MASTER'S THESIS

Graph Parsing as Sequence Labeling

Student: Ana Xiangning Pereira Ezquerro

Supervisors: Carlos Gómez Rodríguez

David Vilares Calvo

A Coruña, March 6, 2025.

I don't want a life that is easy, but one that is worth it.

Acknowledgements

I would first like to acknowledge the effort and dedication of my two mentors, Carlos and David, for guiding me through a significant part of my academic journey and sharing their invaluable experience and knowledge. Special thanks to Carlos, who serves as a great inspiration to me, both as a researcher and as a person, and to David, for all the trust he places in me and his willingness to help me through the most challenging moments.

I would also like to thank all the members of LyS group for the great moments shared during coffee breaks and our travels abroad, as well as the professors and students from the School of Computer Science with whom I have shared almost six years of my life. To those who, despite time and distance, continue to hold me in their thoughts, and even to those who are no longer with me. Without your presence and support, I would not be where I am today.

Finally, my deepest gratitude goes to my family and my dearest friends, both those who are with me in my daily life and those who live far away. I know I am not always the easiest person, yet you remain by my side, offering me your patience, love and support. Thank you for being such an essential part of my life. I hope to make you proud to have me as part of yours.

Abstract

Graph processing is a fundamental task in Computer Science and Artificial Intelligence that involves modeling relationships between nodes in structured data. State-of-the-art approaches, while effective, suffer from quadratic complexity since they process all possible paired connections between the nodes of an input graph, making them computationally expensive for large-scale applications. To address this challenge, we incorporate the principles of the sequence-labeling paradigm to graph parsing, proposing new graph linearizations that encode graph structures as sequences of labels. This transformation enables parsing with linear complexity, significantly improving efficiency. Our work builds on state-of-the-art neural sequence-labeling frameworks and introduces both bounded and unbounded linearizations tailored for graph parsing. We conduct an empirical evaluation, comparing our approach against traditional graph-based methods on benchmark datasets. The results demonstrate that our proposed linearizations achieve competitive performance while reducing computational overhead, paving the way for more scalable and efficient graph processing.

Keywords:

- Natural Language Processing
- Artificial Neural Networks
- Graph Parsing
- Sequence labeling
- Large Language Models

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Graph processing is a core task in Computer Science, given the ubiquity of graphs in representing complex relationships across various domains. From social networks to biological structures and logistic systems, graphs provide a powerful and versatile abstraction for modeling entities and their interconnections. Effective graph processing is essential for computer systems in order to solve problems such as pathfinding, clustering and relationship inference, which are common in many practical applications.

Graphs have been extended in multiple Artificial Intelligence (AI) problems, offering a natural way to encode complex relationships and dependencies in data. In areas like recommender systems, graphs represent user-item interactions [1, 2], while in bioinformatics, they model molecular structures and biological pathways [3]. Similarly, in computer vision, scene graphs capture spatial and semantic relationships among objects [4]. These applications leverage graph-based representations to uncover patterns, propagate information, and enable inference in ways that traditional data structures often cannot achieve.

In Natural Language Processing (NLP), graph processing allows capturing the rich structural and relational information inherent in language. Dependency and constituency parsing [5] use graphs to represent syntactic and semantic structures, facilitating deeper linguistic analysis. Recent advances, such as Graph Neural Networks [6, 7, 8], have extended the capabilities of traditional graph methods by enabling learning directly from graph-structured data, leading to improvements in tasks like text classification [9], machine translation [10] and question answering [11]. This integration of graph processing techniques into NLP has opened new frontiers in understanding and leveraging the complexities of human language.

This project focuses on **graph parsing**, an NLP task that aims to accurately modeling the relationships between nodes in a given input graph. The nature of these relationships defines the specific objectives of the task, which may involve extracting semantic relations (commonly referred to as *semantic parsing* [12]), identifying causal links for emotions (*emotion-cause analysis* [13]), analyzing syntactic dependencies [14], or capturing sentiments [15]. Figure 1.1 shows a example of a semantic graph extracted from the SemEval 2015 Task 18 dataset [16]. This graph represents the semantic relationships between the words of the sentence "*Mr. Vinken is a chairman of Elsevier NV., the Dutch publishing group*", where the words represent the vertices of the graph and the edges represent the paired semantic relations.

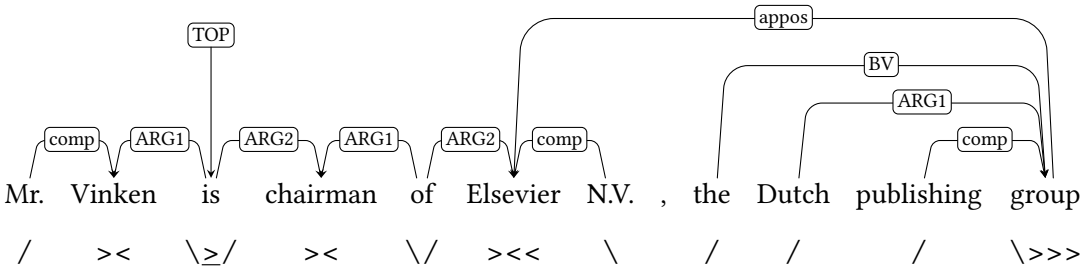


Figure 1.1: Semantic graph extracted from the SemEval 2015 Task 18 dataset [16]. The symbols below represent the bracketing encoding, which will be explained in detail in Section 3.2. Intuitively, in the bracketing encoding each node's label (e.g. >>< for *Elsevier*) is drawing the incoming and outgoing arcs with arrow or slash symbols aligned with the arc's direction. The symbol $\underline{\geq}$ is underlined to remark that the associated arc is not closed by any head, thus being reserved for the root node, marked with the label TOP.

State-of-the-art (SoTA) approaches for graph parsing, named *graph-based* approaches [17, 18], represent graphs as combinations of all possible pairwise relations between nodes, leading to a quadratic increase in complexity as the graph's size (number of nodes) grows. Alternatives such as *transition-based* systems [19, 20] can reduce this complexity to the number of arcs of the graph, leading to a substantial improvement in terms of efficiency when the graph is sparse, but still limited to the same quadratic complexity in the worst case.

This project proposes several **graph linearizations** (also named as *graph encodings*) to reframe graph parsing as a sequence-labeling task. **Sequence Labeling (SL)** approaches allow representing complex structures like graphs as a sequence of labels that matches the size of the input, reducing the complexity of the parsing task to linear. Figure 1.1 shows an example of a **graph linearization** known as bracketing encoding, where each node's label represents the incoming and outgoing arcs using arrow or slash symbols aligned with the arc's direction.

Previous work has proposed SL approaches for constituent [21, 22] and dependency parsing [23, 24, 25], leveraging the more restricted structure of constituent and dependency trees. Since graphs are more expressive, linearizing their architecture poses a great challenge in NLP. To the best of our knowledge, this problem has not been thoroughly addressed until now, although it holds a common interest not only in NLP but also in other AI fields. By converting complex graph structures into linear representations, SL approaches simplify processing and computation, making it a valuable tool for tasks that might require graph processing.

We adopt the SoTA neural framework for SL [21, 24], where a large neural encoder is used to contextualize information of an input sequence and compute dense vector representations, and a simple neural decoder is used to predict the sequence of labels from the latent vectors as a classification task. The full architecture is trained end-to-end with an annotated treebank and evaluated with a common benchmark [12]. Thus, inspired by previous works on dependency parsing [23, 25], this project proposes a set of **bounded** and **unbounded** linearizations for graph parsing and conducts and extensive experimental study to assess their performance and efficiency against traditional graph-based approaches.

1.1 Thesis' overview: Goals and report structure

The main goals of this project are:

1. Proposing several bounded and unbounded graph linearizations, analyzing their properties, computational complexity, and the transformations required for the encoding and decoding processes. This includes developing the encoding algorithm, which transforms the graph structure into a sequence of labels, ensuring that the original graph information can be fully recovered from the compressed representation with the decoding algorithm.

2. Implementing a neural framework designed to train our SL-based parsers using annotated treebanks. The framework incorporates SoTA neural architectures and optimization techniques, enabling effective learning of the proposed graph encodings.

3. Analyzing the performance and efficiency of our systems against potential baselines. This analysis assess factors such as parsing accuracy, computational overhead, scalability, and generalization to unseen graph structures.

Following these principles, the structure of this report is divided into seven chapters. Chapter 1 provides a general overview of the thesis, including a description of its goals, the methodology used, and the materials supporting it. Chapter 2 reviews the background required to understand previous work on graph parsing and explores how it can be reformulated as a SL task, revisiting Deep Learning (DL) models commonly used for parsing and focusing on prior linearization approaches that inspired this project. Chapters 3 and 4 introduce the unbounded and bounded encodings developed in this project, defining their transformation and analyzing their properties and theoretical foundations. Chapter 5 outlines the experimental methodology, describing the neural framework implemented to train our SL-based parsers, the datasets and evaluation metrics selected, the optimization strategies, and the baseline systems against which our models are compared. Chapter 6 presents the results obtained with our parsers and provides an in-depth analysis of their performance and efficiency relative to graph-based baselines. Finally, Chapter 7 summarizes the main insights of the project, highlights its contributions, and discusses potential avenues for future research.

1.2 Methodology

The development of this thesis follows an incremental-iterative approach. At the outset, a set of overarching objectives was defined, accompanied by approximate timelines for completing each iteration. Weekly meetings were held throughout the project to propose new iterations and review progress on previous ones. These meetings aimed to validate the implemented models and their results while identifying and addressing any errors or anomalies encountered during development.

Figure 1.2 shows the Gantt diagram of the project timeline. The project started on January 2024 and it was extended until February 2025. Its main contribution was presented as a long paper, and accepted, at the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP

2024)[1], under the title "*Dependency Graph Parsing as Sequence Labeling*" [26]. The corresponding iterations and conference deadlines are also displayed in the Gantt diagram.

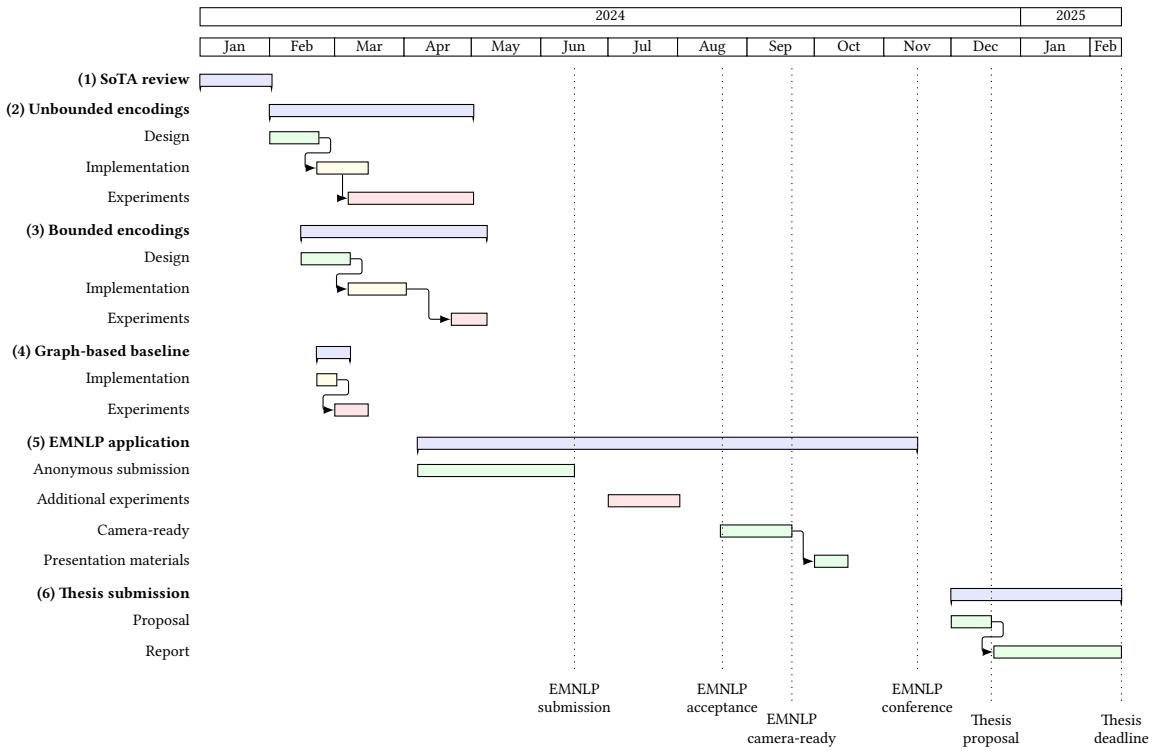


Figure 1.2: Gantt diagram illustrating the project timeline and milestones. Green bars denote iterations focused on report writing or literature review; yellow bars indicate coding iterations, and red bars represent iterations dedicated to experiments. Purple bars display grouped iterations.

The workflow of the project is distributed in five groups of iterations .

1. **SoTA review**: The initial phase focused on reviewing relevant works in graph parsing and existing linearization systems for other parsing tasks (such as dependency parsing). This was a crucial step for justifying the use of SL approaches for graph parsing and evaluating potential algorithms for effective graph linearization.

2. **Unbounded encodings**: These iterations were dedicated to designing, implementing, and deploying unbounded encodings within the neural framework. This project proposes three unbounded encodings that are detailed Chapter 3. In the Gantt diagram, the implementation phase follows the completion of the design iteration. However, the experimental phase overlaps with the implementation phase due to the independent nature of each encoding. Once the first unbounded encoding was implemented, experiments were initiated while the development of subsequent encodings continued in parallel. This overlapping workflow allowed for an efficient use of time and resources, optimizing the overall project timeline.

[1] According to the CORE 2023 ranking, the EMNLP is a top-tier (A*) venue highly regarded in the field of AI and NLP (FoR 4602), with an average rating of 5.0.

3. **Bounded encodings**: This phase groups the design, implementation, and testing of bounded encodings. The project introduces two bounded encodings, focusing on their unique characteristics and performance, which are elaborated on in subsequent sections (Chapter 4).

4. **Graph-based baseline**: To ensure a fair evaluation of the proposed SL parsers, a SoTA graph-based parser was selected, integrated into the neural framework, and assessed under identical conditions. This comparison provides a robust benchmark for evaluating the performance of the proposed methods.

5. **EMNLP application**. The project was submitted to EMNLP 2024, requiring multiple steps, including the preparation of an anonymous submission, starting in April 2024 and concluding in June 2024. Following the initial submission, additional experiments were conducted using new treebanks to enhance the experimental study. Upon acceptance, further iterations focused on preparing the camera-ready version and presentation materials for the conference.

6. **Thesis submission**: This phase included two key deliverables: the proposal manuscript for approval and the final thesis report. The goal of these iterations is to deliver a well-documented presentation of the project's findings.

1.3 Tools, materials and resources

To validate our graph linearizations and develop our neural parsers, we relied on Python 3.11 as the main programming language with the following libraries:

- PyTorch 2.2, a widely used deep learning framework for building and deploying neural networks. PyTorch is especially known for its support of GPU acceleration, making it ideal for training large models, and its seamless integration with CUDA/NVIDIA technologies, enabling fast computations. It provides a dynamic computational graph, which is useful for research and experimentation, allowing for easy debugging and modification of models.

- HuggingFace Transformers 4.45, an open-source library with the official implementations and pretrained weights of SoTA DL models, allowing simple fine-tuning for specific tasks. The library simplifies the use of powerful architectures and provides direct access to an extensive repository of models and datasets, supporting model development and reducing the time required to train new models from scratch.

- SuPar 1.14, an open repository with SoTA approaches for dependency, constituency and semantic parsing. The code was used to run experiments for graph-based baselines.

To train our parsers we relied on two annotated open-source datasets:

- The SemEval 2015 Task 18 dataset [16], which consists of five different treebanks with semantic annotations in the SDP-format: [i] the English treebank annotated with DELPHIN MRS-Derived Bi-Lexical Dependencies (DM) [27]; [ii-iii] the English and Chinese datasets with Enju Predicate–Argument Structures (PAS) [28]; and [iv-v] the English and Czech datasets

with Prague Semantic Dependencies (PSD) [29]. Each treebank provides three splits: a training set, an in-distribution (ID) split, and an out-of-distribution (OOD) split, with the last two reserved for evaluation. For the English and Czech treebanks, we used the recommended split, taking the sentences from Section 20 for validation. For the Chinese treebank, since no specific recommendation is provided, we used the sentences from Section 2 for validation.

- The IWPT 2021 Shared Task dataset [30], a multilingual benchmark that contains enhanced universal dependencies in 17 different languages following the Enhanced CoNLL-format: Arabic, Bulgarian, Czech, English, Estonian, Finnish, French, Italian, Latvian, Dutch, Polish, Slovakian, Swedish, Tamil, and Ukrainian. Each treebank is already split in three subsets for training, validation, and evaluation, which were maintained for our experiments.

Chapter 2
Background

This chapter introduces the foundational concepts in DL, NLP and Natural Language Understanding (NLU) to understand the proposed graph linearizations and the neural framework developed in this project. With the advent of neural networks for processing unstructured data (e.g. text [31, 32] or image [33, 34]), NLP has undergone a significant improvement by adapting neural networks for learning complex patterns and relationships from large-scale annotated datasets. These architectures, designed to capture the sequential nature of language, aim to encode sentences into dense vectorized representations – commonly referred to as embeddings. Numerous architectures have been proposed, each differing in how they contextualize sequential information and learn optimal parameters via error propagation [35, 36].

This chapter reviews the neural networks that have been most influential in NLP tasks, such as recurrent networks and Transformer-based architectures (Section 2.1). It formally introduces the graph parsing task and the approaches that have been designed for the problem (Section 2.2). Finally, it presents the SL approach and its theoretical adaptation to graph parsing (Section 2.3).

2.1 Neural networks for NLP

The first neural network proposed for classification problems was the Feed Forward Network (FFN) [37]. This architecture is conformed by a sequence of connected *layers*, where the output of the previous layer is fed as input to the next one. Each layer is composed by multiple neurons, where each neuron has a weight vector $(\mathbf{w}, b)$ that is optimized via gradient descent [38, 39]. Formally, let l_i be the i-th layer of the FFN with d_i neurons, for $i = 1, ..., m$. Each layer l_i, after performing some operations, always returns a hidden vector $\mathbf{h}_i \in \mathbb{R}^{d_i}$. Since each neuron in layer l_i has a weight vector $(\mathbf{w}, b) \in \mathbb{R}^{d_{i-1}+1}$, when stacking the weight vectors of the d_i neurons in a layer l_i, we obtain a matrix $(\mathbf{W}_i, \mathbf{b}_i) \in \mathbb{R}^{d_i \times (d_{i-1}+1)}$. The hidden vector $\mathbf{h}_i$ is computed from the previous one $\mathbf{h}_{i-1} \in \mathbb{R}^{d_{i-1}}$ as $\mathbf{h}_i = \sigma(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$, where $\sigma : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_i}$ is a non-linear activation function that transforms the output vector.

The process of propagating the vectors $\mathbf{h}_1, .., \mathbf{h}_m$ through the network is called *forward pass*. The FFN is fed with an input feature vector $\mathbf{x} \in \mathbb{R}^{d_0}$ and is passed through each layer until the last one (commonly known as *output layer*) returns its hidden vector $\mathbf{h}_m = \text{FFN}(\mathbf{x})$, which serves as the

output of the full architecture, also denoted as $\hat{\mathbf{y}} = \text{FFN}(\mathbf{x}) \in \mathbb{R}^m$.[1]

The defining property of the FFN is its ability to optimize its weights to approximate the output $\hat{\mathbf{y}} \in \mathbb{R}^m$ to a ground truth $\mathbf{y} \in \mathbb{R}^{d_m}$. In classification tasks, since each input feature vector is associated with a class $y \in \mathcal{C}$, where $\mathcal{C}$ is the set of possible classes. The ground truth is represented as the one-hot encoding of y, meaning $\mathbf{y} \in [0, 1]^{|\mathcal{C}|}$, while $\hat{\mathbf{y}}$ estimates the probability distribution of the input over $\mathcal{C}$. Given a *loss function* that quantifies the similarity between $\hat{\mathbf{y}}$ and $\mathbf{y}$, the FFN iteratively optimizes its weights through gradient descent to refine $\hat{\mathbf{y}}$ and approximate it to the ground truth. The error propagates from the last layer to the first in a process known as *backward pass*. By training on multiple feature vectors $(\mathbf{x}_1, .., \mathbf{x}_n)$ paired with ground truths $(\mathbf{y}_1, ..., \mathbf{y}_n)$, the FFN learns to model complex nonlinear relationships in the input space to produce accurate predictions.

FFNs have demonstrated remarkable effectiveness in both classification and regression tasks. Their primary advantage against traditional Machine Learning (ML) models lies in their ability to automatically learn features directly from annotated data, eliminating the need for manual feature engineering, a key limitation of traditional ML methods such as Support Vector Machines or Decision Trees. In classification tasks, the FFN adjusts its output layer to match the number of classes and minimizes the cross-entropy loss during training to approximate the true probability distribution of the target classes. For regression tasks, FFNs map input features to continuous values, optimizing loss functions such as mean squared error (MSE). Unlike traditional approaches, FFNs can learn both linear and nonlinear patterns without requiring explicit assumptions about the nature of the relationships in the data.

2.1.1 Word embeddings

Early attempts to adapt neural networks to NLP [40] faced significant challenges in handling discrete tokens, such as words, as input [41]. Neural networks rely on optimization processes that require continuous feature representations, but approaches like one-hot encoding – where each word in the vocabulary is represented as a sparse binary vector – proved inadequate. This method not only struggled with optimization in high-dimensional spaces but also suffered from severe scalability issues as the vocabulary size grew.

To address the limitations of one-hot encoding and capture semantic relationships in a latent space, [42, 43] introduced the concept of **learnable word embeddings**. This method maps each input token to a continuous dense vector – commonly referred to as embedding – with its parameters optimized through backpropagation. The training process is designed to ensure that the embedding space reflects semantic relationships between tokens, resulting in dense vectors that encode meaningful semantic properties by the end of training.

Currently, *static* [44, 45] and *contextualized* [31, 46] embeddings are the most common technique used to feed discrete tokens into a neural network. Word2vec [44] introduced a simple method for generating static embeddings using a neural architecture optimized for two different tasks: (i) the bag of words model, which predicts a masked word given its surrounding context; and (ii) the skip-gram model, which predicts the context words given a target work. The trained embeddings have

[1] The hat accentuation (ˆ) is commonly used to denote predictions of a model. For instance, $\hat{\mathbf{y}}$ denotes the prediction of a model that is optimized to learn a ground truth $\mathbf{y}$, where both $\mathbf{y}$ and $\hat{\mathbf{y}}$ are vectors.

proven effective in capturing word similarity and analogies, forming the basis for many advances in NLP. However, they lack the ability to adapt to word meaning changes based on context, which later motivated the development of *contextualized* embeddings.

