VIETNAM NATIONAL UNIVERSITY HOCHIMINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY

RESEARCH PROPOSAL
UNDERGRADUATE RESEARCH REPORT

RESEARCH TOPIC: AUTOREGRESSIVE GRAPH GENERATION APPROACH FOR DEPENDENCY PARSING

INSTRUCTOR
PhD. QUY THI NGUYEN

STUDENT
NHAT VIET NGUYEN- 21520378

1 Summary

Dependency Parsing is a fundamental problem in the field of Natural Language Processing [20], with broad applications in tasks such as Machine Translation, Classification, and Knowledge System Construction [15, 11, 35, 34]. This problem is crucial because the dependency tree it generates encapsulates valuable information about the relationships between elements within a sentence. Traditional approaches to dependency parsing are generally divided into two separated categories, each with its own strengths and weaknesses: graph-based methods [13, 23] and transition-based [27, 29]. The observation that transition-based and graph-based parsers tend to make significantly different errors [24, 25] indicates potential empirical benefits from combining these two types of parsers. In this research, we aim to integrate these approaches in a novel way through autoregressive graph generation [39]. Specifically, we utilize GFlowNets [2] for dependency parsing.

2 Introduction

Dependency parsing, a fundamental task in Natural Language Processing (NLP), involves analyzing the dependency graph of a sentence. This process aims to create a structure that is easily interpretable for both humans and computers. It establishes relationships between words in terms of *head* words and *dependent* words, where each word is connected by an arc indicating how one word modifies another. These arcs, known as links, are crucial for understanding the syntactic and semantic relationships within sentences. The relationships are represented in the form of a dependency graph, where nodes correspond to words and directed edges represent syntactic dependencies.

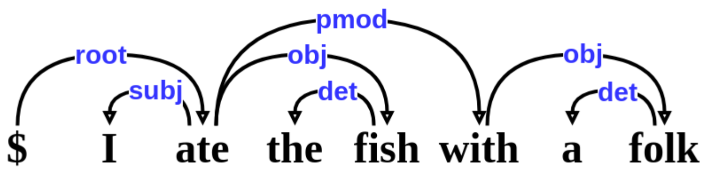

Figure 2.1: Illustration of a dependency graph for the sentence "*I ate the fish with a fork*". The verb "*ate*" is the root, with "*I*" as the subject (subj), "*fish*" as the object (obj), and "*with a fork*" as a prepositional modifier (pmod).

The significance of dependency parsing lies in its ability to capture the syntactic structure of a sentence, providing valuable insights into the relationships between different components. This structural information is crucial for various downstream NLP tasks, such as machine translation, information extraction, text summarization, and sentiment analysis [15, 11, 35, 34, 36]. Machine translation, for example, understanding the syntactic dependencies between words can lead to more accurate translations by preserving the intended meaning and grammatical structure of the source language.

Traditional approaches to dependency parsing can be broadly categorized into two main types: transition-based methods and graph-based methods. Transition-based parsers [27, 29] build dependency trees incrementally, making a series of decisions to construct the tree, while graph-based parsers score and select the highest-scoring tree from all possible trees for a given sentence. Each of these approaches has its own strengths and weaknesses. Transition-based methods are typically faster and can handle non-projective dependencies more easily, but they may suffer from error propagation. On the other hand, graph-based methods [13, 23, 19] can leverage global information to produce more accurate trees but at the cost of increased computational complexity.

Empirically, both strategies perform roughly equally well in identifying most relevant patterns. They exhibit complementary strengths and weaknesses, which makes combining the two parsers particularly effective [24]. These methods typically take two forms. The first involves creating various graph and transition-based parsers and using a meta-system to select a single parse through majority voting [31], either over entire trees or individual arcs. The second method uses the output of one parser (e.g., a transition-based parser) as additional input to another (e.g., a graph-based parser), known as *stacked classifiers* or *stacked parsers* [33].

In this research, we aim to integrate transition-based and graph-based parsing approaches more effectively by defining dependency parsing as a graph generation problem and addressing it using autoregressive graph generation techniques. Specifically, we utilize GFlowNets [2], a novel learning algorithm that generates graph structures compositionally by adding one edge to the sub-graph at a time. By combining the strengths of both traditional approaches and leveraging the rich information within graphs, we seek to achieve empirical gains in parsing performance and create a new direction for this problem.

The following sections will provide a detailed overview of related work (Section 3) and background (Section 4), describe our methodology (Section 5) in depth, present experiments (Section 6), and draw final conclusion on the progress of the research (Section 7).

3 Related Work

Transition-based Parsing

Transition-based parsers [27, 29] construct dependency structure through a sequence of predicted transition actions, i.e., whether adding an arc or not. An arc $(w_i, r, w_j) \in A$ encodes a dependency between two words, where w_i is the head node, w_j is the dependent and r is the dependency type of relation between w_i and w_j.

Typically, these parsers operate using a transition system *configuration*, denoted as (σ, β, A), consists of a *stack* σ containing processed words, a *buffer* β of words awaiting processing, and a *set of arcs* A representing dependency arcs. Transitions, such as shifting a word onto the stack or forming arcs between stack elements, are applied sequentially until a terminal configuration is reached where all words have been processed and a complete dependency tree is formed.

This approach is computationally efficient, leveraging deterministic state transitions to incrementally build the parse tree. Key algorithms in transition-based parsing include variants of arc-eager [27, 30, 29] and arc-standard [28] methods, each with distinct transition sets governing how arcs are formed and how words are moved between the stack and buffer. Additionally, common mechanisms in autoregressive generators, such as beam search and pointer networks [38, 14], are used to enhance transition-based parsers. In this report, we investigate a method that mimics transition-based behavior while capturing the rich information inherent in graph structures.

