1$. By Lemma B.1, solving problem (1) requires computing $\mathbb{E}_{H_S}[n(\pi, \tau)|S; p_k]$, which now becomes NP-hard. But we can utilize Algorithm 3 to obtain an approximate solution. Indeed, if we iteratively run Algorithm 3 in place of the E-step in Algorithm 1 $K$ times to obtain a partition sequence $\pi_S(p)(1), \pi_S(p)(2), \cdots, \pi_S(p)(K)$, and use the averaged partitions to update the frequency $\mathcal{F}^*$, then following similar lines in Appendix B, we can prove an asymptotic version of Theorem 2.6 under Assumption 20, by showing that the averaged frequency over $K$ partitions converges to $\mathbb{E}[n(\pi, \tau)|S; p_k]$ as $K$ tends to infinity, a direct consequence of Law of Large Numbers. We omit the details.

### D Appendix

As discussed in Subsection 2.3.2, a placeholder structure is necessary for well-defined assembly of TreePiece units. However, adding all possible placeholder patterns to vocabulary is impractical for both time and memory. Instead, we shall include only those patterns that most likely occur. To do so, we tokenize every training skeleton and add the results to the TreePiece vocabulary. As illustrated by the “Tokenize” direction in Figure 2, when a node loses a child during tokenization, we attach a placeholder to the missing child’s position.

**Remark D.1.** There may exist new placeholder patterns that are Out-Of-Vocabulary (OOV) at inference time. To mitigate OOV, we apply Algorithm 3 (in place of Algorithm 1) to tokenize each training skeleton $K_0$ times. Both $K_0$ and the sampling coefficient $\theta_0$ will be specified in Appendix E.1.2. Intuitively, with a larger $K_0$ and a smaller $\theta_0$, Algorithm 3 is able to generate more abundant placeholder patterns to cover as many OOV placeholders as possible.

### E Appendix

#### E.1 Model configurations

##### E.1.1 Model architectures

Across all of our experiments, we use RoBERTa-base encoder (Liu et al., 2019) and transformer (Vaswani et al., 2017) decoders. For encoder architecture, we refer the readers to (Liu et al., 2019). All models’ decoder have the same multi-head-Transformer-layer architecture (embedding dimension 768, 12 heads). Both AR and NAR’s decoders have 2 layers. For TreePiece architecture, TreePiece decoder and Utterance decoder has 1 layer each.
Table 3: Optimization hyperparameter choices for all models

<table>
<thead>
<tr>
<th>Modules</th>
<th>TreePiece model</th>
<th>NAR baseline</th>
<th>AR baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rates</td>
<td>Encoder TreePiece decoder Utterance decoder</td>
<td>Encoder Length predictor Decoder</td>
<td>Encoder Decoder</td>
</tr>
<tr>
<td></td>
<td>$4 \times 10^{-6}$ $6 \times 10^{-5}$ $4 \times 10^{-5}$</td>
<td>$2 \times 10^{-5}$ $1 \times 10^{-4}$ $6 \times 10^{-5}$</td>
<td>$4 \times 10^{-6}$ $2 \times 10^{-5}$</td>
</tr>
<tr>
<td>Decay coefficients</td>
<td>0.999</td>
<td>0.999</td>
<td>0.9999</td>
</tr>
</tbody>
</table>

Figure 4: Serializing the skeleton in Figure 2 (LHS) using the “placeholder nest” design.

Note for fairness of comparisons, we let each model have exactly 2 decoder layers in total.

E.1.2 TreePiece vocabulary

We extract from the TOPv2 dataset 162 ontologies in total, and use TOPv2 training split as training corpus to iteratively run vocabulary generation (ref. Appendix A) 0, 100, 200, 300, 400, 500, 600 times to obtain vocabularies of size 162, 262, 362, 462, 562, 762 respectively. Since we only optimize for TreePiece-600 let us only focus on its vocabulary (of size 762). We then apply the Algorithm 2 with $N_0 = 30$ and $c_0 = 0.01$ to train the tokenizer and arrive at a vocabulary of 2153 different TreePiece unit patterns with placeholders. Finally, we follow Appendix D (with $K_0 = 10, d_0 = 0.15$) and expand the TreePiece vocabulary to size 3817. Note the vocabulary obtained this way has less than 0.1% OOV rate on test dataset, compared to 0.45% were we not using the sampling trick in Remark D.1.

E.2 Training details

E.2.1 TreePiece embedding

Within the TreePiece decoder, we tie the classifier head’s weight to the TreePiece unit embedding matrix, and found it beneficial to pretrain this weight rather than randomly initializing it. We take inspirations from (Shrivastava et al., 2022) and create the pretraining corpus by serializing all skeletons in the training dataset. To let the placeholder information blend into the corpus, we introduce a placeholder nest structure and add it to the serialized logical forms, as illustrated by Figure 4. Finally, we use the masked language model (MLM) (Devlin et al., 2019) pre-training objective with mask-rate 0.15 and train for up to 20 epochs until convergence.

E.2.2 Hyperparameter choices

Across all experiments, we set the batch size to be 256, and total number of epochs to be 100 with early stopping when validation EM (ref. Subsection 3.2) stops improving. For optimization, we use Adam optimizer (Kingma and Ba, 2014) with weight decay 0.01 and $\epsilon = 10^{-8}$. In addition, we warm up the learning rate in 5 epochs and then exponentially decay (Senior et al., 2013) at the end of every epoch.

We also observe that each module favors learning rates with different magnitude, so we do learning rate search separately for each module among the interval $[10^{-6}, 10^{-4}]$. For exponential decay coefficients we optimize them among $\{0.9, 0.99, 0.999, 0.9999\}$. Table 3 summarizes our final choices of these hyperparameters.