Unlike static embeddings, which assign a single fixed vector to each word, contextualized embeddings dynamically generate representations that reflect the surrounding words in a sentence, enabling the model to capture nuanced meanings and resolve ambiguities through the context. These embeddings became widely adopted with the release of Large Language Models (LLMs) [31, 47], neural architectures that process entire sequences as input and dynamically integrate information from other tokens in the sequence into each token's embedding, thus producing dense vectors enriched with contextualized sequence properties. When trained on tasks like masked token prediction, these models are guided to use the surrounding context to refine their trainable embeddings. Once the training is complete, the output from the final layer serve as contextualized word representations that encode rich semantic and syntactic information. Contextualized embeddings have shown to be vastly superior to static ones in language understanding tasks and now form the backbone of SoTA NLP solutions.

2.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) were proposed to overcome a key limitation of FFNs when handling the sequential nature of tasks such as time series analysis and NLP. Since FFNs are inherently non-sequential and cannot model temporal dependencies between inputs, RNNs were specifically designed to handle sequences by maintaining a memory (in the form of hidden states) over time, thus capturing dependencies between different elements of an input sequence.

Vanilla RNN The first recurrent cell proposed was the vanilla RNN [48]. For an input feature sequence, $\mathbf{X} = (\mathbf{x}_1, .., \mathbf{x}_n) \in \mathbb{R}^{n \times d_x}$, the vanilla RNN uses a hidden state $\mathbf{h}_i \in \mathbb{R}^{d_h}$ that is dynamically updated storing the information of previous inputs of a current element $\mathbf{x}_i$. At each timestep i, the hidden state is updated based on its previous value ($\mathbf{h}_{i-1}$) and the current input ($\mathbf{x}_i$) via a non-linear operation with trainable matrices $\mathbf{W}_h \in \mathbb{R}^{d_h \times d_h}$, $\mathbf{W}_x \in \mathbb{R}^{d_x \times d_h}$ and $\mathbf{b}_h \in \mathbb{R}^{d_h}$. Then, an output $\mathbf{y}_i \in \mathbb{R}^{d_y}$ is computed based on the current hidden state $\mathbf{h}_i$ using two additional trainable matrices $\mathbf{W}_y \in \mathbb{R}^{d_h \times d_y}$ and $\mathbf{b}_y \in \mathbb{R}^{d_y}$ (Equation 2.1). Note that, by updating the hidden state with the current input, the network is able to learn which features needs to maintain from the previous context $(\mathbf{x}_1, ..., \mathbf{x}_{i-1})$ to return a response $\mathbf{y}_i$ optimizable via backpropagation (Figure 2.1).

$$\begin{aligned} \mathbf{h}_i &= \sigma(\mathbf{h}_{i-1}\mathbf{W}_h + \mathbf{x}_t\mathbf{W}_x + \mathbf{b}_h) \\ \mathbf{y}_i &= \sigma(\mathbf{h}_i\mathbf{W}_y + \mathbf{b}_y) \end{aligned} \tag{2.1}$$

Long-Short Term Memory (LSTM) While vanilla RNNs can model dependencies in an input sequence, they struggle to capture long-range dependencies (due to the limited capacity of a single hidden state) and suffer from the problem of the vanishing gradient [49]. The LSTM [50] addresses this

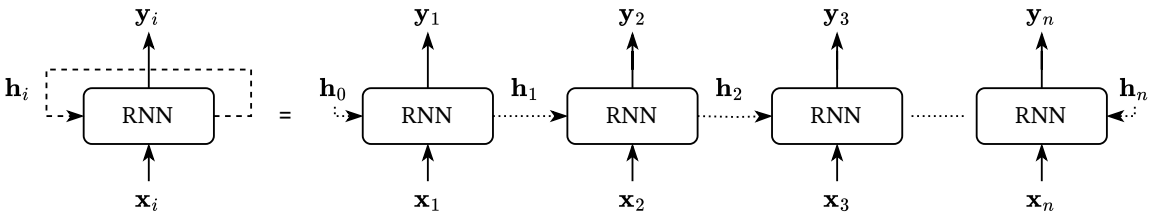


Figure 2.1: Visualization of the forward pass in the vanilla RNN.

limitation by introducing two hidden vectors: the memory cell $\mathbf{c}_i$ and the hidden state $\mathbf{h}_i$. These vectors are updated through more sophisticated non-linear operations, enabling the network to retain information from distant timesteps more effectively. The Bidirectional LSTM (BiLSTM) [51] further enhances the capabilities of the LSTM by enabling bidirectional processing. In a BiLSTM, the input sequence is fed into a standard LSTM in its original order and simultaneously reversed and passed through a second LSTM. This allows the second LSTM to learn dependencies from right-to-left. The hidden states and outputs from both LSTMs are concatenated, yielding a final representation that captures bidirectional dependencies.

The LSTM has proven highly effective in NLU tasks [52, 53]. ELMo [47], one of the first LLMs, leveraged stacked BiLSTM layers as its backbone to produce contextualized word embeddings that dynamically capture word meanings based on context. Pretrained on large text corpora, ELMo showcased remarkable performance across various NLP tasks.

2.1.3 Transformer block

The Transformer [36] redefined sequence modeling with an efficient, parallelizable, and scalable architecture centered on the *self-attention mechanism*, making it the backbone of most SoTA NLP architectures. The Transformer block is conformed by multiple layers, including multi-head self-attention and FFNs, which collectively map an input sequence $\mathbf{X} = (\mathbf{x}_1, .., \mathbf{x}_n) \in \mathbb{R}^{n \times d_x}$ to an output sequence of the same length $\mathbb{Y} = (\mathbf{y}_1, ..., \mathbf{y}_n) \in \mathbb{R}^{n \times d_y}$ through a series of non-linear transformations and attention computations.

The self-attention is the core innovation of the Transformer block, allowing each output $\mathbf{y}_i$ to incorporate information from all elements of the input sequence by making the transformation of each element context-dependent. Specifically, the self-attention module first projects the input sequence $\mathbf{X} \in \mathbb{R}^{n \times d_x}$ into three different representations: (i) the query $\mathbf{Q} = \mathbf{X}\mathbf{W}_q \in \mathbb{R}^{n \times d_k}$, (ii) the key $\mathbf{K} = \mathbf{X}\mathbf{W}_k \in \mathbb{R}^{n \times d_k}$; and (iii) the value $\mathbf{V} = \mathbf{X}\mathbf{W}_v \in \mathbb{R}^{n \times d_v}$; where $\mathbf{W}_q \in \mathbb{R}^{d_x \times d_k}$, $\mathbf{W}_k \in \mathbb{R}^{d_x \times d_k}$, $\mathbf{W}_v \in \mathbb{R}^{d_x \times d_v}$ are trainable matrices. The self-attention computes the matrix product between these three matrices to dynamically extract contextualized features of each input in the sequence (Equation 2.2).

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \underbrace{\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)}_{\text{score matrix } (\mathbf{S})} \mathbf{V} \in \mathbb{R}^{n \times d_v} \tag{2.2}$$

The Transformer parallelizes multiple attention heads within a single block, followed by FFNs

with non-linear activations, residual connections and normalization layers to ensure stable optimization. The final output is a new sequence $\mathbf{Y} = (\mathbf{y}_1, ..., \mathbf{y}_n) \in \mathbb{R}^{n \times d_v}$ where each element is a contextualized non-linear projection of its corresponding input.

2.1.4 Large Language Models

The success of the Transformer block for NLP tasks sparked extensive research focused on LLMs. By stacking multiple Transformers within a single architecture and training this model on vast amounts of textual data, researchers developed powerful models – dubbed as Large Language Models – capable of excelling specific NLP tasks. With a pretraining stage designed to capture semantic and contextual information from text, LLMs learn accurate language representations that could be fine-tuned for specific applications.

LLMs transformed NLP by setting new benchmarks across a wide range of tasks, including question answering, text classification and machine translation. This progress has been driven by three distinct paradigms in pretraining objectives, which define the architecture and applications of these models: **encoder-only models**, such as BERT [31], RoBERTa [46] and XLNet [54], usually pretrained on discriminative tasks such as Masked Language Modeling , **decoder-only models**, such as GPT [32, 55, 56], LLaMA [57, 58, 59] and Mistral [60, 61], pretrained on next token prediction; and **encoder-decoder models**, such as BART [62] and T5 [63], pretrained on text-to-text generation.

Figure 2.2 shows a visualization each paradigm. Encoder-only architectures (Figure 2.2a) process input sequences using Transformer layers to produce contextualized outputs of the same length. Decoder-only architectures (Figure 2.2b) focus on generating the next token $\mathbf{y}_{i+1}$ from an input context $(\mathbf{x}_1, ..., \mathbf{x}_i)$. To efficiently parallelize this paradigm for all tokens in the input sequence, the self-attention operation is modified by setting the upper diagonal of the score matrix (Equation 2.2) to zero, thus ensuring each token $\mathbf{x}_i$ is contextualized only with past elements (this is commonly known as *masked attention*). Encoder-decoder architectures (Figure 2.2c) combine the strengths of both approaches: the encoder first contextualizes the entire input sequence, and the decoder generates the output tokens step-by-step by recurrently feeding its previously generated tokens back into the model.

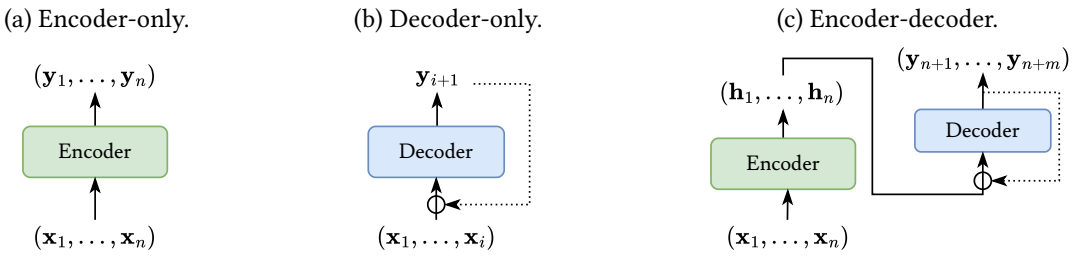


Figure 2.2: Different paradigms for LLMs.

For this work, we focus on **encoder-only** architectures, which are more commonly used for NLU tasks than the other two paradigms, primarily due to the bidirectional nature of their pretraining objectives. Previous studies [64, 65] have explored the integration of generative models into NLU

tasks through an extensive experimental study, demonstrating that masked attention significantly limits the ability of neural networks in natural language reasoning.

BERT BERT [31] is a widely used Transformer-based, encoder-only language model for English. Pretrained on the English Wikipedia [66] and BookCorpus [67] datasets, it optimizes two training objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, random tokens of the input sequence are replaced by a special token `[MASK]`, and BERT is tasked to predict the original tokens. For NSP, two sentences are concatenated interleaved by a special token `[SEP]` to form a single input sequence, and BERT predicts whether the two sentences are consecutive.

To adapt the structure of the input to perform MLM and NSP, BERT adds a special token `[CLS]` at the beginning of an input sequence. Thus, for an input sequence $(\texttt{[CLS]}, w_1, .., w_n)$ with some masked tokens, BERT returns a sequence of contextualized embeddings $(\mathbf{h}_0, \mathbf{h}_1, ..., \mathbf{h}_n)$. Those embeddings associated with the `[MASK]` token are used to predict the original token. The first one $\mathbf{h}_0$, which corresponds to the `[CLS]` token, is used to predict whether the input corresponds to two consecutive sentences.

Fine-tuned on the GLUE benchmark [68], BERT achieved SoTA performance in NLU tasks, such as question answering and sentiment analysis. Since then, it has been widely adopted as encoder for more complex NLP tasks [69, 70, 71], often paired with customized decoders to specific output structures.

RoBERTa RoBERTa [46] was introduced as an improved version of BERT, leveraging the same Transformer-based architecture while addressing limitations in BERT's pretraining process. Specifically, RoBERTa removed the NSP task, increased the training batch size and extended the number of training epochs. These modifications enabled RoBERTa to achieve superior performance on the GLUE benchmark, establishing it as a more effective encoder for NLU tasks.

Building on RoBERTa's success, **XLM-RoBERTa** [72] was introduced as its multilingual extension. It shares the same architecture and pretraining objectives as RoBERTa but was trained on CommonCrawl data [73] spanning over 100 languages. This multilingual capability has made XLM-RoBERTa a valuable tool for cross-lingual NLU tasks.

XLNet XLNet [54] is a Transformer-based model that addresses the limitations of BERT and RoBERTa by replacing MLM by Permutation Language Modeling (PLM) as pretraining objective and integrating the Transformer-XL [74] as backbone. The key idea of PLM is generating all possible permutations of a sentence to train the model on next token prediction. When producing all possible permutations, XLNet captures the bidirectional context without masking tokens during consecutive epochs. As a result, XLNet avoids the independence assumptions of MLM and better leverages the full context of a sequence. These enhancements enable XLNet to outperform BERT and RoBERTa on several NLU benchmarks.

2.2 Graph Parsing

In graph parsing, an input sentence is represented as a directed labeled graph $G = (W, A)$, where the set of nodes is an ordered sequence $W = (w_1, ..., w_n) \in \mathcal{V}^n$ representing the words of a sentence[2], and the set of arcs A fulfills the following conditions:

1. Each arc is denoted as $(h \xrightarrow{r} d)$, where $d \in [1, n]$ and $h \in [0, n]$, and represents a dependency from an outgoing node w_h (the parent or head) to an incoming node w_d (the dependent) with a relationship type $r \in \mathcal{R}$. In Figure 1.1, there is an arc $(3 \xrightarrow{\text{ARG1}} 2)$ that connects the word "*is*" to the word "*Vinken*" with the semantic relation "ARG1".

2. Cycles of length one are not permitted, and each pair of entry points of W has at most one associated arc in A. Formally, for every arc of the form $(h \xrightarrow{r} d)$, it holds $h \neq d$, and if $\exists (h \xrightarrow{r} d) \in A$, then $\nexists (h \xrightarrow{r'} d) \in A$ where $r \neq r'$.

3. Those nodes with an incoming arc from the artificial node w_0 are known as *root* nodes. In Figure 1.1, the only root of the sentence is the word "*is*", and its dependency on the node w_0 is displayed without an incoming arc $(0 \xrightarrow{\text{TOP}} 3)$. The number of roots in a graph is not restricted, so some graphs might have multiple nodes or none at all.

Graph parsing has been adopted in multiple tasks that involve extracting dependency relations in a sentence. In *semantic parsing* [16] the arcs represent semantic dependency relations between words of a sentence (Figure 1.1). In *enhanced dependency parsing* [30] the arcs collect syntactic dependencies between words in a more free-manner than dependency parsing [75]. In *sentiment parsing* [15] the full graph structure is used to represent the relationships, polarity and sentiment expressions of an input sentence. In *emotion-cause analysis* [13] the arcs are associated with emotion causal relations between different utterances of multiple sentences. All these tasks adapt graph parsing to process and input sequence and extract the paired dependencies between its different elements. Thus, in general terms, the goal of a *graph parser* is to accurately extract the set of dependency relations for a given input sequence.

Similar to other NLP tasks, SoTA graph parsing typically follows the encoder-decoder framework [17, 18, 20]. In this framework, the encoder – often a fine-tuned LLM or a custom recurrent or Transformer-based model – learns contextualized representations, while the decoder predicts a structured output from which a predicted set of arcs is recovered. Depending on the design of this structured output, research on this task has primarily followed two main approaches: graph-based approaches and transition-based systems.

2.2.1 Graph-based approaches

Graph-based approaches are characterized by a scoring system that assigns probability scores to all possible arcs in a sentence. Intuitively, these methods predict a score matrix corresponding to the dimensions of the input sentence and apply a threshold to determine which scores represent actual predicted arcs.

[2] We use the symbol $\mathcal{V}$ to denote the set of possible words and $\mathcal{R}$ to denote the set of arc labels.

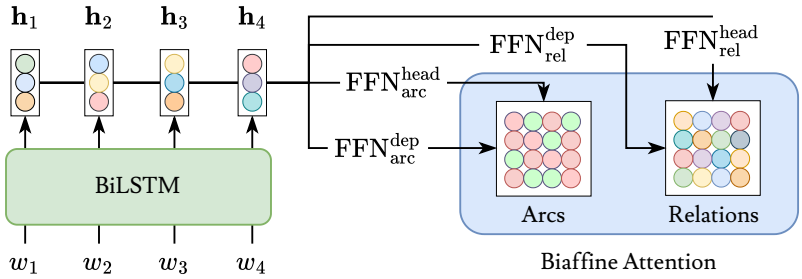


Figure 2.3: Visualization of the biaffine system [17] for an input sentence of $n = 4$ tokens.

Biaffine [17] is one of the most popular graph-based approaches proposed for semantic parsing due to its simplicity and efficient implementation. Figure 2.3 shows the main components of the biaffine parser. For an input sentence $(w_1, ..., w_n) \in \mathcal{V}^n$, a bidirectional encoder[3] returns a sequence of contextualized embeddings, $\mathbf{H} = (\mathbf{h}_1, .., \mathbf{h}_n) \in \mathbb{R}^{n \times d_h}$. Each embedding $\mathbf{h}_i$ is individually fed into four different FFNs to obtain four different representations of the same word w_i: the representation of the word as an arc dependent, as an arc head, as a relation dependent and as a relation head (Equations 2.3-2.6, respectively).

$$\mathbf{h}_i^{\text{(arc-dep)}} = \text{FFN}^{\text{dep}}_{\text{arc}}(\mathbf{h}_i) \in \mathbb{R}^{d_{\text{(arc-dep)}}} \tag{2.3}$$

$$\mathbf{h}_i^{\text{(arc-head)}} = \text{FFN}^{\text{head}}_{\text{arc}}(\mathbf{h}_i) \in \mathbb{R}^{d_{\text{(arc-head)}}} \tag{2.4}$$

$$\mathbf{h}_i^{\text{(rel-dep)}} = \text{FFN}^{\text{dep}}_{\text{rel}}(\mathbf{h}_i) \in \mathbb{R}^{d_{\text{(rel-dep)}}} \tag{2.5}$$

$$\mathbf{h}_i^{\text{(rel-head)}} = \text{FFN}^{\text{head}}_{\text{rel}} \in \mathbb{R}^{d_{\text{(rel-head)}}} \tag{2.6}$$

Figure 2.3 shows that the biaffine decoder contains two biaffine attention modules, where arcs and relation scores are computed. The first module computes the arc score matrix $\mathbf{S}^{\text{arc}} \in \mathbb{R}^{n \times n}$, where each score $s^{\text{arc}}_{d,h}$ at position (d, h) is obtained from the cross information of the arc representation of the dependent $\mathbf{h}_d^{\text{(arc-dep)}}$ and the head $\mathbf{h}_h^{\text{(arc-head)}}$ and assesses the probability of an arc $(h \rightarrow d)$. The second module returns instead the relation score tensor $\mathbf{S}^{\text{rel}} \in \mathbb{R}^{n \times n \times |\mathcal{R}|}$, where each score $s^{\text{rel}}_{d,h,r}$ represents the probability of a relation r in the arc from w_h to w_d. The first module uses a learnable matrix, denoted as $\mathbf{U}^{\text{arc}} \in \mathbb{R}^{d_{\text{(arc-dep)}} \times d_{\text{(arc-head)}}}$, and the second module uses three learnable tensors, denoted as $\mathbf{U}^{\text{rel}} \in \mathbb{R}^{d_{\text{(rel-dep)}} \times |\mathcal{R}| \times d_{\text{(rel-head)}}}$, $\mathbf{W}^{\text{rel}} \in \mathbb{R}^{(d_{\text{(arc-dep)}} + d_{\text{(arc-head)}}) \times |\mathcal{R}|}$ and $\mathbf{b} \in \mathbb{R}^{|\mathcal{R}|}$. By stacking each word representation, $\mathbf{H}^* \in \mathbb{R}^{m \times d_*}$, where $* \in \{\text{(arc-dep)}, \text{(arc-head)}, \text{(rel-dep)}, \text{(rel-head)}\}$, the computation of both tensors can be efficiently performed through a parallelizable tensor product defined in Equation 2.7. Note that the biaffine decoder is just computing each score as the dot product of each representation linearly projected to a learnable space, which actually resembles to the self-attention mechanism (Equation 2.2).

[3] Originally, [17] used a BiLSTM-based encoder, although other studies have reported results using more powerful architectures, such as a fine-tuned LLM [24, 76].

$$
\begin{aligned}
\mathbf{S}^{\text{arc}} &= \mathbf{H}^{(\text{arc-dep})}\mathbf{U}^{\text{arc}}\mathbf{H}^{(\text{arc-head})} \\
\mathbf{S}^{\text{rel}} &= \mathbf{H}^{(\text{rel-dep})}\mathbf{U}^{\text{rel}}\mathbf{H}^{(\text{rel-head})} + \left[\mathbf{H}^{(\text{rel-dep})}, \mathbf{H}^{(\text{rel-head})}\right]\mathbf{W} + \mathbf{b}
\end{aligned}
\tag{2.7}
$$

The biaffine parser was evaluated on the English treebanks of the SemEval 2015 Task 18 dataset [16] and remains as one of the best-performing approaches in semantic parsing. Subsequent works have enhanced its performance by making targeted modifications to either the decoder or the encoder. For example, [18] introduces second-order decoding to better capture arc distributions from the score matrix, while [77] replaces the static embedding layer with dynamic embeddings. Despite these noticeable improvements, the biaffine parser continues to serve as a robust baseline for graph-based approaches.

2.2.2 Transition-based systems

Transition-based systems incrementally construct the graph structure by applying a sequence of predefined *transitions* (or actions) to update the system's current state s_t. Starting from an initial state s_0, at each timestep t, for $t = 1, ..., T$, the transition system performs an action τ_t (such as adding arcs or skipping nodes) that updates s_{t-1} to s_t until a final state s_T is reached.

SoTA transition systems are usually guided by a neural model. Each state of the system is encoded through neural representations to predict the most likely next action based on the current state and context. To train the models, an *oracle* is used to provide the optimal sequence of transitions needed to construct the correct graph.