Graph-based Parsing

Graph-based parsers [13, 23, 19, 12] approach the task of dependency parsing by treating it as a structured prediction problem, where the goal is to find the highest-scoring dependency graph for a given sentence. These parsers typically use a global scoring function that considers the entire tree structure, rather than making local decisions incrementally. The scoring function is often designed to capture various linguistic features and dependencies, providing a more holistic view of the sentence's syntactic structure. Key techniques in graph-based parsing involve algorithms like the Eisner algorithm [13] and the Chu-Liu/Edmonds algorithm [8], which efficiently search for the optimal tree structure.

During our survey, we noted that [7] proposes a graph-based autoregressive parser that adds arcs sequentially while considering previous parsing decisions. However, it does not extend sub-graph structures and, thus, should not be considered a rigorous graph generative model. Similarly, some heuristic algorithms [18, 21] construct the Maximum Spanning Tree incrementally but predict all edge probabilities at once, and therefore cannot be classified as autoregressive graph generation methods. Therefore, the autoregressive graph generation approach truly offers a novel direction for addressing dependency parsing problems, effectively balancing both transition-based and graph-based methods.

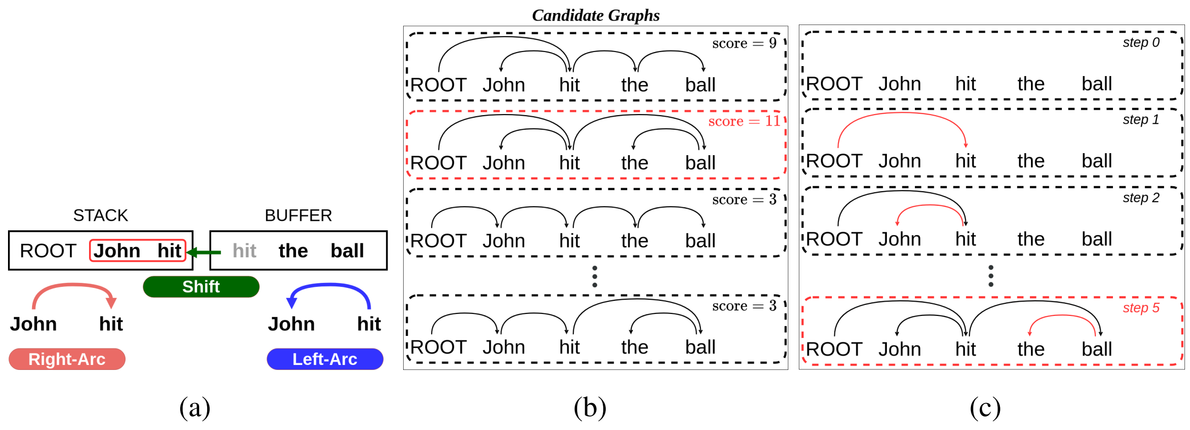

Figure 3.1: Illustrations for (a) Transition-based (b) Graph-based (c) Autoregressive Graph Generation for Dependency Parsing

Autoregressive Graph Generation

Autoregressive Graph Generation is a subfield of deep graph generation focused on learning the probability distribution of graphs, enabling the sampling of new graphs from this distribution. Unlike other methods, autoregressive graph generation constructs a graph through a sequence of steps, typically generating nodes and/or edges consecutively. Autoregressive (AR) models achieve this by factorizing a joint distribution over N random variables using the chain rule of probability. Specifically, the model breaks down the generation process into sequential steps, where each step determines the next action based on the current graph state. The general formulation of AR models is as follows:

$$p(G^{\pi}) = \prod_{i=1}^{N} p(G_i^{\pi} \mid G_1^{\pi}, \cdots, G_{i-1}^{\pi}) = \prod_{i=1}^{N} p(G_i^{\pi} \mid G_{<i}^{\pi}), \tag{3.1}$$

where $G_{<i}^{\pi} = \{G_1^{\pi}, G_2^{\pi}, \cdots, G_{i=1}^{\pi}\}$ represents the set of random variables from the previous steps. Since AR models generate graphs sequentially, applying them requires a pre-specified ordering π of nodes in the graph.

Recently, autoregressive graph generation methods have been applied to various NLP tasks due to their ability to capture complex dependencies and relationships within the data. However, in the context of semantic parsing, autoregressive graph generators are more commonly used for Abstract Meaning Representation (AMR) rather than dependency graphs [41, 4, 40, 5]. This preference arises because dependency parsing presents the challenge of determining the generation order of graph data, which lacks a conventional reading order like text data [6]. Although autoregressive graph generation can capture complex patterns, the absence of a consistent node generation order undermines the model's performance. Some attempts to address this challenge involve sorting nodes by referring to known sequences such as word order [41] or alphanumerical order [3]. However, these methods introduce exposure bias, where the order imposed during training may not match actual dependencies, leading to discrepancies during inference. Additionally, the combinatorial explosion of possible dependency graphs for a given sentence makes sampling for a valid graph nearly intractable for many methods. These challenges hinder the full potential of autoregressive models in dependency parsing. Fortunately, a novel approach for training expressive autoregressive models, GFlowNets [1], introduced by Joshua Bengio in 2021, offers a promising direction for integrating graph-based and transition-based methods in dependency parsing.

4 Background

Notation

Given the source sentence $X = w_1 w_2 \cdots w_n$, and the set of pre-defined dependency relation types $R = \{r_1, \cdots, r_m\}$, a dependency graph $G = (V, A)$ is a labeled directed graph (digraph) in the standard graph theorem. The graph consists of nodes $V = \{ROOT, w_1, w_2, \cdots, w_n\}$ and arcs $A = \{(w_i, r, w_j)\}$ representing all possible dependency relations from head w_i to dependent w_j, labeled with relation type r.