Covington [78] is one of the fundamental transition-based algorithms to parse graph structures. It relies two pointers i and j, constrained to $i < j$, that process the input sentence $(w_1, ..., w_n)$ from left to right, creating arcs that connect the words w_i and w_j. Formally, each state $s_t = (i, j, \hat{A})$ at timestep t is defined by the two pointers and the set of recovered arcs $\hat{A}$. In the initial state, the pointers are fixed at the start of the sequence and the set of recovered arcs is empty, $s_0 = (0, 1, \emptyset)$. At each timestep t, the system performs one of these four actions:

1. *Left-arc(r)*: Creates an arc $(j \xrightarrow{r} i)$, adds it to $\hat{A}$ and decreases i by one if $i > 0$.
2. *Right-arc(r)*: Creates and arc $(i \xrightarrow{r} j)$, adds it to $\hat{A}$ and decreases i by one if $i > 0$.
3. *No-arc*: Decreases i by one.
4. *Shift*: Increases j by one and sets i to $j - 1$.

The final state $s_T = (i, n+1, \hat{A})$ is reached when the pointer j surpasses the length of the sentence, and Covington returns $\hat{A}$ as the set of predicted arcs. Table 2.1 shows the sequence of transitions and states performed by Covington to recover all arcs of the graph in Figure 1.1. Note that some *shift* actions are executed when there are no more arcs between w_j and previous tokens to w_i, thus reducing the number of transitions required to parse the full graph.

	Transition	New arc	Updated state
s_0			(w_0) Mr. Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_1	*shift*		(w_0) Mr. Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_2	*right-arc*(comp)	$(1 \xrightarrow{\text{comp}} 2)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_3	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_4	*left-arc*(ARG1)	$(3 \xrightarrow{\text{ARG1}} 2)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_5	*no-arc*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_6	*right-arc*(TOP)	$(0 \xrightarrow{\text{TOP}} 3)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_7	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_8	*right-arc*(ARG2)	$(3 \xrightarrow{\text{ARG2}} 4)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_9	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{10}	*left-arc*(ARG1)	$(5 \xrightarrow{\text{ARG1}} 4)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{11}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{12}	*right-arc*(ARG2)	$(5 \xrightarrow{\text{ARG2}} 6)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{13}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{14}	*left-arc*(comp)	$(7 \xrightarrow{\text{comp}} 6)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{15}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{16}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{17}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{18}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{19}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{20}	*right-arc*(comp)	$(11 \xrightarrow{\text{comp}} 12)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{21}	*right-arc*(ARG1)	$(10 \xrightarrow{\text{ARG1}} 12)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{22}	*right-arc*(BV)	$(9 \xrightarrow{\text{BV}} 12)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{23}	*no-arc*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{24}	*left-arc*(appos)	$(12 \xrightarrow{\text{appos}} 6)$	(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group
s_{25}	*shift*		(w_0) Mr Vinken is chairman of Elsevier N.V. , the Dutch publishing group

Table 2.1: Covington decoding [78] for the semantic graph of Figure 1.1. The pointer i is marked with a blue bullet (●) and the pointer j is marked with the red bullet (●).

2.3 Sequence-labeling for Graph Parsing

The idea behind the SL paradigm is to represent complex structures (like graphs) as a sequence of labels of the same length as the input sentence. Prior work has applied SL approaches (also named *linearizations*) to dependency and constituency tree parsing, which are intrinsically related to graph parsing, as trees constitute a special case of graphs. For instance, [23] proposed a naive encoding for dependency trees. Dependency trees are graphs where each node is limited to only have one head, so a naive approach is to use the positions of the heads as the sequence of labels. Figure 2.4 shows an example of this encoding: note that each node w_i has only one head (i.e. one incoming arc), so the set of arcs can be represented with a sequence of labels where each label is simply the position of the head of w_i.

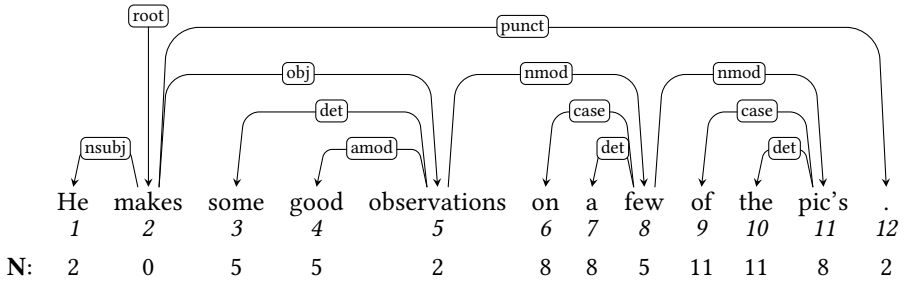


Figure 2.4: Dependency tree extracted from the English EWT dataset [79]. The second row displays the word positions. The third row (**N**) shows the naive encoding proposed by [23].

By representing a complex structure like a tree as a sequence of labels (where each label is associated to only one token of the sentence), the design of the neural architecture to predict these structures is simplified to a tagging task, where the model just needs to perform classification at token level over the set of possible labels. In tagging tasks, the neural architecture commonly follows the encoder-decoder architecture. The encoder is fed with the input sentence $(w_1, ..., w_n)$ and produces contextualized embeddings $(\mathbf{h}_1, .., \mathbf{h}_n)$. The decoder is a one-layered FFN with a softmax activation function that maps each contextualized token embedding to the probability distribution over the set of labels. Note that this approach considerably simplifies the decoder module, avoiding complex neural representations such as biaffine to recreate the original graph structure.

Once the sequence of labels is predicted, the reverse process needs to be executed to recover the original form of the input structure (in graph parsing, a set of arcs is recovered). In the naive encoding for dependency trees (Figure 2.4), all arcs are recovered by creating a connection between each node and the value of its label.

The forward transformation of the input structure to the sequence of labels is known as **encoding**, while the reverse transformation to recover the original structure from the sequence of labels is known as **decoding**. Note that, when the sequence of labels is correctly predicted by a neural model, the decoding process correctly recovers the real graph. Otherwise, if some label is incorrectly predicted, the errors might be propagated to the decoding process, producing incorrect arcs when rebuilding the graph. Figure 2.5 shows an abstract diagram of the SL approach.

Several SL approaches have been proposed for a wide range of NLP tasks, such as entity link-

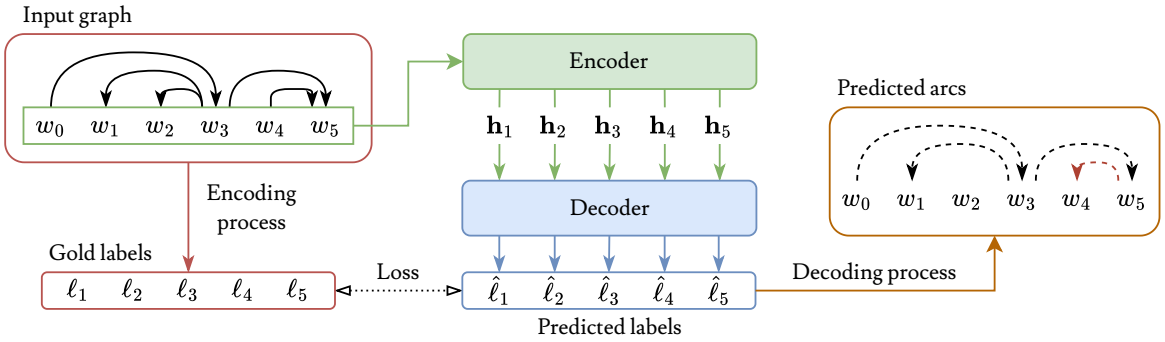


Figure 2.5: Visualization of a SL system for graph parsing. An encoder-decoder architecture learns to contextualize the input sentence and predict a label per input token. The network is trained as a standard classification task using the gold labels obtained through the encoding process. The decoding process recovers a set of arcs from the predicted labels. See that some arcs might be incorrect (colored in red) or missed in the predicted graph due to potential errors produced by the network.

ing [80], event extraction [81], aspect-based sentiment analysis [82] and dependency [24] and constituency [21, 22] parsing. However, its application in graph parsing remains unexplored. This work proposes **graph linearizations** inspired from previous dependency tree encodings (such as the one displayed in Figure 2.4), effectively extending their expressiveness to minimally constrained graphs.

2.3.1 Motivation

Graph-based based approaches like biaffine (Section 2.2.1) and transition-based systems like Covington (Section 2.2.2) have demonstrated competitive performance in graph parsing across several benchmarks [14, 16]. However, both methods present specific challenges that motivate this work on **reframing graph parsing as sequence labeling**.

Since graph-based approaches compute the score of all possible arcs in an input graph, they typically suffer from quadratic complexity with respect to the input length, which can make them computationally expensive for longer sentences. For instance, as demonstrated in the previous section, the biaffine parser [17] operates with a complexity of $O(n^2)$. However its second-order variant [18] increases the complexity to $O(n^3)$, significantly damaging the system's efficiency as the input sequence length grows.

On the other hand, transition-based approaches exhibit variable complexity, as their runtime depends on the number of states generated during parsing, which is strongly influenced by the decisions made by the trained model and the algorithm itself. Covington [78] has also a quadratic complexity $O(n^2)$ in the worst case (parsing all possible head-dependent pairs) and it is still limited to $O(|A|)$ in the best case. Alternatives, such as the transition-based system with pointer networks proposed by [20], although reaching paired performance with graph-based approaches, still face $O(n^2)$ complexity in the average case (for sparse graphs) and $O(n^3)$ in the worst case.

The SL paradigm enables the compression of the graph information into a sequence of labels aligned with the input length, thereby reducing the theoretical complexity[4] of graph parsing to

[4] We refer to theoretical complexity since the SoTA graph parsers do not scale linearly in practice due to the

linear, which represents a substantial improvement over traditional methods. Additionally, as it is displayed in Figure 2.5, reformulating graph parsing as a tagging task provides greater flexibility for designing and training alternative neural architectures, as tagging requires a simpler and more straightforward output structure compared to traditional graph-based or transition-based methods [83, 84].

This work proposes graph linearizations inspired by previous dependency tree encodings [23, 25], effectively extending their expressiveness to minimally constrained graphs, and demonstrating that sequence labeling can combine high efficiency with performance closely matching the state of the art.

2.3.2 Formalization

We now introduce a formal definition for graph linearizations and the notation that will be followed throughout the next chapters to describe our proposed SL approaches. Let $W = (w_1.., w_n) \in \mathcal{V}^n$ be an input sentence and $G = (W, A)$ its potential graph. Let $\mathcal{A}^n$ be the set of all possible set of arcs (constrained to the properties described in Section 2.2) for n-sized graphs and $\mathcal{L}$ the set of possible labels of the SL algorithm.

- The encoding process is an injective function $\varepsilon : \mathcal{A}^n \rightarrow \mathcal{L}^n$ that maps any set of arcs A into a sequence of n labels, denoted as $\ell = (\ell_1, ..., \ell_n) \in \mathcal{L}^n$.
- The decoding process is instead a surjective function $\delta : \mathcal{L}^n \rightarrow \mathcal{A}^n$ that recovers a valid set of arcs from a sequence of labels ℓ.

As shown in Section 2.3, a well-formed graph linearization defines ε and δ to satisfy that a set of arcs A can be recovered from its encoded representation, formally $A = \delta(\varepsilon(A))$, optionally under some additional assumptions over A.[5] For simplicity, from now on we are going to ignore the relationship type r in the arcs defined in Section 2.2, so each arc is now an unlabeled directed arc of the form $(h \rightarrow d)$, where $h \in [0, n]$, $d \in [1, n]$ and $h \neq d$. This component r will be revisited in Chapter 5.

Depending on how the set of labels $\mathcal{L}$ is defined we say that the linearization is bounded or unbounded. When the cardinality of $\mathcal{L}$ is not restricted, we say that the encoding is unbounded. For example, the dependency encoding displayed on Figure 2.4 is unbounded since the set of labels potentially grows with the length of the sentence, since it uses the absolute positions of the head tokens. Otherwise, when $\mathcal{L}$ is fixed independently of $\mathcal{A}^n$, we say that the encoding is bounded. In this work we propose several graph linearizations that are grouped in unbounded (Chapter 3) and bounded algorithms (Chapter 4).

Transformer-based backbone, which always adds a quadratic complexity.

[5] For instance, the linearization displayed in Figure 2.4 is also valid for graphs, but it assumes that A is constrained to only have one incoming arc per node.

Chapter 3

Unbounded linearizations

In this chapter we define our proposed unbounded linearizations with their corresponding encoding and decoding transformation. The encoding process (ε) maps the set of arcs A into a sequence of labels $\ell = (\ell_1, ..., \ell_n) \in \mathcal{L}^n$, where $\mathcal{L}$ is not bounded. The decoding process (δ) performs the reverse transformation and recovers A from ℓ. We grouped the unbounded linearizations in positional and bracketing encodings.

Section 3.1 introduces the **positional linearization**, which is the simplest approach for representing the arc information as a sequence of labels, formalizing the encoding, decoding and postprocessing steps. The postprocessing is an additional step included in the decoding process to ensure that the set of recovered arcs is valid (for example, to ensure that it does not produce cycles of length one), and it is usually needed for predicted labels with potential errors that initially might produce non-valid arcs.

Section 3.2 focuses on the **bracketing linearization** and its hyperparameter k. This linearization makes some assumptions over A to satisfy a consistent recovery, i.e. $A = \delta(\varepsilon(A))$ if and only if A fulfills some conditions. Section 3.2 discusses the **coverage** of the bracketing encoding, this is, the ratio of graphs that this algorithm is able to fully recover by modulating its hyperparameter k.

3.1 Positional encodings

We extend the positional encodings proposed for dependency trees (Figure 2.4) to support encoding multiple heads per node. In Figure 2.4 we showed that, since dependency trees only have one head per node, each label encodes the position of its unique head. In graphs, since each node might have multiple heads (or none at all), the label is instead composed by the sequence of positions of each head.

Encoding Equation 3.1 shows the simplest variant of the positional encoding, referred to as **absolute indexing**[1]. Note that the *sort* function arranges a numerical set in ascending order, and it is needed to avoid creating labels that are essentially the same (e.g. in Figure 3.1, $\ell_4^{\text{A}} = (3, 5)$ means the same that $\ell_4^{\text{A}} = (5, 3)$ so it should always be encoded in one way to avoid increasing the cardinality

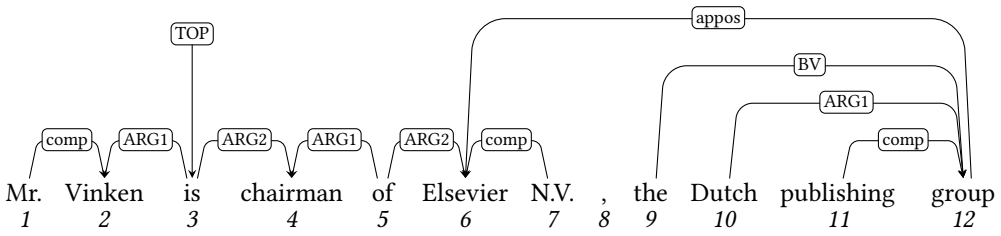
[1] The term *absolute* comes from the use of the absolute positions of the heads.

of $\mathcal{L}$). Equation 3.2 shows the **relative encoding**, which uses relative positions, thus reducing the encoding's reliance on the global positional information of W.

$$\ell_i^{\mathrm{A}} = \mathrm{sort}\Big\{h \mid (h \to i) \in A\Big\} \tag{3.1}$$

$$\ell_i^{\mathrm{R}} = \mathrm{sort}\Big\{h - i \mid (h \to i) \in A\Big\} \tag{3.2}$$

Figure 3.1 (rows **A** and **R**) shows the absolute and relative labels derived from the semantic graph of Figure 1.1. The absolute label of a node w_i is the ordered sequence of the positions of its head: for example, the node w_4 has two heads, w_3 and w_5, so its absolute label is $\ell_4^{\mathrm{A}} = (3, 5)$. The relative label instead subtracts to these positions the dependent position i, so $\ell_4^{\mathrm{R}} = (3 - 4, 5 - 4) = (-1, 1)$. Note that in this example the absolute encoding generates five different labels, $\mathcal{L}_{\mathrm{A}} = \{(1, 3), (0), (3, 5), (5, 7, 12), (9, 10, 11)\}$, while the relative encoding produces four, $\mathcal{L}_{\mathrm{R}} = \{(-1, 1), (-3), (-1, 1, 6), (-1, -2, -3)\}$. In real datasets, sparse graphs often exhibit recurring patterns in the association of heads to dependents. These patterns favor relative encoding, as dependents are consistently encoded with the same label regardless of their absolute position.



A:	-	(1,3)	(0)	(3,5)	-	(5,7,12)	-	-	-	-	-	(9,10,11)
R:	-	(-1,1)	(-3)	(-1,1)	-	(-1,1,6)	-	-	-	-	-	(-1,-2,-3)
B:	/	><	\>/	><	\/	>><	\		/	/	/	\>>>

Figure 3.1: Example of unbounded encodings: absolute (**A**), relative (**R**) and bracketing (**B**). The dash (-) is used to represent an empty sequence. Note that the closing bracket > in ℓ_3 does not match with any opening bracket / since it belongs to the root node.

Decoding The decoding process of both the absolute and relative encoding is straightforward. Each label is independently processed and recovers a subset of arcs connected to the node i: in the case of the absolute indexing, each element of ℓ_i^{A} is decoded as a head of w_i while in the relative indexing the position i needs to be added to the encoded position (Equations 3.3 and 3.4). For instance, in Figure 3.1, the absolute label of w_2 is $\ell_2^{\mathrm{A}} = (1, 3)$, thus, following Equation 3.3, the recovered arcs are $\{(1 \to 2), (3 \to 2)\}$. Instead, the relative label of w_4 is $\ell_4^{\mathrm{R}} = (-1, 1)$, thus, the

recovered arcs are $\{(3 \rightarrow 4), (5 \rightarrow 4)\}$ (Equation 3.4).

$$\delta_{\text{abs}}(\ell) = \bigcup_{i=1}^{n} \big\{(p \rightarrow i) : \forall p \in (\ell_i \cap [0, n])\big\} \tag{3.3}$$

$$\delta_{\text{rel}}(\ell) = \bigcup_{i=1}^{n} \big\{((p + i) \rightarrow i) : \forall p \in (\ell_i \cap [-i, n - i])\big\} \tag{3.4}$$

Postprocessing When training a neural model to learn the sequence of labels from input words, it might produce corrupted positions that create arcs with heads out of the range of $[0, n]$. Those situations are usually solved through heuristics [22, 23] that skip some information in the predicted labels. For positional encodings, the simplest heuristic is to ignore those positions that recover heads that are not in the range $[0, n]$.

3.2 Bracketing encoding

In the bracketing encoding, each label (ℓ_i^{B}) can be visualized as a graphical representation of the incoming and outgoing arcs that are connected to each node w_i. See again Figure 3.1 (row **B**). The node w_{12} has four connections: three incoming arcs from the left, which results in three arcs of the form $(* \rightarrow 12)$ that are translated into three right arrows >, and one outgoing arc to the left $(* \leftarrow 12)$, which is represented with the left slash \. The result is that the the node w_{12} is encoded as $\ell_{12} =$ \>>>.

Encoding Formally, in the bracketing encoding, each label ℓ_i^{B} is a string that adheres to the regular expression *>*<*/*, where the presence of each symbol in the bracket set, $B = \{\backslash, >, <, /\}$, indicates different types of connections to the node w_i:

- The symbol \ indicates the presence of an outgoing arc from w_i to the left.
- The symbol > indicates the presence of an incoming arc to w_i from the left.
- The symbol < indicates the presence of an incoming arc to w_i from the right.
- The symbol / indicates the presence of an outgoing arc from w_i to the right.

Figure 3.1 (row **B**) shows the bracketing encoding for the semantic graph in Figure 1.1. The label of w_6 is $\ell_6^{\text{B}} =$ ><<, indicating that there is an incoming arc from the left, specifically $(5 \xrightarrow{\text{ARG2}} 6)$, and two incoming arcs from the right, which are $(12 \xrightarrow{\text{appos}} 6)$ and $(7 \xrightarrow{\text{comp}} 6)$. The label $\ell_{12}^{\text{B}} =$ \>>> contains three repetitions of the symbol >, indicating that the node w_{12} has three heads from its left, which corresponds to $\{(9 \xrightarrow{\text{BV}} 12), (10 \xrightarrow{\text{ARG1}} 12), (11 \xrightarrow{\text{comp}} 12)\}$; and one dependent to its left: $(12 \xrightarrow{\text{appos}} 6)$.

Decoding In the bracketing encoding, an arc with the form $(h \rightarrow d)$ is encoded with only two symbols that are located in ℓ_h^{B} and ℓ_d^{B}. If the arc is a left arc[2] the symbol located in ℓ_h^{B} is \ and the

[2] Left arcs are those where the dependent is at the left of the head, while right arcs have its dependent at the right of the head.

symbol located in ℓ_d^{B} is <. If the arc is a right arc, the symbol located in ℓ_h^{B} is / and in ℓ_d^{B} is >. For example, the left arc $(3 \overset{\text{ARG1}}{\rightarrow} 2)$ in Figure 3.1 is represented in $\ell_2^{\text{B}} =$ >< and $\ell_3^{\text{B}} =$ \>/. The right arc $(1 \overset{\text{comp}}{\rightarrow} 2)$ is represented with $\ell_1^{\text{B}} =$ / and $\ell_2^{\text{B}} =$ ><. Thus, it is straightforward to see that left arcs are associated to < and \ symbols, while right arcs are represented with / and >, so the decoding process only needs to match the opening brackets < and / in a label ℓ_i^{B} with their corresponding closing ones, \ and >, in a subsequent label $\ell_{>i}^{\text{B}}$[3].

Formally, the bracketing decoding parses the full sequence of brackets from left to right, matching opening brackets (<, /) with subsequent closing ones (\, >) in the sequence. This process is performed with a **transition-based system of two stacks**: $\sigma_{\text{L}} \in [1, n]^*$ for the left arcs and $\sigma_{\text{R}} \in [0, n]^*$ for the right arcs; and a buffer $\beta \subseteq \ell$. Each stack keeps track of the positions of the opening brackets that have been parsed, and the buffer stores the brackets that have not yet been processed. In the initial state, σ_{L} is empty, σ_{R} only contains the index 0, β is initialized with the full sequence brackets and the set of recovered arcs $\hat{A}$ is empty. At each timestep t, the element located at the front of the buffer, represented as $\beta^{\top} = b_i$, contains the bracket symbol $b \in B$ and the index i of the label that is being processed. Depending on b and $\hat{A}$, the system performs one of these actions using the elements at the top of σ_{L} and σ_{R}, denoted as $\sigma_{\text{L}}^{\top}$ and $\sigma_{\text{R}}^{\top}$, respectively:

- *Open-left*: Removes $\beta^{\top}$ and pushes i to σ_{L}.
- *Open-right*: Removes $\beta^{\top}$ and pushes i to σ_{R}.
- *Close-left*: Adds $(i \rightarrow \sigma_{\text{L}}^{\top})$ to $\hat{A}$, pops σ_{L} and removes $\beta^{\top}$.
- *Resolve-right*: Adds $(\sigma_{\text{R}}^{\top} \rightarrow i)$ to $\hat{A}$ and removes $\beta^{\top}$.
- *Close-right*: Adds $(\sigma_{\text{R}}^{\top} \rightarrow i)$ to $\hat{A}$, pops σ_{R} and remove $\beta^{\top}$.
- *Skip*: Removes $\beta^{\top}$.