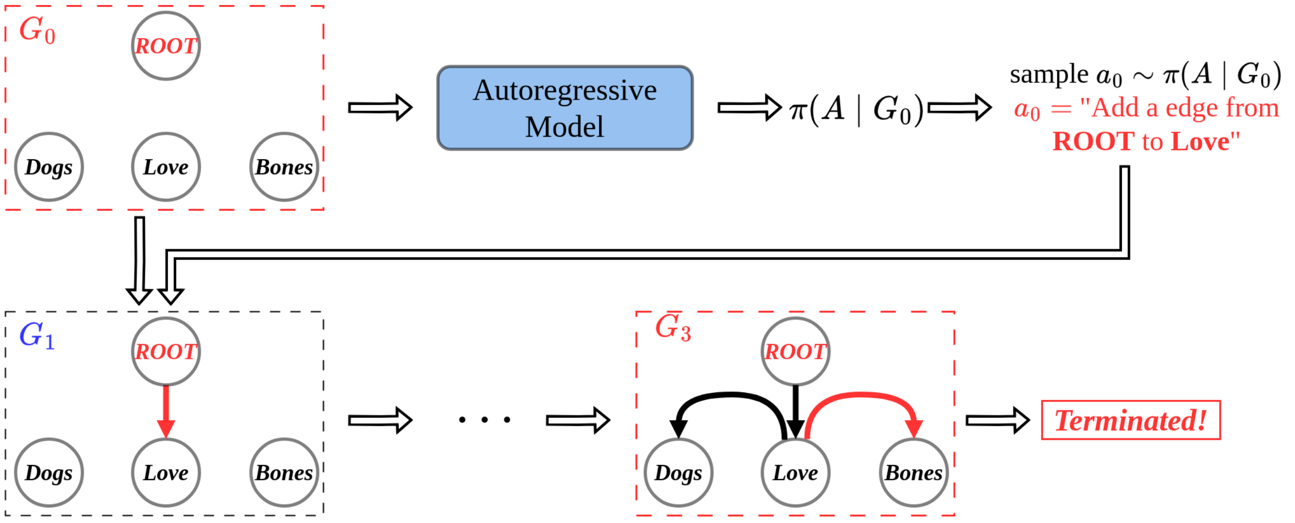

Figure 4.1: Illustration of the autoregressive graph generation process. In this diagram, the AF model takes a graph G_0 as input and determines an action to choose based on its policy. The action then determines the graph G_1 by adding an edge from *ROOT* to *Love* to G_0. This process repeats until obtain a valid dependency graph G_3 for the given sentence.

The task is to generate a dependency graph using autoregressive graph generation, starting from the empty graph G_0, which is a fully disconnected graph over the words in the input sentence X. We use terminologies from reinforcement learning for clarity. A transition $G \to G'$ involves adding an edge to G to form the graph G'; essentially, the graphs are constructed one edge at a time. The set of actions for a transition consists of all possible edges that can be added to G. When a graph G is input into an autoregressive model, it returns a policy $\pi(A \mid G)$, a probability distribution over all possible actions A, that determines the action to take. The sampled action is then executed to transition G to G'. This process repeats until the graph becomes a valid dependency graph, and is called a trajectory. Each valid graph G^*, or possible dependency graph, is associated with a reward $R(G)$ showing model's preference of this graph. Figure 4.1 represents the described process.

Generative Flow Networks

Generative Flow Networks, or GFlowNets, represent a novel class of probabilistic models designed for the generative modeling of discrete and composite objects, such as graphs. The introduction of GFlowNets provides a significant leap forward in the realm of generative modeling, especially when compared to traditional methods like Monte Carlo Markov Chains (MCMC) and variational inference. These conventional approaches often grapple with efficient sampling from high-dimensional and multimodal distributions, particularly when dealing with large combinatorial spaces.

Traditional approaches like MCMC and variational inference often struggle with efficiently exploring such distributions, especially when the sample space is combinatorially large. GFlowNets offer a new perspective by framing the generation of objects as a sequential decision-making process, which allows for the effective exploration and sampling of diverse structures. At the core of GFlowNets is the concept of "flow," which refers to unnormalized probabilities learned by the network to represent the likelihood of reaching different states in the generative process. The flow at any intermediate state is defined as the weighted sum of the non-negative rewards of all terminating states that can be reached from it This ensures a balance where the total incoming flow to a state matches the total outgoing flow, maintaining a consistent probabilistic framework throughout the generative process.

GFlowNets generate samples through a sequence of stochastic steps, constructing a compositional object incrementally. This process allows them to represent rich, multimodal distributions over objects. The training objectives are designed to explore and discover different modes of the target distribution. Unlike MCMC methods that generate samples through long sequences converging to the desired distribution, GFlowNets can generate samples in a single pass. This significantly reduces the computational overhead associated with stochastic searches and mode-mixing issues typical in MCMC methods.

A significant advantage of GFlowNets is their ability to sample objects in a manner proportional to a given reward function. This is akin to converting an energy function $\mathscr{E}(s) = -\log R(s)$ into a sampler, where $R(s)$ is a non-negative reward function corresponding to an unnormalized probability. The key property of GFlowNets is that their sampling policy is trained to ensure the probability of sampling an object s is proportional to $R(s)$. This feature is particularly beneficial for tasks requiring diverse and representative samples from the target distribution, making GFlowNets highly suitable for applications in fields such as Bayesian structure learning, where understanding the distribution over possible models is crucial.

In Bayesian structure learning, for example, GFlowNets provide a method for approximating the posterior distribution over the structure of Bayesian networks. This approach, referred to as DAG-GFlowNet, views the generation of a Directed Acyclic Graph (DAG) as a sequential decision problem, where the graph is constructed edge by edge based on learned transition probabilities. DAG-GFlowNet has shown promise in accurately approximating the posterior distribution over DAGs and has demonstrated superior performance compared to traditional methods, particularly in scenarios involving both observational and interventional data.