Table 3.1 shows the bracketing decoding as a deductive system. See that *open-left* and *open-right* actions are performed when the element at the front of the buffer is an opening bracket (< or /), while *close-left*, *resolve-right* or *close-right* are executed when a closing bracket (\ or >) is found in the buffer. Each stack is storing the opening positions of left and right arcs, respectively. Thus, when a closing bracket is found, the system matches the positions of one of the stacks with the current position of the front of the buffer. In case of the right arcs, the system might pop σ_{R} (with the *close-right* action) or maintain the top element of σ_{R} (*resolve-right*). In fact, σ_{R} is only popped when $\sigma_{\text{R}} \neq 0$. The index 0 is never removed from σ_{R} since the encoding assumes that all nodes with the unmatched symbol >, correspond to a root node connected with w_0.

See Table 3.2 for a better visualization of the decoding process of the graph in Figure 3.1. When the symbol $\backslash_i$ appears in the front of the buffer, the system tries to close a left bracket using the element at the top of σ_{L}. Instead, when the symbol $>_i$ appears, the system tries to close a right bracket from the top position stored at σ_{R}. Note that, when the sequence of brackets is *well-formed*, the *skip* action is never used and σ_{L} is empty in the final state and σ_{R} only contains the index 0.

[3] We use $(> i)$ or $(< i)$ as a subscript to denote a position that comes before i or after i, respectively.

Transition	Preconditions	Actions
open-left		$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}} \vert i, \sigma_{\mathrm{R}}, \beta, \hat{A})}(b = <)$
open-right		$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}} \vert i, \beta, \hat{A})}(b = /)$
close-left	$\vert\sigma_{\mathrm{L}}\vert > 0$ $(i \rightarrow \sigma_{\mathrm{L}}^{\top}) \notin A$	$\frac{(\sigma_{\mathrm{L}} \vert \sigma_{\mathrm{L}}^{\top}, \sigma_{\mathrm{R}}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, \beta, \hat{A} \cup \{(i \rightarrow \sigma_{\mathrm{L}}^{\top})\})}(b = \backslash)$
resolve-right	$\vert\sigma_{\mathrm{R}}\vert > 0$ $(\sigma_{\mathrm{R}}^{\top} \rightarrow i) \notin \hat{A}$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}} \vert \sigma_{\mathrm{R}}^{\top}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}} \vert \sigma_{\mathrm{R}}^{\top}, \beta, \hat{A} \cup \{(\sigma_{\mathrm{R}}^{\top} \rightarrow i)\})}(b = >, \sigma_{\mathrm{R}}^{\top} = 0)$
close-right	$\vert\sigma_{\mathrm{R}}\vert > 0$ $(\sigma_{\mathrm{R}}^{\top} \rightarrow i) \notin \hat{A}$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}} \vert \sigma_{\mathrm{R}}^{\top}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, \beta, \hat{A} \cup \{(\sigma_{\mathrm{R}}^{\top} \rightarrow i)\})}(b = >, \sigma_{\mathrm{R}}^{\top} \neq 0)$
skip		$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b_i \vert \beta, \hat{A})}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, \beta, \hat{A})}$

Table 3.1: Bracketing decoding as a deductive system. Note that the transitions have an ordered preference and, due to the symbol order in the bracket labels, is not possible to produce cycles of length one. The *skip* transition is executed for unclosed left arcs or repeated arcs (arcs with the same head and dependent).

However, when a sequence of labels produces unclosed arcs or repeated arcs, the system is able to skip those symbols due to the preconditions defined for each action.

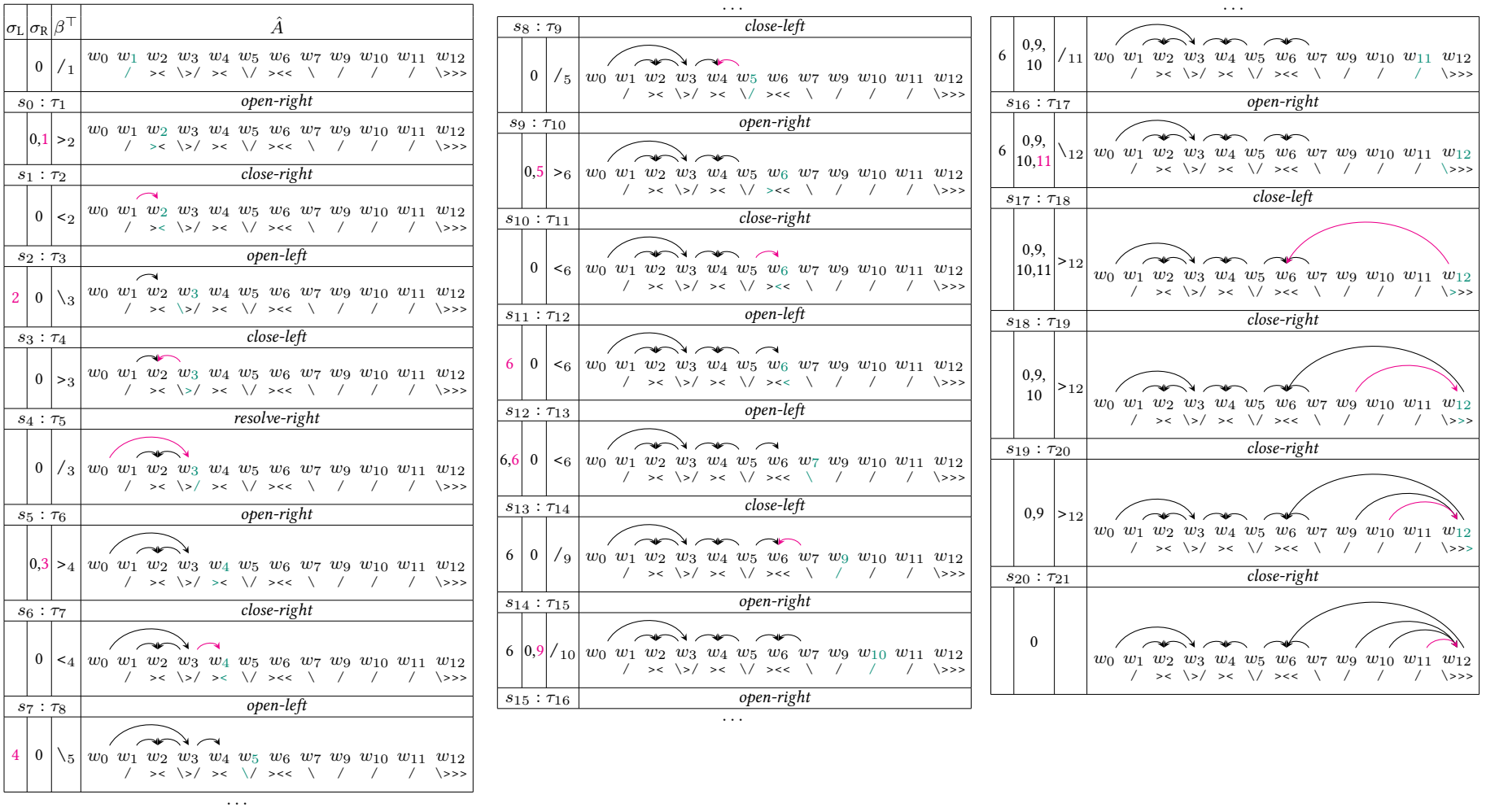
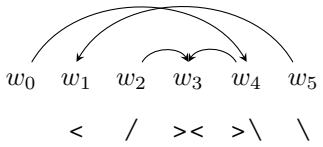


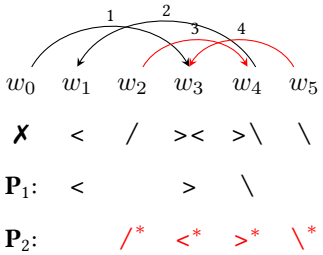
Table 3.2: Bracketing decoding for the graph in Figure 3.1. Words and arc labels have been removed for clarity, as well as the node w_8 with its label ℓ_8 since no arc is connected with it. The table is read from top to bottom and left to right: the columns represent the current state $(\sigma_L, \sigma_R, \beta^\top, \hat{A})$, and $s_{t-1} : \tau_t$ represents the transition executed at timestep t. Note that the state obtained after transition t is represented in the next row. The dots $(\cdots)$ indicate that the table continues in the table of the next column. $\beta^\top$ is colored in green and the system updates are colored in magenta.

Relaxed planarity Take a closer look at the two graphs displayed in Figures 3.2a and 3.2b (row ✗) and their decoding process in Table 3.2d. The two graphs are different since they do not have the same arcs. However, following the encoding described previously, they produce the *same* sequence of labels, which clearly violates the injective property of the encoding transformation. and leads to an inconsistent decoding process, which is not able to recover the second graph (Table 3.2d). This inconsistency arises because the bracketing encoding is limited to only represent arcs that do not cross in the same direction. The first graph (Figure 3.2a) contains some crossing arcs but in different directions, while the second graph (Figure 3.2b) contains crossing arcs in the same direction.

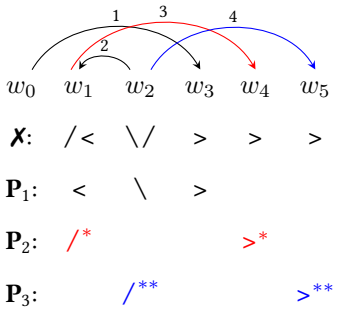
(a) Relaxed 1-planar graph.



(b) Relaxed 2-planar graph.



(c) Relaxed 3-planar graph.



(d) Incorrect decoding process for Figure 3.2b (row ✗).

	σ_L	σ_R	$\beta^\top$	$\hat{A}$	τ_t
s_0		0	$<_1$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< >\ \	*open-left*
s_1	1	0	$/_2$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< >\ \	*open-right*
s_2	1	0,2	$>_3$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< >\ \	*close-right*
s_3	1	0	$<_3$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< >\ \	*open-left*
s_4	1,3	0	$>_4$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< >\ \	*resolve-right*
s_5	1,3	0	$\backslash_4$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< \ \	*close-left*
s_6	1	0	$\backslash_5$	w_0 w_1 w_2 w_3 w_4 w_5 < / >< \ \	*close-left*
s_7		0		w_0 w_1 w_2 w_3 w_4 w_5 < / >< \ \	*close-left*

Figure 3.2: Crossing examples for the bracketing encoding. Different relaxed-planes are colored in red and blue. The numbers located above each arrow represent the order followed to distribute the arcs in different relaxed-planes. See that row ✗ leads to errors in the decoding process (Table 3.2d). For Figure 3.2b, the solution relies on separately encoding each relaxed-plane ($\mathbf{P}_1$, $\mathbf{P}_2$) with different symbols and concatenate them at token level, resulting in $\ell = (<, /^*, ><^*, \backslash>^*, \backslash\backslash^*)$. For Figure 3.2c, a third relaxed-plane is added ($\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{P}_3$), resulting in $\ell = (</^*, \backslash/^{**}, >, >^*, >^{**})$.

We can check that this inconsistency arises for crossing arcs in the same direction with the arcs created in the decoding process displayed in Table 3.2d. To recover the graph in Figure 3.2b, the state s_2 (*close-right* action) should use the position 0 in σ_R, not the index 2, which is the one that is at the top of σ_R. The problem is that the stacks only store the previous opening brackets in each direction, so introducing crossing arcs in the same direction adds an opening bracket between the

real brackets that should be matched.

To overcome this issue, we propose **distributing the arcs** in different relaxed-planes[4], (i.e. mutually exclusive subsets of A that do not contain crossing arcs in the same direction). This distribution follows a deterministic order (shown in Figures 3.2b and 3.2c) by increasingly sorting the arcs of A by their left ($\min\{h, d\}$) and right ($\max\{h, d\}$) component, and iteratively assign them to the lowest valid relaxed-plane. For instance, in Figure 3.2b the arcs $(0 \rightarrow 3)$ and $(2 \rightarrow 4)$ cross each other in the same direction but $(2 \rightarrow 4)$ is the one sent to $\mathbf{P}_2$ because its left component is greater than the left component of $(0 \rightarrow 3)$.

Once the arcs are distributed in different relaxed-planes, each one is independently encoded with other set of equivalent brackets that only match with each other at decoding time (e.g. $B^* = \{\backslash^*, >^*, <^*, /^*\}$ for the second plane and $B^{**} = \{\backslash^{**}, >^{**}, <^{**}, /^{**}\}$ for the third plane), and the labels produced are concatenated at token level.

Figure 3.2c, shows a complex graph where three relaxed-planes are necessary to cover all the arcs. Each one is encoded with different symbols – for the second relaxed-plane we use B^* and for the third, B^{**} – resulting in three different sequences that separately encode each one: $\ell^1 = (<, \backslash, >, \lambda, \lambda)$[5] for $\mathbf{P}_1$, $\ell^2 = (/^*, \lambda, \lambda, >^*, \lambda)$ for $\mathbf{P}_2$ and $\ell^3 = (\lambda, /^{**}, \lambda, \lambda, >^{**})$ for $\mathbf{P}_3$. The final sequence concatenates ℓ^1, ℓ^2 and ℓ^3 at token level, so $\ell = (</^*, \backslash/^{**}, >, >^*, >^{**})$. The decoding process of this sequence is displayed in Table 3.3. The labels are separated again by the relaxed-plane they come from and are independently decoded to recover the corresponding arcs. At the end, the recovered arcs of each process are joined to return a unique set of recovered arcs $\hat{A}$.

Hyperparameter k When extending the bracketing encoding to multiple relaxed-planes it is possible to modulate its coverage by fixing the number of supported relaxed-planes (k). For an input graph, the encoding process first distributes its arcs in, at most, k relaxed-planes. Those arcs that cannot be assigned in any relaxed-plane – since they cross with other arc in the same direction – are ignored for the encoding, with the limitation of not being recovered after the decoding. Note that when fixing k, the bracketing linearization is only able to recover graphs restricted to k or less relaxed-planes. For example, in Figure 3.2c, if the encoding fixes $k = 2$, only $\mathbf{P}_1$ and $\mathbf{P}_2$ are encoded, resulting in $\ell = (</^*, \backslash, >, >^*, \lambda)$, and the bracketing linearization is limited to only recovering the arcs in $\mathbf{P}_1$ and $\mathbf{P}_2$. Increasing k allows covering more complex arcs with a higher number of crossing arcs in the same direction. In real datasets, setting $k = 3$ covers more than the 99% of the annotated sentences in our treebanks, so k is usually restricted to the range $[1, 3]$.

The full encoding process is formally defined in Algorithm 1. Note that the procedure DISTRIBUTE processes the arcs of A in a deterministic order[6]. Then ENCODE-ONE encodes each relaxed-plane as described in Section 3.2, adding to each symbol as many asterisks as the value of p. Each label is concatenated at token level and the final sequence is returned as output.

The decoding process is defined in Algorithm 2. The main function is DECODE, which accepts

[4] In the literature, the term *plane* usually refers to a subset of non-crossing arcs, independently of their direction. Under this definition the graph of Figure 3.2a is 2-planar, but since the crossing arcs are in different directions, we say that it is *relaxed 1-planar*.

[5] Here we use λ to denote an empty string.

[6] The function also arranges the arcs of an input set by their left and right components.

(a) Decoding of $\mathbf{P}_1$.

σ_L	σ_R	$\beta^\top$	$\hat{A}$
	0	$<_1$	$w_0\ w_1\ w_2\ w_3\ w_4\ w_5$ $<\ \backslash\ >$
$s_0 : \tau_1$			*open-left*
1	0	$\backslash_2$	$w_0\ w_1\ w_2\ w_3\ w_4\ w_5$ $<\ \backslash\ >$
$s_1 : \tau_2$			*close-left*
	0	$>_3$	$w_0\ w_1\ w_2\ w_3\ w_4\ w_5$ $<\ \backslash\ >$
$s_2 : \tau_3$			*resolve-right*
	0		$w_0\ w_1\ w_2\ w_3\ w_4\ w_5$ $<\ \backslash\ >$

(b) Decoding of $\mathbf{P}_2$.

σ_L	σ_R	$\beta^\top$	$\hat{A}$
		$/^*_1$	$w_1\ w_2\ w_3\ w_4\ w_5$ $/^*\ \ >^*$
$s_0 : \tau_1$			*open-right*
1		$>^*_4$	$w_1\ w_2\ w_3\ w_4\ w_5$ $/^*\ \ >^*$
$s_1 : \tau_2$			*close-right*
			$w_1\ w_2\ w_3\ w_4\ w_5$ $/^*\ \ >^*$

(c) Decoding of $\mathbf{P}_3$.

σ_L	σ_R	$\beta^\top$	$\hat{A}$
		$/^{**}_2$	$w_1\ w_2\ w_3\ w_4\ w_5$ $/^{**}\ \ >^{**}$
$s_0 : \tau_1$			*open-right*
	2	$>^{**}_5$	$w_1\ w_2\ w_3\ w_4\ w_5$ $/^{**}\ \ >^{**}$
$s_1 : \tau_2$			*close-right*
			$w_1\ w_2\ w_3\ w_4\ w_5$ $/^{**}\ \ >^{**}$

Table 3.3: Correct decoding process for the graph in Figure 3.2c. Same notation as in Table 3.2. Each plane is separately decoded with a different procedure. For a clearer visualization, the red and blue colors of the brackets of each relaxed-planes are suppressed though the asterisk remains to distinguish them. Note that in the decoding of $\mathbf{P}_2$ and $\mathbf{P}_3$ σ_R is initialized as empty, instead of with the index 0. This prevents creating root nodes in any relaxed-plane but $\mathbf{P}_1$, which is always correct since, due to the deterministic arc distribution, the root nodes are always assigned to $\mathbf{P}_1$.

the sequence ℓ as input and the value of the hyperparameter k. In the main loop, the decoding extracts the brackets in ℓ that previously belonged to the p-th relaxed-plane (for $p = 1, .., k$). The function EXTRACT-LABELS obtains those brackets with $p - 1$ asterisks. For example, when calling EXTRACT-LABELS$((<\!/^*, \backslash/^{**}, >, >^*, >^{**}), 2)$ it will return $(/^*, \lambda, \lambda, >^*, \lambda)$. The DECODE-ONE function performs the decoding process displayed in Table 3.1, recovering the arcs of the p-the relaxed-plane. At the end, all the recovered arcs are joined to return a single set.

Algorithm 1 Bracketing encoding algorithm.

procedure ENCODE(A, n, k)
  $D \leftarrow$ DISTRIBUTE(A, k)
  $\ell \leftarrow (\lambda : i = 1, ..., n)$
  for $p = 1, ..., |D|$ **do**
    $\ell' \leftarrow$ ENCODE-ONE($D_p, p - 1, n$)
    for $i = 1, ..., n$ **do**
      $\ell_i \leftarrow \text{concat}(\ell_i, \ell'_i)$
  return ℓ

procedure ENCODE-ONE(A, p, n)
  $\ell \leftarrow (\lambda : i = 1, ..., n)$
  for $(h \rightarrow d) \in A$ **do**
    if $h > d$ **then** ▷ left arc
      $\ell_h \leftarrow \text{concat}(\ell_h, \backslash(^*)^p)$
      $\ell_d \leftarrow \text{concat}(\ell_d, <(^*)^p)$
    else ▷ right arc
      $\ell_h \leftarrow \text{concat}(\ell_h, /(^*)^p)$
      $\ell_d \leftarrow \text{concat}(\ell_d, >(^*)^p)$
  return ℓ

procedure DISTRIBUTE(A, k)
  $D \leftarrow (\emptyset)$
  for $a \in \text{sort}(A)$ **do**
    $p \leftarrow 1$
    added $\leftarrow$ **false**
    while not added **and** $p \leq |D|$ **do**
      if not RELAXED-CROSS(a, D_p) **then**
        $D_p \leftarrow D_p \cup \{a\}$
        added $\leftarrow$ **true**
      else
        $p \leftarrow p + 1$
    if not added **and** $|D| < k$ **then**
      push($D, \{a\}$)
  return D

procedure RELAXED-CROSS(a, A)
  for $a' \in A$ **do**
    if CROSS(a, a') **and** DIR(a) = DIR(a') **then**
      return true
  return false

Algorithm 2 Bracketing decoding algorithm.

procedure DECODE(ℓ, k)
  $\hat{A} \leftarrow \emptyset$
  for $p = 1, ..., k$ **do**
    $\ell^p \leftarrow$ EXTRACT-LABELS($\ell, p, \hat{A}$)
    DECODE-ONE($\ell^p, p, \hat{A}$)
  return $\hat{A}$

procedure EXTRACT-LABELS(ℓ, p)
  $\ell^p \leftarrow (\lambda : i = 1, ..., |\ell|)$
  for $\ell_i \in \ell$ **do**
    for $b \in \ell_i$ **do**
      if $\text{count}(b, *) + 1 = p$ **then**
        $\ell^p_i \leftarrow \text{concat}(\ell^p_i, \text{remove}(b, *))$
  return ℓ^p

procedure DECODE-ONE($\ell, p, \hat{A}$)
  if $p = 1$ **then**
    $\sigma_{\text{R}} \leftarrow [0]$
  else
    $\sigma_{\text{R}} \leftarrow []$
  $\sigma_{\text{L}} \leftarrow []$
  for $\ell_i \in \ell$ **do**
    for $b \in \ell_i$ **do**
      if $b =$ **<** **then**
        push(σ_{L}, i)
      else if $b =$ **/** **then**
        push(σ_{R}, i)
      else if $b = \backslash \wedge |\sigma_{\text{L}}| > 0 \wedge (i \rightarrow \sigma_{\text{L}}^{\top}) \notin \hat{A}$ **then**
        $\hat{A} \leftarrow \hat{A} \cup \{(i \rightarrow \sigma_{\text{L}}^{\top})\}$
        pop(σ_{L})
      else if $b =$ **>** $\wedge |\sigma_{\text{R}}| > 0 \wedge (\sigma_{\text{R}}^{\top} \rightarrow i) \notin \hat{A}$ **then**
        $\hat{A} \leftarrow \hat{A} \cup \{\sigma_{\text{R}}^{\top} \rightarrow i)\}$
        pop(σ_{R})
  return $\hat{A}$

Chapter 4

Bounded linearizations

This chapter introduces our bounded linearizations for graph parsing. As seen in the previous chapter, the positional and bracketing encodings do not limit the set of possible labels ($\mathcal{L}$). In the case of the positional encoding, $\mathcal{L}$ can grow with the sentence length when creating longer arcs. On the other hand, the set of possible labels in the bracketing encoding can also grow by creating denser graphs: adding more incoming connections to a node indefinitely increases the repetitions of the symbols > and <, creating more labels in $\mathcal{L}$.