Trajectory Balance Condition

Trajectory balance is a concept in the context of Markovian flows and policies that aims to ensure consistency in the likelihood of trajectories traversed in both forward and backward directions. It involves maintaining equilibrium between the probabilities of moving forward and backward along a trajectory while considering the rewards associated with the states visited.

In the trajectory balance framework, a Markovian flow F and the corresponding distribution over complete trajectories are denoted by P. Let P_F and P_B represent the forward and backward policies determined by F. By manipulating the equations, we obtain the trajectory balance constant for any complete trajectory $\tau = (s_0 \to s_1 \to \ldots \to s_n = x)$ as:

$$Z \prod_{t=1}^{n} P_F\left(s_t \mid s_{t-1}\right) = F(x) \prod_{t=1}^{n} P_B\left(s_{t-1} \mid s_t\right) \tag{4.1}$$

where it is noted that $P\left(s_n = x\right) = \frac{F(x)}{Z}$. This trajectory balance constraint, along with the detailed balance constraint, is a special case of a general constraint studied as a training objective.

The trajectory balance constraint is converted into an objective to be optimized along trajectories sampled from a training policy. If a model with parameters θ outputs estimated forward policy $P_F(-|s;\theta)$ and backward policy $P_B(-|s;\theta)$ for states s, along with a globe scalar Z_θ estimating $F(s_0)$, the trajectory loss for a trajectory $\tau = (s_0 \to s_1 \to \ldots \to s_n = x)$ is defined as:

$$\mathscr{L}_{\mathrm{TB}}(\tau) = \left(\log \frac{Z_\theta \prod_{t=1}^{n} P_F\left(s_t \mid s_{t-1};\theta\right)}{R(x) \prod_{t=1}^{n} P_B\left(s_{t-1} \mid s_t;\theta\right)}\right)^2 \tag{4.2}$$

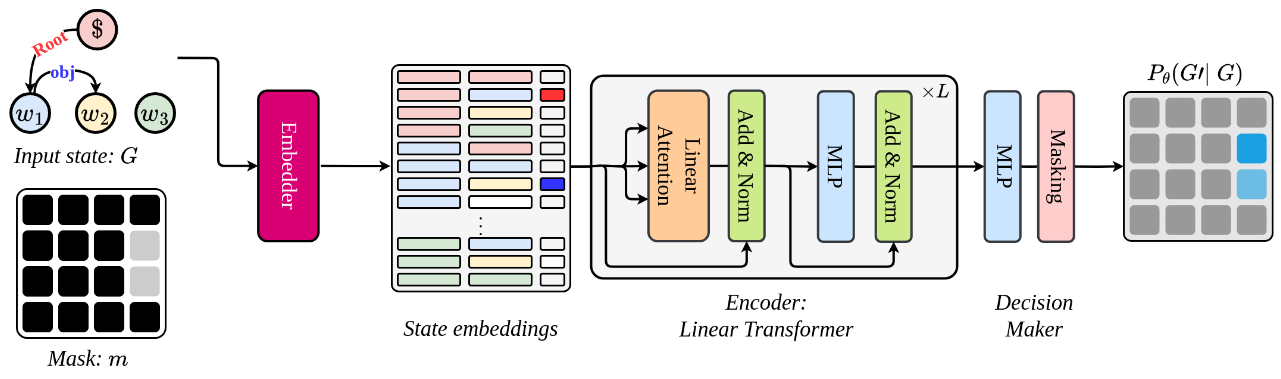

Figure 4.2: Proposed network architecture for the autoregressive model.

The trajectory loss is updated along trajectories sampled from a training policy π_θ, using stochastic gradient descent. The correctness of the algorithm is guaranteed by Proposition 1, which establishes the relationship between the forward and backward policies and the Markovian flow satisfying the trajectory balance constraint. Specifically, Proposition 1 states that if the forward and backward policies, along with the normalizing constant of a Markovian flow, satisfy the trajectory balance constraint, then the trajectory loss is zero for all complete trajectories. Conversely, if the trajectory loss is zero for all complete trajectories, then the corresponding Markovian flow satisfies the trajectory balance constraint and the forward policy samples proportionally to the reward.

In certain scenarios, it may be beneficial to fix the backward policy P_B and train only the parameters related to the forward policy P_F and the normalizing constant Z_θ. For instance, setting $P_B(-|s)$ to be uniform over all the parents of a state s can be a natural choice in cases where constructing a model to output a distribution over parents is challenging, such as in the molecule domain. This choice simplifies the training process, especially in scenarios where modeling the distribution over parents is challenging.

5 Methodology

Our objective in this research is to construct a dependency graph using an autoregressive generation approach. This involves creating a distribution over all possible dependency trees for a given sentence. The challenge arises from the discrete and combinatorially large space of Directed Acyclic Graphs (DAGs). To address this, we employ a GFlowNet, which is well-suited for this task, as it treats the dependency tree as a connected, acyclic graph where each node has only one parent.

1 Autoregressive Model Architecture

To satisfy all the requirements of GFlowNet, it requires the design of a neural network defining a policy function for each input graph. This raises two key questions:

First, how can we represent the graph in a feature form that the model can understand? Second, how can we use these features to map to a probability distribution over the set of actions?

1.1 Graph Representations

Given an input graph G with d nodes, we represent it as a set of d^2 possible edges, including self-loops. Each directed edge is embedded using the embeddings of its source and target nodes, along with an additional vector indicating the edge between these two nodes. In other words, the graph is represented using the set of possible dependency links between every two words w_i and w_j in the sentence X.