We now introduce two types of bounded encodings, where the cardinality of $\mathcal{L}$ is fixed independently of the length of the sentence or the density of the graph. The first one is named **4k-bit encoding** (Section 4.1) and the second is the **6k-bit encoding** (Section 4.2). Both are modulated by an hyperparameter k, which has a slightly similar meaning to the hyperparameter used in the bracketing encoding. In both encodings, the original input set of arcs (A) is distributed in k mutually exclusive *subsets* that are independently processed through the specific encoding and decoding functions.

Both linearizations are based on **bits**, so each label ℓ_i is a sequence of m bits, denoted as $\ell_i = (b_i^0 ... b_i^{m-1}) \in \{0,1\}^m$. Specifically, in the $4k$-bit encoding, $m = 4k$, while in the $6k$-bit encoding, $m = 6k$. By using m bits to represent a label, the cardinality of $\mathcal{L}$ is always fixed to $|\mathcal{L}| = 2^m$.

Notation Let $\ell = (\ell_1, ..., \ell_n) \in \{0,1\}^{mn}$ be the sequence of labels from the $4k$ or $6k$-bit encoding. As explained previously, each label is a sequence of $4k$ or $6k$ bits and it can be divided in k groups (or *bases*) of 4 and 6 bits, respectively:

$$\ell_i = (b_i^0 ... b_i^{4k-1}) = \Big(\overbrace{\ell_i^p \in \{0,1\}^4}^{p\text{-th base of 4 bits}} \Big)_{p=1}^{k} \qquad 4k\text{-bit label} \tag{4.1}$$

$$\ell_i = (b_i^0 ... b_i^{6k-1}) = \Big(\underbrace{\ell_i^p \in \{0,1\}^6}_{p\text{-th base of 6 bits}} \Big)_{p=1}^{k} \qquad 6k\text{-bit label} \tag{4.2}$$

As explained in the previous chapter, the bracketing encoding uses a different set of symbols (distinguished with an asterisk) to encode different relaxed-planes and then concatenates the brackets at token level to build each label (e.g. >$<^*$ in Figure 3.2). Similarly, in the $4k$ and $6k$-bit encoding, each label ℓ_i is also a concatenation of k **bases of 4 or 6 bits**, respectively, where each base ℓ_i^p en-

codes the information of the p-th subset in which A is distributed. We denote these k subsets by $(T_1, ..., T_k)$ and index them using the letter p.

For example, if a label of the $4k$-bit encoding is of the form $\ell_i = (1001\ 1100)$, then $k = 2$ (since there are two groups of 4 bits) and the first base is denoted as $\ell_1^1 = 1001$ and encodes the information of the subset T_1, while the second base is $\ell_1^2 = 1100$ and encodes the information of T_2. This is formally specified in Equations 4.1 and 4.2, where ℓ_i^p denotes the 4-bit or 6-bit base used to encode the i-th label with the arc information of the p-th subset (T_p). Note that the subscript i always denotes the position in the input sentence, while the superscript p denotes the subset T_p that is being considered.

When separating the p-th base of each label $\ell_i \in \ell$, the result is denoted as $\ell^p = (\ell_1^p, ..., \ell_n^p)$[1] and we say that ℓ^p is the p-th **subsequence** of ℓ. Note that ℓ^p is obtained after encoding T_p, and recovers T_p again through the decoding process. Thus, to fully recover A from the encoding and decoding operations, the bit linearization first distributes A into $(T_1, ..., T_k)$ and separately encodes each subset T_p to obtain ℓ^p, for $p = 1, ..., n$. Then, the decoding process independently processes each ℓ^p to rebuild a subset of arcs $\hat{T}_p$. The final set of recovered arcs is obtained as $\hat{A} = \bigcup_{p=1}^{k} \hat{T}_p$.

Figure 4.1 displays how the notation is used for the $4k$-bit encoding where $k = 3$. The labels generated in the $4k$-bit encoding with $k = 3$ have 12 bits that are grouped in bases of 4 bits (e.g. $\ell_1 = (0100\ 1101\ 1101)$ has 12 bits that are obtained after concatenating $\ell_1^1 = 0100$, $\ell_1^2 = 1101$ and $\ell_1^3 = 1101$). The encoding and decoding operations independently process each subsequence (ℓ^1, ℓ^2 and ℓ^3).

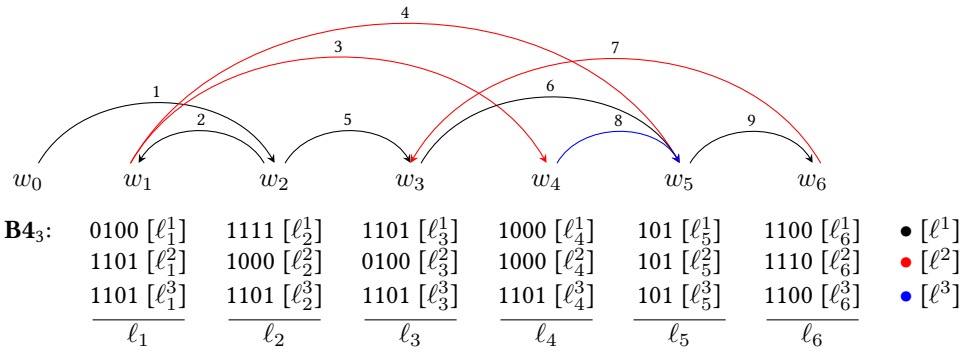


Figure 4.1: Graph example and notation of the $4k$-bit labels with $k = 3$ ($\mathbf{B4}_3$). The numbers located above each arc denote the order followed to distribute them in the k subsets. The arcs assigned to the first subset T_1 are colored in black, the arcs of the second subset T_2 are colored in red, and the arcs of the third plane T_3 are colored in blue.

4.1 $4k$-bit encoding

This section describes the encoding and decoding process of the $4k$-bit linearization. In general terms, the $4k$-bit algorithm first distributes the arcs of A in k mutually exclusive subsets $(T_1, .., T_k)$. Then, independently applies the encoding procedure to each subset.

[1] In the $4k$-bit encoding, $\ell^p \in \{0,1\}^{4n}$, while in the $6k$-bit encoding, $\ell^p \in \{0,1\}^{6n}$.

Arc distribution To effectively encode an input set of arcs (A), the encoding process first sorts the arcs of A[2] and then distributes them into k mutually exclusive subsets, denoted as $(T_1, .., T_k)$, where each subset T_p for $p = 1, .., k$, satisfies the following conditions:

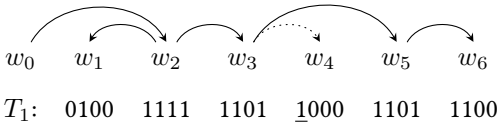
1. There are no crossing arcs in the same direction.
2. Each node must have one and only one incoming arc.

The distribution algorithm initializes the k subsets as empty and then processes the arcs of A in order. For each arc $(h \rightarrow d)$, it tries to assign it to the first subset T_1 if: (i) there is no crossing arc in T_1 in the same direction and (ii) w_d has no other incoming arc in T_1. If these conditions are not satisfied, the algorithm tries with the next subset T_2 and so on until the k subsets are processed. If it is not possible to assign the arc in any subset, this means that the graph requires more than k subsets to be fully represented with the $4k$-bit encoding and those arcs are discarded.

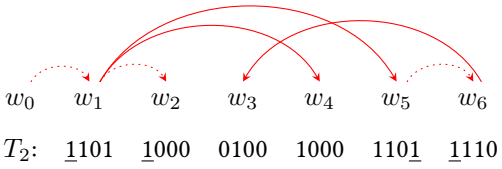
After distributing the arcs of an input set A, there is an issue with the condition (2). As defined in the properties of Section 2.2, it might happen that a node in a graph does not have any incoming arc, and thus it would not be possible to fulfill the condition (2) in any subset T_p. To solve this issue, the $4k$-bit encoding creates *artificial arcs* that are located always from the previous node to encode each subset T_p.

Figure 4.1 shows the initial distribution with different colors and Figure 4.2 shows the final distribution after creating the artificial arcs (displayed with dotted lines). See that each subset fulfills the two conditions previously exposed: (1) each subset does not have crossing arcs in the same direction (T_2 has crossing arcs but in different directions), and (2) all nodes have one and only one incoming arc in each subset by creating artificial arcs (T_1 creates only one incoming arc to w_4 but in T_3 almost all arcs are artificial).

(a) $4k$-bit encoding of T_1.



(b) $4k$-bit encoding of T_2.



(c) $4k$-bit encoding of T_3.

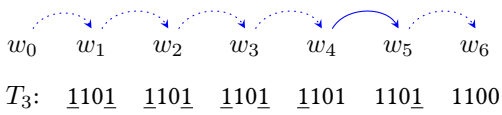


Figure 4.2: $4k$-bit encoding of the graph introduced in Figure 4.1. Artificial arcs are displayed with dotted lines, and the bits associated with them are underlined.

[2] Ordering the arcs of A is necessary to ensure that the subset assignment is deterministic. In practice, we sort the arcs of A by the left and right component.

Encoding Once the arcs of A are distributed into the k subsets (and artificial arcs are created to ensure that each node has one head), each subset T_p is independently encoded using 4 bits per label, denoted as $\ell_i^p = (b_i^{p.0} b_i^{p.1} b_i^{p.2} b_i^{p.3})$, where each bit is activated (i.e. set to 1) under the following conditions:

- $b_i^{p.0}$ is activated if w_i has a left head in T_p.
- $b_i^{p.1}$ is activated if w_i is the outermost dependent of its head in the same direction[3] in T_p.
- $b_i^{p.2}$ is activated if w_i has left dependents in T_p.
- $b_i^{p.3}$ is activated if w_i has right dependents in T_p.

See the example graph in Figure 4.1: the labels ℓ_4^2 and ℓ_5^2 of the subset T_2 (in red) are $\ell_4^2 = 1000$ and $\ell_5^2 = 110\underline{1}$. For both, the first bit ($b_4^{2.0}$ and $b_5^{2.0}$) is activated, since they share the same left head in T_2. Only the second bit of ℓ_5^2 is activated, since the node w_5 is the farthest dependent of the shared left head in T_2. Finally, only ℓ_5^2 has the last bit activated since it has a right dependent (w_6), while w_4 does not have any dependents, so the last two bits of its label ℓ_4^2 are set to 0. We have underlined the last bit of ℓ_5^2 to better remark that this bit is representing an artificial arc, although in practice we do not have this distinction.

Algorithm 3 formalizes the encoding process of the $4k$-bit encoding, The main function is ENCODE, which takes the set of arcs A, the size of the graph and the value of the hyperparameter k and returns the final label sequence $\ell = (\ell_1, ..., \ell_n) \in \{0,1\}^{4kn}$. The ENCODE-ONE function performs the actual encoding of each subset T_p and returns the p-th subsequence $\ell^p = (\ell_1^p, ..., \ell_p^n) \in \{0,1\}^{4n}$. The function RELAXED-CROSS is the same as in Algorithm 1. See that each subset T_p is passed through the CREATE-ARTIFICIAL function before encoding to create the artificial arcs using the previous position of each node with no assigned head. Lines 20-21 specify the outermost dependent condition, where the function DIR determines the direction of an arc $(h \rightarrow d)$, specifically, $\text{DIR}(h \rightarrow d) = \text{sign}(d-h)$.

Decoding Similarly to the bracketing linearization, the $4k$-bit decoding relies on a **stack-based transition system** to recover the arcs of a certain subset T_p given its corresponding subsequence of labels $\ell^p = (\ell_1^p, ..., \ell_n^p)$. The system pushes and pops elements from a buffer $\beta \subseteq \ell^p$ to two different stacks, $\sigma_{\text{L}} \in ([1,n] \times \{0,1\})^*$ for left arcs and $\sigma_{\text{R}} \subseteq [1,n]^*$ for right arcs; and creates arcs between the elements at the front of the buffer and at the top of the stacks. In the initial state, $s_0 = ([], [0], \ell^p, \emptyset)$, the left stack is empty, the right stack contains the node 0, the buffer contains the subsequence of bits corresponding to the subset T_p and the set of predicted arcs ($\hat{T}_p$) is empty. The element at the top of the buffer is the p-th base of the i-th label, $\beta^\top = \ell_i^p = (b_i^{p.0} b_i^{p.1} b_i^{p.2} b_i^{p.3})$; and the element at the top of σ_{L} is a tuple denoted as $\sigma_{\text{L}}^\top = (\sigma_{\text{L}}^d, \sigma_{\text{L}}^\omega)$, where $\sigma_{\text{L}}^d \in [1,n]$ and $\sigma_{\text{L}}^\omega \in \{0,1\}$. In fact, σ_{L}^d represents the position of the left arc that has been opened in previous labels and needs a buffer position to be closed. σ_{L}^ω instead represents whether the dependent of this left arc is the leftmost one or more elements of σ_{L} need to be popped to recover all the left arcs associated with the buffer position. The system defines the following actions depending on the bits of $\beta^\top$:

[3] Note that the second bit only considers other dependents in the same direction, so, given w_i and its associated incoming arc $(h \rightarrow i) \in T_p$, $b_i^{p.1}$ is activated if $\nexists (h \rightarrow j) \in T_p$ where $\text{sign}(h-j) = \text{sign}(h-i)$ and $|h-j| > |h-i|$.

Algorithm 3 $4k$-bit encoding algorithm.

```
1: procedure ENCODE(A, n, k)
2:     T ← DISTRIBUTE(A, k)
3:     ℓ ← (λ : i = 1, ..., n)
4:     for p = 1, ..., |T| do
5:         T_p ← CREATE-ARTIFICIAL(T_p, n)
6:         ℓ' ← ENCODE-ONE(T_p, n)
7:         for i = 1, ..., n do
8:             ℓ_i ← concat(ℓ_i, ℓ'_i)
9:     return ℓ
10:
11: procedure ENCODE-ONE(A, n)
12:     ℓ ← (0000 : i = 1, ..., n)
13:     for α ≔ (h → d) ∈ A do
14:         if h > d then
15:             ℓ^2_h ← 1
16:         else
17:             if h ≠ 0 then
18:                 ℓ^3_h ← 1
19:             ℓ^0_d ← 1
20:         if ∄α' ≔ (h → d') ∈ A :
21:           DIR(α) = DIR(α') ∧ |α| < |α'| then
22:             ℓ^1_d ← 1
23:     return ℓ
24: procedure DISTRIBUTE(A, k)
25:     T ← (∅)
26:     for α ≔ (h → d) ∈ sort(A) do
27:         p ← 1
28:         added ← false
29:         while not added and p ≤ |T| do
30:             if not (RELAXED-CROSS(α, T_p)
31:               or ASSIGNED(d, T_p)) then
32:                 T_p ← T_p ∪ {α}
33:                 added ← true
34:             else
35:                 p ← p + 1
36:         if not added and |T| < k then
37:             push(T, {a})
38:     return D
39:
40: procedure ASSIGNED(d, A)
41:     for (h' → d') ∈ A do
42:         if d' = d then
43:             return true
44:     return false
45:
46: procedure CREATE-ARTIFICIAL(A, n)
47:     for d = 1, ..., n do
48:         if not ASSIGNED(d, A) then
49:             add(A, ((d − 1) → d))
50:     return A
```

- *Resolve-left*: Adds $(i \to \sigma_{\mathrm{L}}^{d})$ to $\hat{T}_p$ and pops σ_{L}.
- *Close-left*: Adds $(i \to \sigma_{\mathrm{L}}^{d})$ to $\hat{T}_p$, pops σ_{L} and updates the third bit ($b_i^{p.2} \leftarrow 0$) of $\beta^{\top}$.
- *Resolve-right*: Adds $(\sigma_{\mathrm{R}}^{\top} \to i)$ to $\hat{T}_p$.
- *Close-right*: Adds $(\sigma_{\mathrm{R}}^{\top} \to i)$ to $\hat{T}_p$ and pops σ_{R}.
- *Open-right*: Pushes i to σ_{R}.
- *Open-left*: Pushes $(i, b_i^{p.1})$ to σ_{L}.
- *Skip*: Removes the element at the front of β.

Table 4.1 shows the $4k$-bit decoding as a deductive system and Table 4.2 shows a decoding example step-by-step of the graph displayed in Figure **??**. The $4k$-bit decoding process resembles the bracketing decoding system (Table 3.1): there are opening actions (*open-right* and *open-left*), which add elements to σ_{R} and σ_{L}, and closing actions (*resolve-right*, *close-right*, *resolve-left* and *close-left*), which add new arcs to $\hat{T}_p$ (the set o f recovered arcs) by creating connections between i (the index of the label that is located at the front of the buffer) and the top of σ_{L} or σ_{R}. Note that in the bracketing decoding system, each opening or closing action ended removing the element at the front of the buffer, thus searching for new bracket symbols in ℓ. In the $4k$-bit encoding, the *skip* transition is the only one that removes the elements of the buffer, and it must be explicitly called to force the transition system to look up for the next labels.

Transition	Precondition	Action
resolve-left	$\lvert\sigma_{\text{L}}\rvert > 0$ $(i \to \sigma_{\text{L}}^{d}) \notin \hat{T}_p$	$\frac{(\sigma_{\text{L}}\lvert(\sigma_{\text{L}}^{d}, \sigma_{\text{L}}^{\omega}), \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p \cup \{(i \to \sigma_{\text{L}}^{d})\})}(b^2 = 1, \sigma_{\text{L}}^{\omega} = 0)$
close-left	$\lvert\sigma_{\text{L}}\rvert > 0$ $(i \to \sigma_{\text{L}}^{d}) \notin \hat{T}_p$	$\frac{(\sigma_{\text{L}}\lvert(\sigma_{\text{L}}^{d}, \sigma_{\text{L}}^{\omega}), \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^10b^3\rvert\beta, \hat{T}_p \cup \{(i \to \sigma_{\text{L}}^{d})\})}(b^2 = 1, \sigma_{\text{L}}^{\omega} = 1)$
resolve-right	$\lvert\sigma_{\text{R}}\rvert > 0$ $(\sigma_{\text{R}}^{\top} \to i) \notin \hat{T}_p$	$\frac{(\sigma_{\text{L}}, \sigma_{\text{R}}\rvert\sigma_{\text{R}}^{\top}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}\rvert\sigma_{\text{R}}^{\top}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p \cup \{(\sigma_{\text{R}}^{\top} \to i)\})}(b^0b^1 = 10)$
close-right	$\lvert\sigma_{\text{R}}\rvert > 0$ $(\sigma_{\text{R}}^{\top} \to i) \notin \hat{T}_p$	$\frac{(\sigma_{\text{L}}, \sigma_{\text{R}}\rvert\sigma_{\text{R}}^{\top}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p \cup \{(\sigma_{\text{R}}^{\top} \to i)\})}(b^0b^1 = 11)$
open-right	$\sigma_{\text{R}}^{\top} \neq i$	$\frac{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}\rvert i, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}(b^3 = 1)$
open-left	$\sigma_{\text{L}}^{d} \neq i$	$\frac{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}\rvert(i, b^1), \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}(b^0 = 0)$
skip		$\frac{(\sigma_{\text{L}}, \sigma_{\text{R}}, b^0b^1b^2b^3\rvert\beta, \hat{T}_p)}{(\sigma_{\text{L}}, \sigma_{\text{R}}, \beta, \hat{T}_p)}$

Table 4.1: $4k$-bit decoding as a deductive system. For better readability, the 4 bits at the front of the buffer are represented as $b^0b^1b^2b^3$. Note that the transitions have an order preference. The *skip* transition is always executed once closing and opening actions have been performed.

We first analyze the meaning of the opening actions in Table 4.1. *Open-right* is performed when $b_i^{p.3}$ is activated, i.e. when w_i has a dependent in future positions $w_{>i}$. *Open-left* is performed when $b_i^{p.0} = 0$, i.e. when $w_i i$ has a head in future positions $w_{>i}$. Both actions are computed only once, since the preconditions ensure that, when the index i is added to one stack, it is not possible to add it again. The opening actions are creating unclosed right and left arcs: *open-right* stores the head position of a right arc (which needs to be closed with right dependents that will come in future labels of the form $\ell_{>i} = 1 \cdots$), while *open-left* stores the dependent position of a left arc (which will be resolved with a future label of the form $\ell_{>i} = \cdots 1\cdot$).

Now let's take a look to the closing transitions (*resolve-right*, *close-right*, *resolve-left* and *close-left*). *Close-right* is exactly the same as *resolve-right* but with the exception that *close-right* pops σ_{R}. This means that, when performing *close-right*, we are no longer able to recover right arcs where $\sigma_{\text{R}}^{\top}$ is the head. Thus, when conditioning *close-right* to a new label where $b_i^{p.0}b_i^{p.1} = 11$, we are effectively controlling that no arcs of the form $(\sigma_{\text{R}}^{\top} \to (> i))$ exist since $b_i^{p.1} = 1$ indicates that w_i is the rightmost dependent of its head. Alternatively, the *close-left* and *resolve-left* are almost the same: they are both performed when $b_i^{p.2} = 1$, which indicates that w_i has left dependents which positions are stored in σ_{L}, and close a left arc of the form $i \to \sigma_{\text{L}}^{d}$. However, the *close-left* action updates $b_i^{p.2} = 0$ to prevent other *resolve-left* or *close-left* actions to be performed with the current label. This occurs when $\sigma_{\text{L}}^{\omega} = 1$, which indicates that σ_{L}^{d} is the leftmost dependent of its parent, so no more arcs of the form $i \to (< \sigma_{\text{L}}^{d})$ should be added to $\hat{T}_p$.

Algorithm 4 shows the pseudocode of the $4k$-bit decoding. The main function, as in Algorithm

2, is ENCODE, which independently decodes each subsequence and joins its subset of recovered arcs to return a single set $\hat{A}$. In the main loop, the ENCODE function uses the GET-SUBSEQUENCE function to extract the p-th base of each label in ℓ. Note that $\ell^p \in \{0,1\}^{4n}$ corresponds to the p-th subsequence of bits that encodes the information of T_p. Then, the DECODE-ONE function implements the transition-based system that parses ℓ^p creating new arcs.