The embedding of a dependency link $l_{i,j}$ from w_i to w_j, with label $r_{i,j}$ can be represented as follows:

$$h_i^{(arc\text{-}head)} = MLP^{(arc-head)}(\texttt{PLMs}(w_i)) \tag{5.1}$$

$$h_j^{(arc\text{-}dep)} = MLP^{(arc-dep)}(\texttt{PLMs}(w_j)) \tag{5.2}$$

$$h_{i,j} = \text{Embeddings}(r_{i,j}) \tag{5.3}$$

$$l_{i,j} = h_i^{(arc\text{-}head)} \oplus h_j^{(arc\text{-}dep)} \oplus h_{i,j} \tag{5.4}$$

In this representation, each node in the graph G (or each word in X) is assigned two different roles: the *arc-head* (head) representation and the *arc-dep* (dependent) representation. We obtain them by applying MLPs to the pretrained embeddings for those words from a Large Language Model [10, 9, 32]. The embedding of the relation $r_{i,j}$ between two words is a learned embedding specific to each pre-defined relation. And we denote graph representation as $\mathbf{H} = \{l_{i,j}\}$, with $1 \leq i, j \leq d$. It is worth noting that we construct the graph by sequentially adding one edge at a time while keeping the nodes V fixed. Consequently, we only need to initialize the node embeddings in the graph once, while updating the embeddings of the relations at each transition based on the policy that AR model returns. This comprehensive approach ensures that the graph structure and the dependencies between words are effectively captured and represented, enabling the neural network to understand and process the input graph.

1.2 Policy Creation

For every graph G fed into the autoregressive (AR) model, we need to return the policy $\pi(A \mid G)$ to allow GFlowNet to sample an action and perform a transition to the next graph. For instance, with the initial graph G_0 in 4.1, the model must express the probability distribution over all possible edges that can be added to G_0. Specifically, we will need a backbone that strong enough to capture the complex information of the graph representations $\mathbf{H}$. Neural networks, with their capacity to generalize to unseen graphs, are well-suited for this task.

Our choice of neural network architecture is motivated by multiple factors: we want an architecture (1) that is invariant to the order of the inputs, since G is represented as a set of edges, (2) that transforms a set of input edges into a set of output probabilities for each edge to be added, in order to define the policy $\pi(A \mid G)$, and (3) whose parameters θ do not scale too much with d, since the total number of edges is already d^2. While the Transformer [37] could be a natural choice, its self-attention layers scale as d^4 with the input size d^2, limiting its applicability for larger dependency graphs.

Fortunately, an alternative for Transformer is Linear Transformer [17] does not suffer from quadratic scaling in the input size. This architecture approximates the attention mechanism as follows:

$$Q = \mathbf{H}W_Q \qquad K = \mathbf{H}W_K \qquad V = \mathbf{H}W_V \tag{5.5}$$

$$LinAttn_k(\mathbf{H}) = \frac{\sum_{j=1}^{J} \left(\phi(Q_k)^T \phi(K_j)\right) V_j}{\sum_{j=1}^{J} \phi(Q_k)^T \phi(K_j)}, \tag{5.6}$$

where $\mathbf{H}$ is the graph representations and the input of the k-th linearized attention layer, $\phi(\cdot)$ is a non-linear feature map, J is the size of the input graph $\mathbf{H}$ which is d^2, and Q, K, and V are linear transformations of $\mathbf{H}$ corresponding to the queries, keys, and values respectively, as is standard with Transformers.

The encoded information in the final layers of Linear Transformer backbone is then mapped to the space of edges probabilities of being selected, calculated by:

$$P_\theta(G' \mid G) = \text{softmax}(\text{MLP}(LinAttn_L(\mathbf{H}))) \tag{5.7}$$

where $P_\theta(G' \mid G)$ is the probability distribution over all possible edges to transition graph G to G', i.e., the policy $\pi(A \mid G)$.

Beside, the relationship corresponding to all pairs of word are figured out by Deep Biaffine Attention [12]:

$$r_{i,j} = \text{DeepBiaffine}\left(h_i^{(arc\text{-}head)}, h_j^{(arc\text{-}dep)}\right) \tag{5.8}$$

The generation proceeds via repeating the aforementioned operations until we have a valid dependency graph, i.e., a connected, acyclic, and single-headed graph.

1.3 Mask over actions

To guarantee the integrity of the constructed graph, we have to ensure that adding a new edge to the graph G results in a valid graph G'. That means the new edge (1) must not be already present in G, (2) must not introduce a cycle, and (3) all nodes have only one parent node. To enforce these constraints, we adopt a mask construction method [16] that can filter out invalid edge using the adjacency matrix of G and the adjacency matrix of the transitive closure of G^T, the transpose of G.