Algorithm 4 $4k$-bit decoding algorithm.

procedure DECODE(ℓ, k)
  $\hat{A} \leftarrow \emptyset$
  for $p = 1, ..., k$ **do**
    $\ell^p \leftarrow$ GET-SUBSEQUENCE(ℓ, p)
    $\hat{A} \leftarrow$ DECODE-ONE($\ell^p, \hat{A}$)
  return $\hat{A}$

procedure GET-SUBSEQUENCE(ℓ, p)
  $\ell^p \leftarrow (\lambda : i = 1, ..., |\ell|)$
  for $i = 1, ..., |\ell|$ **do**
    $(b_i^0, ..., b_i^{4k}) \leftarrow \ell_i$
    $\ell_i^p \leftarrow (b_i^{4(p-1)}, ..., b_i^{4p-1})$
  return ℓ^p

procedure DECODE-ONE($\ell, \hat{A}$)
  $\sigma_{\text{R}} \leftarrow [0]; \sigma_{\text{L}} \leftarrow []$
  for $\ell_i := (b^0 b^1 b^2 b^3) \in \ell$ **do**
    if $b^2 = 1 \wedge |\sigma_{\text{L}}| > 0 \wedge (i \rightarrow \sigma_{\text{L}}) \notin \hat{A}$ **then**
      $(\sigma_{\text{L}}^d, \sigma_{\text{L}}^\omega) \leftarrow \text{pop}(\sigma_{\text{L}})$
      while $\sigma_{\text{L}}^\omega \neq 1$ **do**
        $\hat{A} \leftarrow \hat{A} \cup \{(i \rightarrow \sigma_{\text{L}}^d)\}$
        $(\sigma_{\text{L}}^d, \sigma_{\text{L}}^\omega) \leftarrow \text{pop}(\sigma_{\text{L}})$
      $\hat{A} \leftarrow \hat{A} \cup \{(i \rightarrow \sigma_{\text{L}}^d)\}$
    if $b^0 = 1 \wedge |\sigma_{\text{R}}| > 0 \wedge (\sigma_{\text{R}}^\top \rightarrow i) \notin \hat{A}$ **then**
      if $b^1 = 1$ **then**
        $\hat{A} \leftarrow \hat{A} \cup \{(\text{pop}(\sigma_{\text{R}}) \rightarrow i)\}$
      else
        $\hat{A} \leftarrow \hat{A} \cup \{(\sigma_{\text{R}}^\top \rightarrow i)\}$
    if $b^3 = 1$ **then**
      $\text{push}(\sigma_{\text{R}}, i)$
    if $b^0 = 0$ **then**
      $\text{push}(\sigma_{\text{L}}, (i, b^1))$
  return $\hat{A}$

Postprocessing The $4k$-bit might create artificial arcs to consistently encode each subset. This poses a problem when recovering the original arcs from the decoding process, since there is no way to know if a right arc of the form $((i-1) \rightarrow i)$ is artificial or comes from the real input graph. To solve this problem, in practice, the artificial arcs are labeled with an special relationship type r_{NULL} that allows distinguishing them from real arcs (which are labeled with a real relationship $r \in \mathcal{R}$) and removing them in the postprocessing step.

Relation with the bracketing encoding When analyzing the information represented in each bit, one might come to the conclusion that the $4k$-bit encoding is a compact reformulation of the bracketing encoding where repeated brackets are contracted in a single symbol. In fact, there is a close relation between the brackets represented in a single label and the bits that are activated in the $4k$-bit encoding. The third $b_i^{p.2}$ and fourth $b_i^{p.3}$ bits are activated with the presence of at least one slash symbol, \ and /, respectively, since they indicate that there is left or right arc from the current node w_i. In the $4k$-bit encoding, instead of having multiple repetitions of \ or / to indicate how many dependents the node has, all of them are encoded in two bits, which indicates if there is at least one dependent in each direction. The arrow symbols > and < are encoded with the first ($b_i^{p.0}$) and second ($b_i^{p.1}$) bit. Since each susbet T_p is constrained to have a unique incoming arc for each node, in the bracketing encoding this is translated into having in each label either the arrow > or <. If the incoming arc comes from the left, the arrow used is > and the first bit is activated. If

σ_L	σ_R	$\beta^\top$	$\hat{A}$
	0	0100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*open-left*
(1,1)	0	0100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*skip*
(1,1)	0	1111	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*close-right*
(1,1)		1111	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*close-left*
		1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*open-right*
	2	1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*skip*
	2	1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*close-right*
		1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*open-right*
	3	1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 0100 1111 1101 000 1101 1100
τ			*skip*
...			

	3	000
τ		*resolve-right*
	3	1000
τ		*skip*
	3	1101
τ		*close-right*
		1101
τ		*open-right*
	5	1101
τ		*skip*
	5	1100
τ		*close-right*
		1100
τ		*skip*

Table 4.2: $4k$-bit decoding process for the subset T_1 of the graph in Figure 4.2. The continuation of the table is displayed in a second column. The updates of σ_L, σ_R and $\hat{A}$ are colored in magenta and the key elements that are used to apply a transition are colored in green. For example, the first transition is *open-left*, which is executed since the first bit of $\beta_\top = 0100$ is not activated, and σ_L is updated in the next state with $(1, 1)$.

the incoming arc comes from the right, the arrow used is < and bit is deactivated. The second bit $b_i^{p.1}$ is used to represent when the head of w_i does not have further dependents, so its position can be skipped at decoding time and removed from its respective stack. Note that, since the first bit can only represent two situations (whether a head that comes from the left or the right) there is no way to represent the absence of a head or the presence of multiple heads, so the subset T_p requires each node to have only one head.

Figure 4.4 shows a comparison of the bracketing and $4k$-bit encoding, where we can see the equivalence of each bit into bracket symbols. The arrows colored in magenta match with the ac-

tivation of the second bit, since they indicate that they are the leftmost or rightmost dependent of its head. Table 4.3 shows the decoding procedure of each encoding and how a lot of states of the bracketing and $4k$-bit decoding match each other through a different sequence of transitions. In fact, the main difference between the bracketing and $4k$-bit encoding when closing arcs is that the bracketing encoding, when closing an arc, always pops the element of the corresponding stack, thus requiring repeated positions in each stack when more arcs are connected to the index stored. The $4k$-bit encoding allows storing each position only once, and use the second bit (stored in the left stack or in the label of the right dependent) to decide whether to pop the head of σ_{R} or continue creating left arcs by popping σ_{L}.

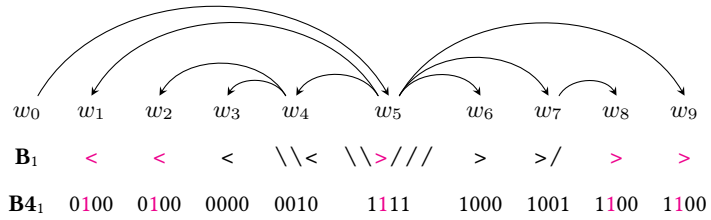


Figure 4.4: Comparison of the bracketing ($\mathbf{B}_1$) and $4k$-bit ($\mathbf{B4}_1$) encoding where $k = 1$. Arrows of the rightmost and leftmost dependents of each head are colored in magenta. See that the arrows < and > are always represented in the $4k$-bit encoding with the second bit activated (also in magenta).

$\mathbf{B}_1$			$\mathbf{B4}_1$			$\hat{A}$	$\tau(\mathbf{B}_1)$	$\tau(\mathbf{B4}_1)$
σ_L	σ_R	$\beta^\top$	σ_L	σ_R	$\beta^\top$			
	0	<		0	0100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*open-left*	*open-left* *skip*
1	0	<	(1,1)	0	0100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*open-left*	*open-left* *skip*
1,2	0	<	(1,1), (2,1)	0	0000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*open-left*	*open-left* *skip*
1,2, 3	0	\	(1,1), (2,1), (3,0)	0	0010	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*close-left*	*resolve-left*
1,2	0	\	(1,1), (2,1)	0	0010	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*close-left*	*close-left*
1	0	<	(1,1)	0	0000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*open-left*	*open-left* *skip*
1,4	0	\	(1,1), (4,0)	0	1111	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*close-left*	*resolve-left*
1	0	\	(1,1)	0	1111	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*close-left*	*close-left*
	0	>		0	1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > > 0100 0100 0000 0010 1111 1000 1001 1100 1100	*resolve-right*	*close-right*
...								

	0	///			1101	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*open-right* *open-right* *open-right*	*open-right* *skip*
	0,5, 5,5	>		5	1000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*close-right*	*resolve-right* *skip*
	0,5, 5	>		5	1001	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*close-right*	*resolve-right* *skip*
	0,5	/		5	1001	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*open-right*	*open-right* *skip*
	0,5, 7	>		5,7	1100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*close-right*	*close-right* *skip*
	0,5	>		5	1100	w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><	*close-right*	*close-right* *skip*
	0					w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8 w_9 < < < \\< \\>/// > >/ > ><		

Table 4.3: Comparison of the bracket and $4k$-bit decoding for the Figure 4.4. The first three columns show the state of the bracketing decoding system and the next three columns the $4k$-bit decoding system. Same colors as in Tables 3.2 and 4.2. The second column is the continuation of the decoding process. We show the bracketing decoding transitions in $\tau(\mathbf{B}_1)$ and the $4k$-bit decoding transition in $\tau(\mathbf{B4}_1)$.

4.2 $6k$-bit encoding

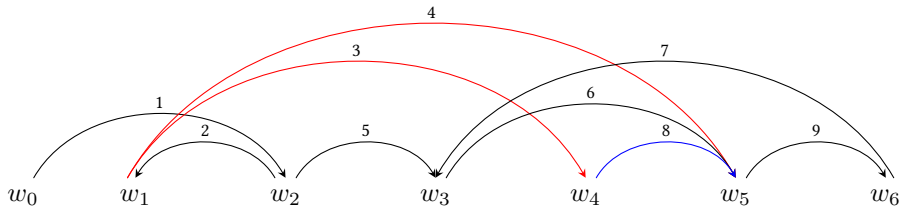
The main drawback of the $4k$-bit linearization is its limitation to graphs with one and only one parent per node, since only one bit is used to determine if the parent of w_i comes from the left or from the right, thus creating a several amount of artificial arcs (Figure 4.2). The $6k$-bit encoding solves this issue by expanding the first bit of the $4k$-bit encoding into two bits that encode whether a node has a left or right parent. Note that two bits can encode four different situations where w_i (1) has only a left parent, (2) has only a right parent, (3) has both a left and a right parent or (4) has no parent. The second bit of the $4k$-bit encoding is also expanded in other two bits that encode whether w_i is the outermost dependent of its left and right parent (if exists). By expanding the first two bits of the $4k$-bit encoding into four bits, we propose a new linearization dubbed as $6k$-bit encoding.

Similarly, the $6k$-bit encoding is restricted to represent the presence of, at most, one parent per node in each direction. Trying to encode more than one parent per direction will lead to ambiguities at decoding time since the decoding system will not be able to recognize whether it should maintain opened positions in the stacks or remove them.

Arc distribution As for the $4k$-bit encoding, the $6k$-bit encoding defines a distribution function to assign arcs to the k subsets $(T_1, ..., T_k)$ such that each subset fulfills the following conditions:

- There are no crossing arcs in the same direction.
- Each node cannot have more than one incoming arc per direction.

Figure 4.5 shows the arc distribution of the $6k$-bit encoding for the example in Figure 4.1. The number of required subsets to cover all arcs is the same as in the $4k$-bit encoding, although for other graphs k does not need to match between both algorithms. See that each subset has no crossing arcs in the same direction and nodes are only allowed to have, at most, one incoming arc per direction. For instance, w_3 in T_1 (black) has two incoming arcs but from different directions, while others, such as w_4 in T_1, might not have any incoming arc.



B6$_3$:	000110 $[\ell_1^1]$	111001 $[\ell_2^1]$	111110 $[\ell_3^1]$	000000 $[\ell_4^1]$	111000 $[\ell_5^1]$	110001 $[\ell_6^1]$	• $[\ell^1]$
	001000 $[\ell_1^2]$	000000 $[\ell_2^2]$	000000 $[\ell_3^2]$	100000 $[\ell_4^2]$	110000 $[\ell_5^2]$	000000 $[\ell_6^2]$	• $[\ell^2]$
	000000 $[\ell_1^3]$	000000 $[\ell_2^3]$	000000 $[\ell_3^3]$	001000 $[\ell_4^3]$	110000 $[\ell_5^3]$	000000 $[\ell_6^3]$	• $[\ell^3]$

Figure 4.5: Graph example of Figure 4.1 and notation of the $6k$-bit labels with $k = 3$ (**B6**$_3$). Same color legend as in Figure 4.1: the numbers located above each arc denote the distribution order, the arcs of T_1 are colored in black, T_2 in red and T_3 in blue.

Encoding As for the $4k$-bit encoding, the $6k$-bit encoding independently encodes each subset T_p using 6 bits per label, denoted as $\ell_i^p = (b_i^{p.0} b_i^{p.1} b_i^{p.2} b_i^{p.3} b_i^{p.4} b_i^{p.5})$, and then the labels are concatenated at token level to obtain $\ell_i = (\ell_i^1, ..., \ell_i^k) \in \{0,1\}^{6k}$. In the base ℓ_i^p, each bit is activated under the following conditions:

- $b_i^{p.0}$ is activated if w_i has an incoming arc from the left.
- $b_i^{p.1}$ is activated if w_i is the rightmost dependent of its head.
- $b_i^{p.2}$ is activated if w_i has right dependents.
- $b_i^{p.3}$ is activated if w_i has an incoming arc from the right.
- $b_i^{p.4}$ is activated if w_i is the leftmost dependent of its head.
- $b_i^{p.5}$ is activated if w_i has left dependents.

See that the arrangement of each bit in the $6k$-bit encoding differs from the $4k$-bit encoding. The $4k$-bit encoding uses the first two bits to encode the information of the head of w_i and the last two bits for the dependents of w_i. The $6k$-bit encoding instead uses the first three bits to encode the information of the right arcs connected to w_i (left head and right dependents) and the last three bits for the left arcs (right head and left dependents).

$$\ell_i^p = (\overbrace{b_i^{p.0} b_i^{p.1} b_i^{p.2}}^{\text{right arcs}} \; \underbrace{b_i^{p.3} b_i^{p.4} b_i^{p.5}}_{\text{left arcs}}) \tag{4.3}$$

Figure 4.5 also shows the labels obtained after encoding each subset. Let's take a look to $\ell_3^1 = 111110$, which encodes the arc information of w_3 in T_1 (black arcs). There are two right arcs $\{(2 \rightarrow 3), (3 \rightarrow 5)\}$ and one left arc $(6 \rightarrow 3)$ connected with w_3, either as a head or as a dependent. The information of the two right arcs is displayed in the first three bits, 111. The first one indicates that w_3 has a left head and the third one indicates that w_3 also has right dependents. The second bit indicates that w_3 is the rightmost dependent of its left head. The information of the left arc is encoded in the second group of bits, 110, where the fourth and fifth bits activated have an analogous meaning as the first and second bit but for the left arcs: w_3 has a right head and it is its leftmost dependent. The last bit is not activated, which means that there are no left dependents for w_3.

Algorithm 5 shows the formal definition of the $6k$-bit encoding process. The main function is ENCODE, which takes the input set of arcs A, the graph's size and the value of the hyperparameter k. First, it distributes A into k subsets using the function DISTRIBUTE, which processes the arcs of A in order and tries to add each arc to the lowest valid subset. To ensure that adding a new arc α in a subset T_p is supported by the encoding, the algorithm uses the function RELAXED-CROSS (inherited from Algorithm 1) and RELAXED-ASSIGNED, which ensure that there are no arcs in T_p that cross α or have the same dependent in the same direction. Once the arcs are distributed, each subset is encoded with the function ENCODE-ONE, which processes the arcs activating the bits in the label of its corresponding head and dependent. The function IS-OUTERMOST allows identifying if the

dependent of an arc is the outermost dependent of its head in the same direction. The ENCODE-ONE function returns a sequence of labels of the form $\ell^p \in \{0,1\}^{6n}$, and it is concatenated at token level to return $\ell \in \{0,1\}^{6kn}$.

Algorithm 5 $6k$-bit encoding algorithm.

```
1:  procedure ENCODE(A, n, k)
2:      D ← DISTRIBUTE(A, k)
3:      ℓ ← (λ : i = 1, ..., n)
4:      for p = 1, ..., |D| do
5:          ℓ' ← ENCODE-ONE(T_p, n)
6:          for i = 1, ..., n do
7:              ℓ_i ← concat(ℓ_i, ℓ'_i)
8:      return ℓ
9:
10: procedure ENCODE-ONE(A, n)
11:     ℓ ← (000000 : i = 1, ..., n)
12:     for α := (h → d) ∈ A do
13:         if h > d then
14:             ℓ_d^3 ← 1
15:             ℓ_h^5 ← 1
16:             ι ← 4
17:         else
18:             if h ≠ 0 then
19:                 ℓ_h^2 ← 1
20:             ℓ_d^0 ← 1
21:             ι ← 1
22:         if IS-OUTERMOST(α, A) then
23:             ℓ_d^ι ← 1
24:     return ℓ
```

```
1:  procedure DISTRIBUTE(A, k)
2:      T ← (∅)
3:      for a := (h → d) ∈ sort(A) do
4:          p ← 1
5:          added ← false
6:          while not added and p ≤ |D| do
7:              if not (RELAXED-CROSS(a, T_p)
8:                or RELAXED-ASSIGNED(d, D_p)) then
9:                  T_p ← T_p ∪ {a}
10:                 added ← true
11:             else
12:                 p ← p + 1
13:         if not added and |D| < k then
14:             push(D, {a})
15:     return D
16:
17: procedure RELAXED-ASSIGNED((h → d), A)
18:     for α' := (h' → d') ∈ A do
19:         if d' = d ∧ DIR(h → d) = DIR(α') then
20:             return true
21:     return false
22:
23: procedure IS-OUTERMOST((h → d), A)
24:     for α' := (h → d') ∈ A : d' ≠ d do
25:         if DIR(α') = DIR(h → d) ∧ |h − d| < |α'| then
26:             return false
27:     return true
```

Decoding The $6k$-bit decoding process relies on a **transition system** similar to the $4k$-bit encoding. The sequence of labels ℓ is factorized into the k subsequences $(\ell^1, ..., \ell^k)$. Each subsequence ℓ^p, for $p = 1, .., k$ is independently processed by the transition system to recover the corresponding subset of arcs $\hat{T}_p$.

The transition system has the same components as the ones described in the $4k$-bit decoding. The buffer $\beta \subseteq \ell^p$ stores the labels that have not been parsed yet. Two stacks, $\sigma_{\text{L}} \in ([1,n] \times \{0,1\})^*$ and $\sigma_{\text{R}} \in [1,n]^*$, store the opened positions of the left and right arcs that need to be resolved with future labels. The system is initialized as in the $4k$-bit decoding, so $s_0 = ([], [0], \ell^p, \emptyset)$. The element at the top of the buffer is the p-th base of the i-th label, denoted as $\beta^\top := \ell_i^p = (b_i^{p.0} b_i^{p.1} b_i^{p.2} b_i^{p.3} b_i^{p.4} b_i^{p.5})$; and $\sigma_{\text{L}}^\top$ is also the tuple $(\sigma_{\text{L}}^d, \sigma_{\text{L}}^\omega) \in [1,n] \times \{0,1\}$ that stores the opened position of a left arc and the indicator of whether the dependent σ_{L}^d is the leftmost dependent of its head or not.

The system defines the same actions as the $4k$-bit decoding system:

- *Resolve-left*: Adds $(i \rightarrow \sigma_{\text{L}}^d)$ to $\hat{T}_p$ and pops σ_{L}.
- *Close-left*: Adds $(i \rightarrow \sigma_{\text{L}}^d)$ to $\hat{T}_p$, pops σ_{L} and updates the last bit ($b_i^{p.5} \leftarrow 0$) of $\beta^\top$.
- *Resolve-right*: Adds $(\sigma_{\text{R}}^\top \rightarrow i)$ to $\hat{T}_p$.

- *Close-right*: Adds $(\sigma_{\mathrm{R}}^{\top} \rightarrow i)$ to $\hat{T}_p$ and pops σ_{R}.
- *Open-right*: Pushes i to σ_{R}.
- *Open-left*: Pushes $(i, b_i^{p.1})$ to σ_{L}.
- *Skip*: Removes the element at the front of β.

Table 4.4 defines the $6k$-bit decoding as a deductive system. Although the $4k$ and $6k$-bit decoding share the same set of actions, the premises differ from each other. The *resolve-left* and *close-left* actions are performed when the last bit ($b_i^{p.5}$) is activated. The fifth bit ($b_i^{p.4}$) is the one introduced in the left stack to track whether w_i is the leftmost dependent of its right head.

Transition	Precondition	Action
resolve-left	$\lvert\sigma_{\mathrm{L}}\rvert > 0$ $(i \rightarrow \sigma_{\mathrm{L}}^{d}) \notin \hat{T}_p$	$\frac{(\sigma_{\mathrm{L}}\lvert(\sigma_{\mathrm{L}}^{d}, \sigma_{\mathrm{L}}^{\omega}), \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p \cup \{(i \rightarrow \sigma_{\mathrm{L}}^{d})\})}(b^5 = 1, \sigma_{\mathrm{L}}^{\omega} = 0)$
close-left	$\lvert\sigma_{\mathrm{L}}\rvert > 0$ $(i \rightarrow \sigma_{\mathrm{L}}^{d}) \notin \hat{T}_p$	$\frac{(\sigma_{\mathrm{L}}\lvert(\sigma_{\mathrm{L}}^{d}, \sigma_{\mathrm{L}}^{\omega}), \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^40\lvert\beta, \hat{T}_p \cup \{(i \rightarrow \sigma_{\mathrm{L}}^{d})\})}(b^5 = 1, \sigma_{\mathrm{L}}^{\omega} = 1)$
resolve-right	$\lvert\sigma_{\mathrm{R}}\rvert > 0$ $(\sigma_{\mathrm{R}}^{\top} \rightarrow i) \notin \hat{T}_p$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}\lvert\sigma_{\mathrm{R}}^{\top}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}\lvert\sigma_{\mathrm{R}}^{\top}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p \cup \{(\sigma_{\mathrm{R}}^{\top} \rightarrow i)\})}(b^0b^1 = 10)$
close-right	$\lvert\sigma_{\mathrm{R}}\rvert > 0$ $(\sigma_{\mathrm{R}}^{\top} \rightarrow i) \notin \hat{T}_p$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}\lvert\sigma_{\mathrm{R}}^{\top}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p \cup \{(\sigma_{\mathrm{R}}^{\top} \rightarrow i)\})}(b^0b^1 = 11)$
open-right	$\sigma_{\mathrm{R}}^{\top} \neq i$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}\lvert i, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}(b^2 = 1)$
open-left	$\sigma_{\mathrm{L}}^{d} \neq i$	$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}\lvert(i, b^4), \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}(b^3 = 1)$
skip		$\frac{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, b^0b^1b^2b^3b^4b^5\lvert\beta, \hat{T}_p)}{(\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}}, \beta, \hat{T}_p)}$

Table 4.4: $6k$-bit decoding as a deductive system. Same notation as in Table 4.1.

Table 4.5 shows the $6k$-bit decoding process for the first subset T_1 (black arcs) displayed in Figure 4.5. See that the behavior is really similar to the $4k$-bit decoding (Table 4.2) but focusing on different components of the system to perform an action. See also that the $6k$-bit decoding allows skipping positions without creating any arc (for the label $\ell_4^1 = 000000$).

Algorithm 6 shows the pseudocode of the $6k$-bit decoding. Note that it is almost the same as the $4k$-bit decoding (Algorithm 4) – in fact, the IS-VALID function is exactly the same. Only the positions of some bits change: b^2 in Algorithm 4 is replaced by b^5 (line 25), b^3 is replaced by b^2 in line 36, $b^0 = 0$ is replaced by $b^3 = 1$ in line 38 and b^1 is replaced by b^4 in line 39.

σ_L	σ_R	$\beta^\top$	$\hat{A}$
	0	000110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_1			*open-left*
(1,1)	0	000110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_2			*skip*
(1,1)	0	111001	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_3			*close-left*
	0	111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_4			*close-right*
		111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_5			*open-right*
	2	111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_6			*skip*
	2	111110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_7			*close-right*
		111110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_8			*open-right*
	3	111110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_9			*open-left*
...			

...			
(3,1)	3	111110	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{10}			*skip*
(3,1)	3	000000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{11}			*skip*
(3,1)	3	111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{12}			*close-right*
(3,1)		111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{13}			*open-right*
(3,1)	5	111000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{14}			*skip*
(3,1)	5	110001	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{15}			*close-left*
	5	110000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{16}			*close-right*
		110000	w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001
τ_{17}			*skip*
			w_0 w_1 w_2 w_3 w_4 w_5 w_6 000 111 111 000 111 110 110 001 110 000 000 001

Table 4.5: $6k$-bit decoding for T_1 in Figure 4.5. Same notation as in Table 4.2. For space limitation, the 6 bits in each base are separated in two groups (the first three bits are displayed in the first row and the last three bits in the second row).

Algorithm 6 $6k$-bit decoding algorithm.

procedure DECODE(ℓ, k)
  $\hat{A} \leftarrow \emptyset$
  for $p = 1, ..., k$ **do**
    $\ell^p \leftarrow$ GET-SUBSEQUENCE(ℓ, p)
    $\hat{A} \leftarrow$ DECODE-ONE($\ell^p, \hat{A}$)
  return $\hat{A}$

procedure GET-SUBSEQUENCE(ℓ, p)
  $\ell^p \leftarrow (\lambda : i = 1, ..., |\ell|)$
  for $i = 1, ..., |\ell|$ **do**
    $(b_i^0, ..., b_i^{6k}) \leftarrow \ell_i$
    $\ell_i^p \leftarrow (b_i^{6(p-1)}, ..., b_i^{6p-1})$
  return ℓ^p

procedure DECODE-ONE($\ell, \hat{A}$)
  $\sigma_{\mathrm{R}} \leftarrow [0]$; $\sigma_{\mathrm{L}} \leftarrow []$
  for $\ell_i := (b^0 b^1 b^2 b^3 b^4 b^5) \in \ell$ **do**
    if $b^5 = 1 \wedge |\sigma_{\mathrm{L}}| > 0 \wedge (i \to \sigma_{\mathrm{L}}^d) \notin \hat{A}$ **then**
      $(\sigma_{\mathrm{L}}^d, \sigma_{\mathrm{L}}^\omega) \leftarrow \mathrm{pop}(\sigma_{\mathrm{L}})$
      while $\sigma_{\mathrm{L}}^\omega \neq 1$ **do**
        $\hat{A} \leftarrow \hat{A} \cup \{(i \to \sigma_{\mathrm{L}}^d)\}$
        $(\sigma_{\mathrm{L}}^d, \sigma_{\mathrm{L}}^\omega) \leftarrow \mathrm{pop}(\sigma_{\mathrm{L}})$
      $\hat{A} \leftarrow \hat{A} \cup \{(i \to \sigma_{\mathrm{L}}^d)\}$
    if $b^0 = 1 \wedge |\sigma_{\mathrm{R}}| > 0 \wedge (\sigma_{\mathrm{R}}^\top \to i) \notin \hat{A}$ **then**
      if $b^1 = 1$ **then**
        $\hat{A} \leftarrow \hat{A} \cup \{(\mathrm{pop}(\sigma_{\mathrm{R}}) \to i)\}$
      else
        $\hat{A} \leftarrow \hat{A} \cup \{(\sigma_{\mathrm{R}}^\top \to i)\}$
    if $b^2 = 1$ **then**
      $\mathrm{push}(\sigma_{\mathrm{R}}, i)$
    if $b^3 = 1$ **then**
      $\mathrm{push}(\sigma_{\mathrm{L}}, (i, b^4))$
  return $\hat{A}$

Chapter 5

Framework and Experiments

This chapter describes the experimental methodology used to evaluate the performance of our SL approach against SoTA graph parsers. Following previous studies, we train a neural-based model on annotated treebanks to predict label sequences as a tagging task. Section 5.1 details the neural architecture used as the backbone for our parsers. Section 5.2 formalizes the evaluation metrics commonly used in graph parsing to compare our SL methods with baseline systems. Section 5.3 outlines the specific training configurations for the models and Section 5.4 details the data used and its properties to run the experiments.

5.1 Neural framework

Our neural framework adheres to the encoder-decoder architecture, extensively applied for tagging tasks [23, 24, 25, 47]. Figures 5.1 and 5.2 show a visualization of our pipeline at inference and training time, respectively. For both, the encoder module (green box) contextualizes an input sentence[1] $W = (w_0, w_1, ..., w_n) \in \mathcal{V}^{n+1}$ through recurrent or Transformer-based layers and returns a sequence of contextualized word embeddings, $\mathbf{H} = (\mathbf{h}_0, \mathbf{h}_1, ..., \mathbf{h}_n) \in \mathbb{R}^{(n+1)\times d_h}$. The decoder (blue boxes) is conformed by two FFNs with a LeakyReLU [85] activation. The first FFN, denoted as $\text{FFN}_\ell : \mathbb{R}^{d_h} \rightarrow \mathbb{R}^{|\mathcal{L}|}$, learns the gold labels $(\ell_1, ..., \ell_n)$ by projecting each contextualized embedding $\mathbf{h}_i$, for $i = 1, .., n$, into the probability distribution space of $\mathcal{L}$. The predicted label $\hat{\ell}_i$ is obtained with the $\arg\max()$ operator over $\text{FFN}_\ell(\mathbf{h}_i)$.

Once the predicted label sequence is computed with FFN_ℓ, the decoding process is executed to obtain $\hat{A}$ (yellow lines). To assign relationship types to each arc $\hat{\alpha} := (h \rightarrow d) \in \hat{A}$, the contextualized representations of the words w_h and w_d are concatenated, building an arc embedding of the form $(\mathbf{h}_h; \mathbf{h}_d)$ and fed to the second FFN, denoted as $\text{FFN}_r : \mathbb{R}^{2d_h} \rightarrow \mathbb{R}^{|\mathcal{R}|}$. The output, $\text{FFN}_r(\mathbf{h}_h; \mathbf{h}_d)$, represents the probability distribution over $\mathcal{R}$ for the relationship type of the arc $(h \rightarrow d)$. By passing each arc embedding through FFN_r, the result is a labeled predicted graph (purple lines) that is evaluated against the original input graph.

At training time, the full neural architecture is optimized via gradient descent with the cross-entropy loss between (i) the gold and predicted labels and (ii) the gold and predicted relations. To

[1] Note that w_0 is not a proper token from the original sentence, but an special token (such as the [CLS] token used in BERT) that represents the root context of the sentence.

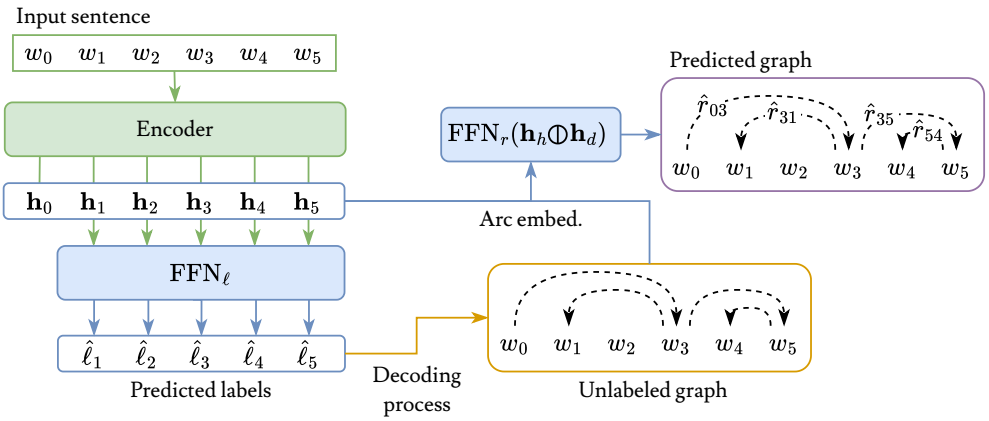


Figure 5.1: Neural framework proposed for our SL approach at inference time.

pair each gold relation with a prediction from the model, the real arcs are used instead of decoding the predicted labels. When using the real arcs to feed arc embeddings to FFN_r, it is possible to apply the cross-entropy loss between the output of FFN_r and the real relationship type (Figure 5.2).

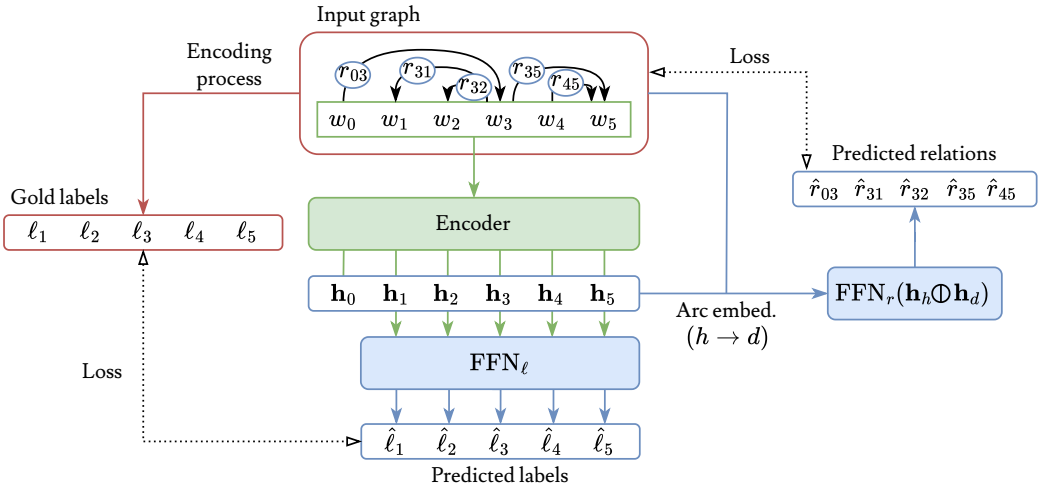


Figure 5.2: Neural framework proposed for our SL approach at training time.

5.2 Evaluation metrics

The performance of our SL-based graph parser is evaluated by comparing the set of predicted labeled arcs, $\hat{A}$, with the real set of arcs A. We adopted the evaluation metrics proposed in the SemEval 2015 Task 18 [16], which include the unlabeled (and labeled) precision, recall, f-score and exact match.

Let UTP (unlabeled true positives) be the number of correctly predicted unlabeled arcs (arcs of $\hat{A}$ that appear in A without considering the relationship label r) and TP (true positives) be the the number of correctly predicted labeled arcs. Formally,

$$\text{UTP} = |\{(h \xrightarrow{r'} d) \in \hat{A} : (h \xrightarrow{r} d) \in \hat{A},\ r' \in \mathcal{R}\}|, \qquad \text{TP} = |A \cap \hat{A}| \tag{5.1}$$

The unlabeled precision (UP), recall (UR), f-score (UF) and exact match (UM) are defined in Equation 5.2. The UP and UR evaluate the ratio of correct unlabeled arcs against the number of predicted arcs and the number of real arcs. The UF leverages the harmonic mean between UP and UR. The exact match assess whether all the arcs predicted are in A, meaning that the full graph is correctly predicted without considering the relationship types.

$$\text{UP} = \frac{\text{UTP}}{|\hat{A}|}, \quad \text{UR} = \frac{\text{UTP}}{|A|}, \quad \text{UF} = 2\frac{\text{UP} \cdot \text{UR}}{\text{UP} + \text{UR}}, \quad \text{UM} = \mathbb{I}(\text{UTP} = |A| = |\hat{A}|) \tag{5.2}$$

The labeled metrics are displayed in Equation 5.3. Each labeled metric is an analogy of its unlabeled counterpart using the TP value instead of UTP. In the case of the exact match, the LM takes value 1 if all the predicted graph is exactly the same as the input graph.

$$\text{LP} = \frac{\text{TP}}{|\hat{A}|}, \quad \text{LR} = \frac{\text{TP}}{|\hat{A}|}, \quad \text{LF} = 2\frac{\text{LP} \cdot \text{LR}}{\text{LP} + \text{LR}}, \quad \text{LM} = \mathbb{I}(\hat{A} = A) \tag{5.3}$$

5.3 Training configuration

In order to accurately assess the performance of our SL approaches, we conducted multiple experiments varying the model configuration and the encoder architecture: training from scratch a 4-layered BiLSTM (Section 2.1.2) and finetuning XLM-RoBERTa$^{\text{L}}$ [72] or XLNet$^{\text{L}}$ [54] (Section 2.1.4) for English treebanks. The hidden dimension of the BiLSTM is fixed to $d_h = 400$ and for the pretrained LLMs the original embedding size is maintained ($d_h = 1024$). For the bracketing, $4k$-bit and $6k$-bit encodings, we also varied the hyperparameter k, testing different values to increase the coverage of each linearization. For the bracketing encoding, k denotes the number of relaxed-planes supported, so increasing its value allows recovering more complex graphs (Section 3.2). We conducted experiments for the bracketing encoding with $k = 2$ and $k = 3$. For the $4k$-bit and $6k$-bit linearizations, the hyperparemeter k modulates the number of subsets in which A is distributed (Sections 4.1 and 4.2), thus larger values ensure recovering more arcs, so we tested k for $\{2, 3, 4\}$.

All models were optimized with AdamW [86] (the learning rate was fixed to $\eta = 10^{-3}$ when the BiLSTM was set as encoder or $\eta = 10^{-5}$ when an LLM was finetuned) during 200 epochs with early stopping over the development set.

Baseline We selected the biaffine parser [17] (Section 2.2.1) to compare the performance of our graph linearizations against the SoTA in graph parsing. Originally, the biaffine parser uses a BiLSTM-based encoder, but for a fair comparison, we report results with XLM-RoBERTa and XLNet as encoder. We relied on SuPar 1.14 implementation to train again the biaffine parser with the same training configuration as our SL-based models.

5.4 Datasets

We train our models on the SemEval 2015 Task 18 (SDP) [16] and IWPT 2021 Shared Task (IWPT) [30] datasets. As specified in Section 1.3, both datasets are open-source and contain multilingual graph

annotations in the SDP and Enhanced CoNLL formats. In case of the SDP dataset, its annotations are constrained to acyclic graphs.

Tables 5.1 and 5.2 show some statistics of the collected treebanks. See that the SDP corpus (Table 5.1) does not contain cycles. Almost the 99% of the graphs are have 1 or 2 relaxed planes, with makes the bracketing encoding with $k = 2$ sufficient to cover almost all graphs. The PSD annotations are the ones with more relaxed 3-planar graphs. The Chinese treebank seems to have denser graphs, since the number of heads per node is greater than one and the number of arcs per graph surpasses the average length.

The statistics of the IWPT corpus (Table 5.2) show that all treebanks contain a considerable amount of graphs with at least one cycle. Key characteristics such as the average length and the average number of relaxed planes per graph varies between languages. The Arabic treebanks contains the longer sequences. Lituanian has the higher number of relaxed planes per graph. The 96% of the graphs are covered with 1 or 2 relaxed planes. In general terms, the graphs from the IWPT dataset are denser than the ones from the SDP corpus, since in almost all cases, the number of heads per node is greater than 1.

	#sents	n	**#cycles**	**#relaxed-planes**					**h/n**	**d/n**	**#arcs**	**#relaxed-planes**	**#roots**
				1	**2**	**3**	**4**	**5**					
DM$_{en}$	33964	22.52	0	86.28	13.68	0.05	0.00	0.00	0.78	0.74	17.68	1.14	1.00
	1692	22.28	0	86.94	13.06	0.00	0.00	0.00	0.79	0.74	17.55	1.13	1.00
	1410	22.66	0	84.11	15.89	0.00	0.00	0.00	0.77	0.73	17.60	1.16	1.00
	1849	17.08	0	88.64	11.20	0.16	0.00	0.00	0.75	0.70	13.11	1.12	0.99
PAS$_{en}$	33964	22.52	0	83.70	16.20	0.10	0.00	0.00	1.01	0.95	22.96	1.16	1.00
	1692	22.28	0	86.52	13.42	0.06	0.00	0.00	1.00	0.95	22.53	1.14	1.00
	1410	22.66	0	82.91	17.02	0.07	0.00	0.00	1.01	0.96	23.15	1.17	1.00
	1849	17.08	0	83.02	16.77	0.22	0.00	0.00	0.99	0.92	17.35	1.17	1.00
PSD$_{en}$	33964	22.52	0	77.51	20.99	1.44	0.05	0.01	0.70	0.66	15.80	1.24	1.13
	1692	22.28	0	76.71	22.16	0.95	0.12	0.06	0.70	0.66	15.74	1.25	1.12
	1410	22.66	0	77.38	21.21	1.28	0.14	0.00	0.69	0.66	15.79	1.24	1.14
	1849	17.08	0	81.23	17.79	0.87	0.11	0.00	0.67	0.62	11.57	1.20	1.26
PSD$_{cs}$	40047	23.45	0	74.22	24.04	1.66	0.08	0.00	0.77	0.73	18.34	1.28	1.23
	2010	22.99	0	75.32	23.33	1.24	0.05	0.05	0.78	0.73	17.97	1.26	1.18
	1670	22.99	0	73.35	24.67	1.98	0.00	0.00	0.76	0.72	17.71	1.29	1.20
	5226	16.82	0	78.66	19.15	2.01	0.17	0.00	0.78	0.71	13.19	1.24	1.28
PAS$_{zh}$	25896	22.43	0	75.60	24.12	0.28	0.00	0.00	1.02	0.94	22.95	1.25	1.00
	2440	27.95	0	73.16	26.60	0.25	0.00	0.00	1.02	0.95	28.60	1.27	1.00
	8976	23.89	0	75.12	24.64	0.23	0.00	0.00	1.02	0.95	24.47	1.25	1.00
	8976	23.89	0	75.12	24.64	0.23	0.00	0.00	1.02	0.95	24.47	1.25	1.00

Table 5.1: Treebank statistics for the SDP dataset. Number of sentences (**#sents**), sentences with cycles (**#cycles**), percentage of graph with k-relaxed planes (**#planes**, average sentence length (n), average number of heads and dependents per node (**h/n** and **d/n** respectively), average number of arcs (**#arcs**), planes (**#relaxed-planes**) and roots (**#roots**) per graph.