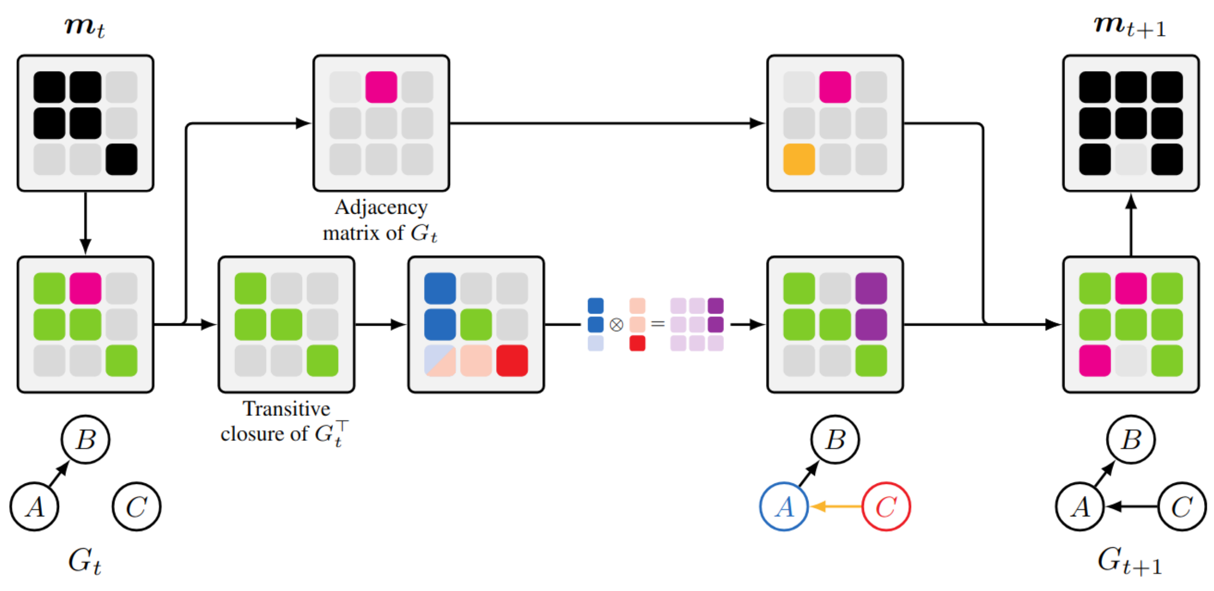

Figure 5.1: Illustration of online mask proposed by [16]

Since the mask can be composed in 2 parts, we can simply update each part whenever a new edge is added:

- **Adjacency matrix**: To update the adjacency matrix, we set the appropriate entry in the matrix when a new edge is added.
- **Transitive closure**: To update the transitive closure of the transpose, we compute the outer product of the column corresponding to the target of the edge with the row corresponding to the source of the edge. This outer product is then added to the initial transitive closure.

Additionally, we can filter out edges that would result in a node having more than one parent by summing the entries in the adjacency matrix. The total operations for these updates can be performed in $O(d^2)$, where d is the number of nodes in the graph G.

2 Reward

The reward function for a sampled graph is designed to encourage similarity to a target dependency graph. With a valid graph G, the reward $R(G)$ is given by:

$$R(G) = \exp^{(1-\text{GED}(G,G_{target}))}, \tag{5.9}$$

where GED(G, G_{target}) represents the normalized graph edit distance [26] between the sampled graph G and the target graph G_{target}. The graph edit distance GED(G, G_{target}) is a value between 0 and 1, indicating the number of edits needed to tranform G into G_{target}. The reward increases as the similarity between the sampled graph and the target graph increases.

3 Algorithms

3.1 Training GFlowNet

In order to train GFlowNet for dependency parsing problem, we closely follow that of a standard GFlowNet, and just do a small change so that it can be trained with dataset $\mathscr{D} = \{(X_i, Y_i)\}_{i=1}^N$ with input sentence X_i and corresponding ground-truth dependency graph Y_i. Specifically, the algorithm is describe in Algorithm 1. The objective is to learn the parameters θ of the forward and backward conditional policies $P_F(- \mid G, X; \theta)$ and $P_B(- \mid G', X; \theta)$, and the log-partition function $\log Z_\theta(X)$ such that probability being sampled of a valid graph G^* is proportional to its reward $P(G^* \mid X) \propto R(G \mid X)$. To this end, we consider an extension of the trajectory balance objective:

$$\mathscr{L}(\tau, X; \theta) = \left(\log \frac{Z_\theta(X) \prod_{G \to G' \in \tau} P_F(G' \mid G, X; \theta)}{R(G^* \mid Y) \prod_{G \to G' \in \tau} P_B(G \mid G', X; \theta)} \right)^2, \tag{5.10}$$

Algorithm 1 Training Conditional GFlowNet

```
Require: Reward function $R : \mathscr{S}^f \to \mathbb{R}_{\geq 0}$, sampling distribution $p(x)$
 1: Initialize: models $P_F, Z$ with parameters $\theta$
 2: repeat
 3:     Sample input sentence and gold tree $(x, y) \sim p(x)$
 4:     Initialize graph embeddings $G_0$
 5:     $z \leftarrow$ Sample trajectory $\tau = (G_0 \to \cdots \to G_n)$ from policy $P_F(-|-, x; \theta)$
 6:     Compute reward $R(z, y)$ and corresponding loss $\mathscr{L}_{TB}(\tau, x; \theta)$
 7:     $\theta \leftarrow \theta - \eta \nabla_\theta \mathscr{L}_{TB}(\tau, x)$
 8: until convergence monitoring on running $\mathscr{L}_{TB}(\tau)$
```

3.2 Sampling from GFlowNet

The algorithm for sampling from GFlowNet is summarized in Algorithm 2.

Algorithm 2 Inference

```
Require: Input sentence and gold tree (x, y), trained parameters $\theta$, beam size $k$
 1: Initialize: models $P_F, Z$ with parameters $\theta$
 2: Initialize state embeddings $s_0$
 3: Initialize an empty list: $z\_list \leftarrow []$
 4: for $i = 1$ to $k$ do
 5:     Sample trajectory $\tau = (s_0 \to \cdots \to s_n)$ from policy $P_F(-|-, x; \theta)$
 6:     $z\_list \leftarrow z\_list \cup z$
 7: end for
 8: return Aggregrate($z\_list$)
```

6 Experiment

Currently, the PyTorch code for the algorithm has been completed[1]*. With reasonably adjusted hyperparameters, the model can handle sentences with up to 47 words on a 15 GB RAM T4 GPU. However, the maximum number of words in a sentence in dependency parsing datasets can be around 160. This implies the need for a GPU with approximately 50 GB of RAM to experiment on all datasets for this task. This computational cost issue arises from the nature of most frameworks like PyTorch or TensorFlow, which are stateful, causing the model's weights to be stored on the GPU for each transition step in the trajectory. The team is currently transitioning to a stateless framework to avoid storing model weights. Therefore, this section will provide an overview of the evaluation criteria and the treebanks used as evaluation sets instead.*

1 Evaluation Metrics

In the field of dependency parsing, evaluating the performance of a model involves several key metrics. These metrics provide insights into different aspects of the model's accuracy and reliability. Below are the primary evaluation metrics used:

1.1 Unlabeled Attachment Score (UAS)

The Unlabeled Attachment Score (UAS) measures the percentage of words that are correctly attached to their heads, regardless of the dependency labels. It focuses on the structure of the dependency tree without considering the specific types of relationships.

$$\text{UAS} = \frac{\text{Number of correctly attached words}}{\text{Total number of words}} \times 100 \tag{6.1}$$

UAS is crucial for understanding how well the model captures the syntactic structure of sentences.

1.2 Labeled Attachment Score (LAS)

The Labeled Attachment Score (LAS) extends UAS by also considering the correctness of the dependency labels assigned to the edges. It measures the percentage of words that are correctly attached to their heads with the correct dependency labels.

$$\text{LAS} = \frac{\text{Number of correctly attached and labeled words}}{\text{Total number of words}} \times 100 \tag{6.2}$$

LAS provides a more comprehensive evaluation by considering both the structure and the semantic relationships.

[1]Source code: https://github.com/nv259/dp_gfn.git

1.3 ROOT Accuracy

ROOT Accuracy evaluates the percentage of sentences in which the root of the dependency tree is correctly identified. This metric is essential because the root represents the main verb or predicate of the sentence, which is crucial for understanding the overall sentence structure.

$$\text{ROOT Acc} = \frac{\text{Number of correctly identified roots}}{\text{Total number of sentences}} \times 100 \tag{6.3}$$

ROOT Accuracy highlights the model's ability to identify the central element of a sentence.

1.4 Complete Match

Complete Match measures the percentage of sentences where the entire dependency tree is correctly predicted, including both the structure and the labels. This is a stringent metric that indicates the model's overall accuracy in generating perfect dependency parses.

$$\text{Complete Match} = \frac{\text{Number of sentences with perfect parses}}{\text{Total number of sentences}} \times 100 \tag{6.4}$$

Complete Match is the most challenging metric, as it requires the model to get every detail of the dependency tree correct.

2 Universal Dependencies (UD) Treebanks

Treebanks play a critical role in the development and evaluation of dependency parsers. They are used for training parsers, they act as the gold labels for evaluating parsers, and they also provide useful information for corpus linguistics studies.

The Universal Dependencies[2] (UD) project is a collaborative effort dedicated to creating consistent syntactic annotations across various languages, which currently has almost 200 dependency treebanks in more than 100 languages [22]. Its primary goal is to develop treebanks that share a common framework for representing dependency relations and part-of-speech tags.

By establishing a unified annotation scheme, UD facilitates cross-lingual NLP research and development. It offers a comprehensive collection of treebanks, encompassing both resource-rich and low-resource languages. This broad coverage enables researchers to compare models across different linguistic domains and to explore transfer learning techniques.

[2]https://universaldependencies.org/

7 Conclusion

In this research, we proposed a new direction for dependency parsing - autoregressive graph generation approach. By balancing the advantages and disadvantages of both transition-based and graph-based, this approach (theoretically) shows great potential in improving the parsers performance. To this end, we are conducting research on GFlowNets, a novel framework for generating graphs in an autoregressive manner, to address the dependency parsing problem. By combining the strengths of both traditional approaches and leveraging the power of deep learning, we seek to achieve empirical gains in parsing performance and open a promising approach for dependency parsing.

References

[1] Emmanuel Bengio et al. “Flow network based generative models for non-iterative diverse candidate generation”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 27381–27394.

[2] Yoshua Bengio et al. “Gflownet foundations”. In: *The Journal of Machine Learning Research* 24.1 (2023), pp. 10006–10060.

[3] Michele Bevilacqua, Rexhina Blloshmi, and Roberto Navigli. “One SPRING to rule them both: Symmetric AMR semantic parsing and generation without a complex pipeline”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 14. 2021, pp. 12564–12573.

[4] Deng Cai and Wai Lam. “Core Semantic First: A Top-down Approach for AMR Parsing”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3799–3809. DOI: 10.18653/v1/D19-1393. URL: https://aclanthology.org/D19-1393.

[5] Deng Cai and Wai Lam. “AMR Parsing via Graph-Sequence Iterative Inference”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 1290–1301. DOI: 10.18653/v1/2020.acl-main.119. URL: https://aclanthology.org/2020.acl-main.119.

[6] Xiaohui Chen et al. “Order matters: Probabilistic modeling of node sequence for graph generation”. In: *arXiv preprint arXiv:2106.06189* (2021).

[7] Hao Cheng et al. “Bi-directional attention with agreement for dependency parsing”. In: *arXiv preprint arXiv:1608.02076* (2016).

[8] Yoeng-Jin Chu. “On the shortest arborescence of a directed graph”. In: *Scientia Sinica* 14 (1965), pp. 1396–1400.

[9] Alexis Conneau et al. “Unsupervised Cross-lingual Representation Learning at Scale”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, July 2020, pp. 8440–8451. DOI: 10.18653/v1/2020.acl-main.747. URL: https://aclanthology.org/2020.acl-main.747.

[10] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. 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)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

[11] Yuan Ding and Martha Palmer. “Synchronous dependency insertion grammars: A grammar formalism for syntax based statistical MT”. In: *Proceedings of the Workshop on Recent Advances in Dependency Grammar*. 2004, pp. 90–97.

[12] Timothy Dozat and Christopher D Manning. “Deep biaffine attention for neural dependency parsing”. In: *arXiv preprint arXiv:1611.01734* (2016).

[13] Jason Eisner. “Three new probabilistic models for dependency parsing: An exploration”. In: *arXiv preprint cmp-lg/9706003* (1997).

[14] Daniel Fernández-González and Carlos Gómez-Rodríguez. “Left-to-right dependency parsing with pointer networks”. In: *arXiv preprint arXiv:1903.08445* (2019).

[15] Daniel Gildea and Daniel Jurafsky. “Automatic labeling of semantic roles”. In: *Computational linguistics* 28.3 (2002), pp. 245–288.

[16] Paolo Giudici and Robert Castelo. “Improving Markov chain Monte Carlo model search for data mining”. In: *Machine learning* 50 (2003), pp. 127–158.

[17] Angelos Katharopoulos et al. “Transformers are rnns: Fast autoregressive transformers with linear attention”. In: *International conference on machine learning*. PMLR. 2020, pp. 5156–5165.

[18] Eliyahu Kiperwasser and Yoav Goldberg. “Easy-first dependency parsing with hierarchical tree LSTMs”. In: *Transactions of the Association for Computational Linguistics* 4 (2016), pp. 445–461.

[19] Terry Koo and Michael Collins. “Efficient third-order dependency parsers”. In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. 2010, pp. 1–11.

[20] Sandra Kübler, Ryan McDonald, and Joakim Nivre. "Dependency parsing". In: *Dependency parsing*. Springer, 2009, pp. 11–20.

[21] Zuchao Li, Hai Zhao, and Kevin Parnow. "Global greedy dependency parsing". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 05. 2020, pp. 8319–8326.

[22] Marie-Catherine de Marneffe et al. "Universal Dependencies". In: *Computational Linguistics* 47.2 (July 2021), pp. 255–308. ISSN: 0891-2017. DOI: 10.1162/coli_a_00402. eprint: https://direct.mit.edu/coli/article-pdf/47/2/255/1938138/coli_a_00402.pdf. URL: https://doi.org/10.1162/coli%5C_a%5C_00402.

[23] Ryan McDonald, Koby Crammer, and Fernando Pereira. "Online Large-Margin Training of Dependency Parsers". In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. Ed. by Kevin Knight, Hwee Tou Ng, and Kemal Oflazer. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 91–98. DOI: 10.3115/1219840.1219852. URL: https://aclanthology.org/P05-1012.

[24] Ryan McDonald and Joakim Nivre. "Characterizing the errors of data-driven dependency parsing models". In: *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*. 2007, pp. 122–131.

[25] Ryan McDonald and Joakim Nivre. "Analyzing and integrating dependency parsers". In: *Computational Linguistics* 37.1 (2011), pp. 197–230.

[26] Richard Myers, RC Wison, and Edwin R Hancock. "Bayesian graph edit distance". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22.6 (2000), pp. 628–635.

[27] Joakim Nivre. "An efficient algorithm for projective dependency parsing". In: *Proceedings of the eighth international conference on parsing technologies*. 2003, pp. 149–160.

[28] Joakim Nivre. "Incrementality in deterministic dependency parsing". In: *Proceedings of the workshop on incremental parsing: Bringing engineering and cognition together*. 2004, pp. 50–57.

[29] Joakim Nivre. "Incremental non-projective dependency parsing". In: *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*. 2007, pp. 396–403.

[30] Joakim Nivre, Johan Hall, and Jens Nilsson. "Memory-based dependency parsing". In: *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-2004) at HLT-NAACL 2004*. 2004, pp. 49–56.

[31] Joakim Nivre and Ryan McDonald. "Integrating Graph-Based and Transition-Based Dependency Parsers". In: *Proceedings of ACL-08: HLT*. Ed. by Johanna D. Moore et al. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 950–958. URL: https://aclanthology.org/P08-1108.

[32] Matthew E. Peters et al. "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 2227–2237. DOI: 10.18653/v1/N18-1202. URL: https://aclanthology.org/N18-1202.

[33] Kenji Sagae and Alon Lavie. "Parser combination by reparsing". In: *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. 2006, pp. 129–132.

[34] Libin Shen, Jinxi Xu, and Ralph Weischedel. "A New String-to-Dependency Machine Translation Algorithm with a Target Dependency Language Model". In: *Proceedings of ACL-08: HLT*. Ed. by Johanna D. Moore et al. Columbus, Ohio: Association for Computational Linguistics, June 2008, pp. 577–585. URL: https://aclanthology.org/P08-1066.

[35] Rion Snow, Daniel Jurafsky, and Andrew Ng. "Learning syntactic patterns for automatic hypernym discovery". In: *Advances in neural information processing systems* 17 (2004).

[36] Kristina Toutanova et al. "Compositional learning of embeddings for relation paths in knowledge base and text". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1434–1444.

[37] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[38] David Weiss et al. "Structured training for neural network transition-based parsing". In: *arXiv preprint arXiv:1506.06158* (2015).

[39] Jiaxuan You et al. "Graphrnn: Generating realistic graphs with deep auto-regressive models". In: *International conference on machine learning*. PMLR. 2018, pp. 5708–5717.

[40] Sheng Zhang et al. "AMR Parsing as Sequence-to-Graph Transduction". In: *CoRR* abs/1905.08704 (2019). arXiv: 1905.08704. URL: http://arxiv.org/abs/1905.08704.

[41] Sheng Zhang et al. “Broad-Coverage Semantic Parsing as Transduction”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3786–3798. DOI: 10.18653/v1/D19-1392. URL: https://aclanthology.org/D19-1392.