	#sents	n	#cycles	#relaxed-planes					h/n	d/n	#arcs	#relaxed-planes	#roots
				1	2	3	4	5					
ar	6075	36.85	1386	65.32	32.49	1.86	0.30	0.02	1.05	1.00	39.22	1.37	1.08
	909	33.27	225	69.97	28.49	1.32	0.22	0.00	1.05	1.00	35.24	1.32	1.06
	680	41.56	178	64.26	33.82	1.76	0.15	0.00	1.05	1.00	44.11	1.38	1.06
bg	8907	13.96	1111	90.14	9.68	0.18	0.00	0.00	1.02	0.93	14.38	1.10	1.00
	1115	14.43	141	89.69	10.22	0.09	0.00	0.00	1.02	0.94	14.86	1.10	1.00
	1116	14.09	136	90.59	9.41	0.00	0.00	0.00	1.02	0.93	14.49	1.09	1.00
cs	102133	17.42	16451	65.04	33.58	1.28	0.09	0.01	1.05	0.97	18.66	1.36	1.23
	11182	16.72	1739	66.28	32.67	1.00	0.04	0.01	1.05	0.96	17.85	1.35	1.23
	13067	16.84	2114	65.92	33.14	0.88	0.05	0.01	1.05	0.96	17.93	1.35	1.23
en	18213	16.85	1533	81.29	18.52	0.19	0.00	0.00	1.04	0.93	17.69	1.19	1.00
	2845	14.52	180	83.69	16.10	0.21	0.00	0.00	1.03	0.91	15.25	1.17	1.00
	3972	15.69	355	84.39	15.46	0.15	0.00	0.00	1.03	0.92	16.42	1.16	1.00
et	27470	13.79	2560	95.27	4.72	0.01	0.00	0.00	1.01	0.91	13.92	1.05	1.03
	3868	13.85	382	92.63	7.37	0.00	0.00	0.00	1.01	0.91	14.04	1.07	1.07
	4127	14.94	377	91.03	8.92	0.05	0.00	0.00	1.01	0.91	15.16	1.09	1.08
fi	12217	13.33	1855	78.90	19.19	1.77	0.12	0.02	1.06	0.96	14.41	1.23	1.00
	1364	13.42	203	78.30	20.01	1.32	0.29	0.07	1.06	0.96	14.57	1.24	1.00
	2555	14.44	414	84.54	14.21	1.17	0.08	0.00	1.05	0.96	15.24	1.17	1.00
fr	2231	22.64	546	80.55	19.23	0.18	0.04	0.00	1.04	0.95	23.77	1.20	1.00
	412	24.28	112	80.83	18.93	0.24	0.00	0.00	1.04	0.97	25.51	1.19	1.00
	2745	12.45	193	94.72	5.21	0.07	0.00	0.00	1.02	0.92	12.76	1.05	1.00
it	13121	21.04	2245	85.73	14.17	0.10	0.00	0.00	1.03	0.96	21.95	1.14	1.00
	564	21.11	104	87.06	12.94	0.00	0.00	0.00	1.03	0.96	21.94	1.13	1.00
	482	21.61	88	86.31	13.49	0.21	0.00	0.00	1.03	0.96	22.49	1.14	1.00
lt	2341	20.35	217	51.82	44.55	3.03	0.47	0.13	1.10	1.01	22.74	1.53	1.37
	617	18.74	69	58.67	39.87	1.46	0.00	0.00	1.07	0.99	20.37	1.43	1.29
	684	15.86	42	53.22	44.01	2.34	0.44	0.00	1.08	0.99	17.49	1.50	1.36
lv	10156	16.50	1526	73.50	24.25	2.10	0.13	0.01	1.06	0.97	17.76	1.29	0.99
	1664	15.60	194	75.12	22.84	1.86	0.18	0.00	1.04	0.95	16.68	1.27	0.98
	1823	14.48	207	78.72	19.69	1.37	0.16	0.05	1.02	0.93	15.32	1.23	0.96
nl	18051	14.46	1814	84.24	15.17	0.58	0.01	0.00	1.02	0.92	14.95	1.16	1.00
	1394	16.45	122	84.86	14.49	0.65	0.00	0.00	1.03	0.93	17.02	1.16	1.00
	1471	15.37	107	82.39	17.06	0.54	0.00	0.00	1.03	0.91	16.01	1.18	1.00
pl	31496	12.27	1933	79.65	19.50	0.80	0.04	0.00	1.04	0.94	13.04	1.21	1.11
	3960	12.07	256	80.15	19.09	0.76	0.00	0.00	1.04	0.93	12.82	1.21	1.11
	4942	13.18	382	77.22	22.04	0.67	0.08	0.00	1.05	0.95	14.05	1.24	1.13
ru	48814	17.83	3979	67.29	32.30	0.41	0.00	0.00	1.04	0.97	18.80	1.33	1.23
	6584	18.00	511	65.31	34.23	0.44	0.02	0.00	1.05	0.97	19.04	1.35	1.28
	6491	18.08	507	65.23	34.42	0.35	0.00	0.00	1.05	0.97	19.07	1.35	1.25
sk	8483	9.50	469	78.52	21.10	0.38	0.00	0.00	1.04	0.91	9.96	1.22	1.19
	1060	12.01	105	81.13	18.40	0.47	0.00	0.00	1.05	0.93	12.70	1.19	1.14
	1061	12.00	117	77.38	22.34	0.28	0.00	0.00	1.05	0.93	12.79	1.23	1.19
sv	4303	15.49	631	85.99	13.97	0.05	0.00	0.00	1.05	0.95	16.42	1.14	1.00
	504	19.44	86	76.19	23.61	0.20	0.00	0.00	1.06	0.99	20.85	1.24	1.00
	2219	17.78	425	84.68	15.19	0.14	0.00	0.00	1.05	0.98	18.81	1.15	1.00
ta	400	15.82	1	97.75	2.25	0.00	0.00	0.00	1.02	0.95	16.19	1.02	1.00
	80	15.79	22	98.75	1.25	0.00	0.00	0.00	1.05	0.97	16.68	1.01	1.01
	120	16.57	38	98.33	1.67	0.00	0.00	0.00	1.03	0.96	17.29	1.02	1.00
uk	5496	16.81	746	63.36	35.94	0.69	0.02	0.00	1.06	0.97	18.10	1.37	1.25
	672	18.71	143	61.61	37.80	0.60	0.00	0.00	1.07	0.99	20.20	1.39	1.20
	892	19.19	121	65.25	33.86	0.90	0.00	0.00	1.05	0.96	20.61	1.36	1.17

Table 5.2: Treebank statistics for the IWPT dataset. Same notation as in Table 5.1

Chapter 6
Results

This chapter presents the results of our experimental study, evaluating the proposed graph linearizations in terms of performance against the biaffine baseline and analyzing the coverage of the encodings (Section 6.1); and assesses the impact of each linearization method on the system's speed (Section 6.2)

6.1 Performance evaluation

Tables 6.1, 6.2 and 6.3 present the UF and LF score in each treebank using XLM-RoBERTa (or XLNet for English treebanks) as encoder. For the SDP dataset, we report results on both the in-distribution and out-of-distribution test sets (Section 1.3), which provide a better reference for the parser's performance across different data distributions. The **coverage** is also reported to provide a reference of the amount of arcs that each graph linearization is able to reconstruct. In this context, the coverage is defined as the subset of arcs in a graph that are recoverable when applying the decoding function to the sequence of labels produced during encoding. For instance, when the bracketing encoding with $k = 1$ is applied to a relaxed 2-planar graph, the subset of recoverable arcs is not the original set of arcs, since crossing arcs in the same direction are skipped. In Tables 6.1-6.3, the coverage is measured in terms of f-score (CF) by computing as the unlabeled f-score between A and the encoding-decoding reconstruction, $\delta(\varepsilon(A))$, with an specific linearization. See that, for all treebanks, the CF increases with k for the bracketing (B), and bit (B4 and B6) encodings, since more relaxed-planes or arc subsets are supported.

Table 6.1 shows that our SL parsers outperform the baseline in 2 out of 5 treebanks in terms of UF score and in all treebanks in terms of LF score for the in-distribution sets. For the out-of-distribution sets (Table 6.2), the biaffine system only outperforms our linearizations in the Czech treebank in terms of the UF score. Table 6.3 breaks down the performance in all treebanks of the IWPT dataset. The SL approach outperforms biaffine in both UF and LF scores for 3 out of 17 treebanks (Bulgarian, Dutch, and Slovak) and in one of these metrics for another 3 treebanks (French, Italian, and Tamil). Overall, the denser graphs in the IWPT dataset negatively impact the SL-based approaches, as higher values of k are required to cover more graphs or a larger number of labels is generated.

The bracketing and $6k$-bit encoding stand out as the best performing SL approaches. Only in the Polish treebank, the $4k$-bit encoding achieves the highest UF and LF scores, though it still remains

	DM_{en}			PAS_{en}			PSD_{en}			PSD_{cs}			PAS_{zh}		
	UF	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**
A	88.66	87.94	100	86.66	85.29	100	89.56	79.19	100	90.04	85.33	100	76.20	74.02	100
R	91.92	91.23	100	90.29	88.86	100	89.74	79.39	100	89.60	84.92	100	78.89	76.66	100
B_2	95.16	94.46	99.95	**95.82**	**94.31**	99.98	92.31	81.80	99.73	92.75	88.14	99.77	87.74	85.39	99.97
B_3	94.63	93.75	100	95.73	94.21	100	92.33	81.65	99.98	92.74	88.02	99.99	**87.78**	**85.50**	100
$B4_2$	86.45	85.84	90.45	79.84	78.82	82.46	**92.87**	81.96	99.58	92.88	88.24	99.53	77.54	75.53	87.34
$B4_3$	92.64	91.64	97.64	89.65	88.23	92.89	92.68	81.99	99.95	93.11	88.33	99.96	83.70	81.46	94.20
$B4_4$	95.07	94.35	99.56	93.79	92.35	97.27	92.80	82.00	100	93.39	**88.79**	99.99	86.19	83.91	97.12
$B6_2$	91.44	90.87	96.09	87.70	86.64	91.28	92.66	81.88	99.69	93.37	88.54	99.72	81.92	79.87	92.58
$B6_3$	94.90	94.15	99.44	93.58	92.16	97.38	92.61	**82.13**	99.96	**93.44**	88.61	99.97	85.72	83.54	96.98
$B6_4$	**95.23**	**94.52**	99.95	95.32	93.87	99.27	92.74	81.89	100	93.30	88.45	99.99	87.06	84.77	98.61
DM	95.07	94.31	100	95.69	94.12	100	92.95	82.08	100	93.65	88.73	100	87.80	85.49	100

Table 6.1: SDP performance on the in-distribution set. First column denotes the decoder used: absolute (A), relative (R), bracketing (B), $4k$-bit (B4) and $6k$-bit (B6) encoding. The subscript in B, B4 and B6 denote the value of the hyperparameter k. The biaffine (DM) performance is displayed in the last row. Best SL-based parser is highlighted in bold and the best overall parser (baseline included) is underlined. The coverage of each encoding is displayed in the **CF** column. Languages are specified with the ISO-639 code.

	DM_{en}			PAS_{en}			PSD_{en}			PSD_{cs}		
	UF	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**
A	87.47	86.35	100	86.22	84.62	100	88.55	78.24	100	86.92	71.78	100
R	89.94	88.90	100	90.14	88.51	100	88.47	78.26	100	86.43	71.43	100
B_2	92.56	91.54	99.95	**94.82**	**93.18**	99.98	91.45	81.38	99.73	89.69	74.89	99.77
B_3	91.92	90.77	100.00	94.77	93.12	100	91.60	81.48	99.98	89.91	75.15	99.99
$B4_2$	85.54	84.71	90.45	81.37	80.18	82.46	91.44	80.97	99.58	90.11	75.22	99.53
$B4_3$	89.97	88.56	97.64	89.71	88.02	92.89	91.90	81.41	99.95	89.97	75.25	99.96
$B4_4$	92.41	91.28	99.56	93.03	91.40	97.27	91.73	81.34	100.00	90.13	75.19	99.99
$B6_2$	89.07	88.22	96.09	88.31	87.01	91.28	91.78	81.35	99.69	**90.44**	**75.72**	99.72
$B6_3$	92.05	91.05	99.44	93.27	91.70	97.38	91.88	81.53	99.96	90.40	75.61	99.97
$B6_4$	**92.71**	**91.67**	99.95	94.34	92.64	99.27	**91.95**	**81.61**	100.00	90.34	75.56	99.99
DM	92.52	91.43	100	94.13	92.39	100	91.81	81.27	100	90.63	75.44	100

Table 6.2: SDP performance on the out-of-distribution set. Same notation as in Table 6.1.

one point below biaffine. In general terms, the bracketing and bit encodings outperform the positional encodings, likely due to their lower number of unique generated labels ($|\mathcal{L}|$), which makes them easier to learn for a neural model. Positional encodings heavily rely on the global positional context and produce a higher number of unique labels, which is more challenging for the network to learn, and can lead to a worse performance such as the one displayed for the IWPT English treebank.

6.2 Speed analysis

In order to study the impact of SL approaches in terms of efficiency, we computed the Pareto front of the performance (UF score) against inference speed (in tokens per second) of each configuration. Figures 6.1 and 6.2 show the comparison in a selected subset of treebanks. Figure 6.1a shows the Pareto front for the DM_{en} in-distribution test set. The subset of optimal solutions is highlighted in bold: **$B6_4$** ■ and **B_2** □ with XLNet, **Biaf** ◆ and **B_3** ◇ with XLM and **B_3** ● and **B_2** ○ with BiLSTMs. These

	ar			bg			cs			nl			en			et		
	UF	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**
A	75.00	69.17	100	89.60	85.75	100	92.01	88.98	100	90.15	86.57	100	35.52	33.00	100	66.63	61.06	100
R	82.06	75.58	100	89.86	86.60	100	92.68	89.89	100	89.33	85.92	100	89.23	86.31	100	85.86	81.93	100
B_2	87.85	80.98	99.82	95.07	92.23	99.98	93.55	90.78	99.76	93.05	90.14	99.85	91.16	88.19	99.94	88.89	85.25	99.99
B_3	87.82	81.22	99.94	94.91	91.96	100	93.63	90.90	99.98	92.94	90.05	100	90.86	87.88	100	89.06	85.43	100
$B4_2$	87.84	81.21	99.77	95.16	92.16	99.86	93.68	90.93	99.69	93.50	90.69	99.89	91.39	88.45	99.80	88.85	85.18	99.98
$B4_3$	87.81	81.11	99.90	95.38	92.32	99.96	93.91	91.11	99.94	93.37	90.39	99.98	91.38	88.18	99.97	89.24	85.47	100
$B4_4$	88.09	81.27	99.94	95.46	92.51	99.98	93.86	91.08	99.98	93.50	90.50	99.99	91.30	88.37	99.99	88.91	85.24	100
$B6_2$	87.68	80.97	99.85	**95.56**	**92.65**	99.89	**94.01**	**91.35**	99.82	93.63	**90.81**	99.92	91.46	**88.53**	99.87	**89.31**	**85.70**	99.99
$B6_3$	88.10	81.37	99.93	95.42	92.60	99.97	93.94	91.21	99.96	93.53	90.55	99.99	91.38	88.43	99.97	89.17	85.46	100
$B6_4$	**88.33**	**81.60**	99.95	95.25	92.48	99.98	93.85	91.11	99.98	**93.82**	90.81	99.99	**91.50**	88.48	99.99	89.14	85.51	100
DM	89.72	82.69	100	95.56	92.47	100	94.76	91.99	100	93.49	90.18	100	92.44	89.29	100	90.87	87.69	100

	fi			fr			it			lv			lt			pl		
	UF	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**
A	84.52	80.97	100	80.79	76.58	100	88.68	86.12	100	81.72	77.42	100	64.19	58.55	100	91.75	87.13	100
R	85.90	82.34	100	83.06	78.66	100	89.25	87.16	100	84.64	80.63	100	72.85	66.42	100	90.96	86.47	100
B_2	91.19	88.16	99.60	91.06	87.60	99.97	94.26	92.21	99.96	89.94	86.48	99.49	83.61	77.49	99.41	93.19	88.80	99.70
B_3	91.20	88.13	99.94	90.59	86.65	100	94.42	92.47	100	89.85	86.42	99.95	83.85	78.03	99.87	93.50	89.09	99.98
$B4_2$	91.70	88.60	99.60	92.37	88.64	99.87	94.70	92.69	99.83	90.55	87.22	99.44	84.96	78.82	99.46	93.87	89.42	99.70
$B4_3$	91.58	88.41	99.87	92.60	88.88	99.98	94.64	92.70	99.98	90.43	87.11	99.82	85.03	78.74	99.83	93.95	89.52	99.93
$B4_4$	91.64	88.56	99.94	92.61	88.47	100	94.61	92.58	100	90.79	87.40	99.93	84.45	78.36	99.92	**94.07**	**89.57**	99.97
$B6_2$	92.02	89.03	99.69	92.60	88.82	99.93	**94.70**	**92.72**	99.94	90.27	87.06	99.57	85.09	**79.50**	99.59	93.84	89.39	99.78
$B6_3$	91.64	88.71	99.89	91.66	88.06	99.99	94.67	92.66	99.99	90.86	87.55	99.84	84.96	79.00	99.90	93.88	89.51	99.95
$B6_4$	**92.03**	**89.07**	99.95	**92.96**	**89.80**	100	94.29	92.31	100	**90.86**	**87.56**	99.94	**85.27**	79.49	99.96	93.95	89.54	99.98
DM	93.54	90.93	100	93.71	89.77	100	94.97	92.65	100	93.01	89.97	100	89.16	83.38	100	95.40	91.05	100

	ru			sk			sv			ta			uk		
	UF	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**	**UF**	**LF**	**CF**
A	93.28	91.45	100	83.53	79.39	100	77.50	73.48	100	34.22	27.27	100	70.49	67.21	100
R	93.08	91.33	100	86.79	82.59	100	81.79	78.05	100	64.69	53.69	100	81.92	78.74	100
B_2	94.73	93.09	99.93	93.31	90.15	99.79	88.57	84.90	99.92	73.62	62.02	100	90.32	87.65	99.77
B_3	94.75	93.13	100	93.61	90.41	99.96	88.75	85.10	100	73.62	62.02	100	90.35	87.75	99.99
$B4_2$	94.69	93.09	99.90	93.79	90.41	99.72	90.18	86.69	99.74	75.81	63.36	99.95	91.10	88.31	99.75
$B4_3$	94.79	93.19	99.99	94.08	90.51	99.94	**90.41**	86.62	99.97	76.01	**64.54**	100	91.28	88.54	99.95
$B4_4$	94.78	93.20	100	93.85	90.32	99.99	89.97	86.37	99.99	76.01	64.54	100	90.93	88.24	99.99
$B6_2$	94.94	93.39	99.94	94.06	90.86	99.85	90.32	**86.87**	99.86	**76.16**	63.48	100	91.38	**88.97**	99.86
$B6_3$	95.03	**93.52**	99.99	**94.26**	**90.94**	99.97	90.25	86.85	99.98	76.16	63.48	100	**91.41**	88.90	99.97
$B6_4$	**95.04**	93.51	100	93.89	90.73	99.99	90.03	86.53	100	76.16	63.48	100	91.16	88.66	99.99
DM	95.82	94.24	100	94.22	90.61	100	91.28	87.46	100	76.08	65.74	100	93.38	90.68	100

Table 6.3: IWPT performance on the test set.

configurations represent the best trade-off between performance and speed. There is a substantial difference between encoders: in all figures the BiLSTMs ○ are considerably faster than the XLM ◇ and XLNet □ encoders, but this comes at the cost of lower UF scores. XLNet is also consistently slower than XLM, probably due to the denser representations of the Transformer-XL architecture of XLNet against the vanilla Transformer of the XLM.

When comparing encodings in a single encoder configuration we see that positional encodings are the fastest approach, followed by the bracketing and $6k$-bit encoding and lastly the $4k$-bit encoding. Positional encodings are faster since their decoding implementation is easily parallelizable

– each label independently creates a subset of $\hat{A}$ –, while the other encodings rely on a transition-based system to decode the predicted arcs. In the case of the $4k$-bit encoding, the decoding process is the slowest one since it produces more connections that are passed through the FFN_r and requires an extra postprocessing step to remove the predicted artificial arcs. The bracketing and $6k$-bit approaches seem to be the best trade-off between performance and efficiency, specially in the IWPT treebanks, where biaffine suffers in terms of speed due to the larger number of arcs per graph (see Table 5.2, **#arcs**).

In the selected treebanks, biaffine is excluded from the set of optimal solutions in the DM_{en} and PAS_{en} in-distribution sets and the IWPT_{sk} and IWPT_{ta} test sets. The bracketing encoding with XLMs or XLNet is only excluded PAS_{zh} and IWPT_{sk}, while the $6k$-bit encoding is excluded in the PSD_{cs} and PAS_{zh}. The SL approaches of the Pareto front are always faster than biaffine and in the DM_{en}, PAS_{en}, PSD_{en}, IWPT_{sk} and IWPT_{ta} they reach a superior or paired performance to the baseline.

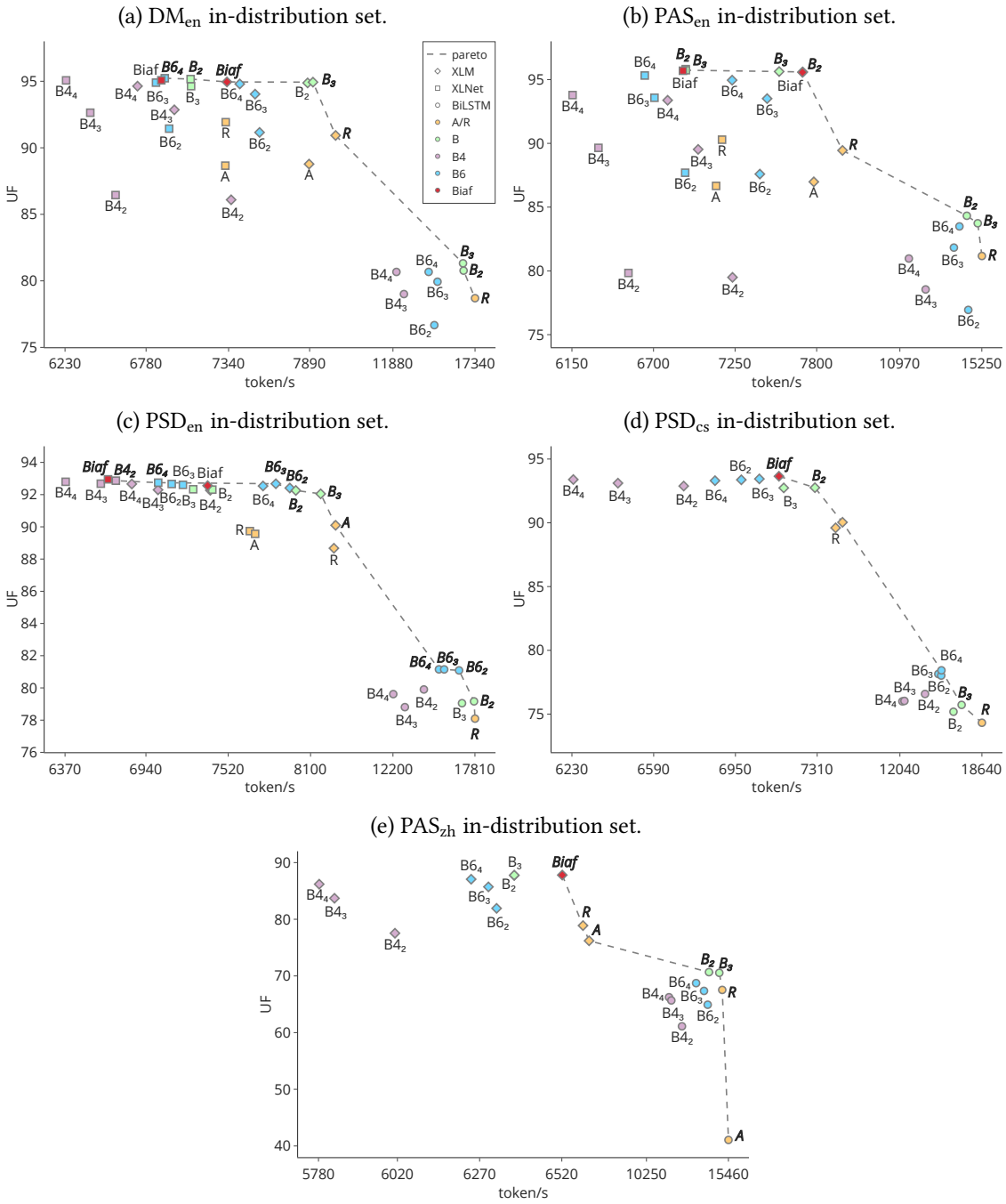


Figure 6.1: Pareto front of performance (UF) against inference speed (tokens per second) for the SDP in-distribution sets. For better visualization, the legend in Figure 6.1a is shared between the rest of them (Figure 6.1b-6.1e). The color indicates the encoding: absolute (A ○) and relative (R ○) in yellow, bracketing (B ○) in green, $4k$-bit (B4 ○) in purple, $6k$-bit (B6 ○) in blue and biaffine (Biaf ○) in red; and the shape indicates the encoder: BiLSTM ○, XLM ◇, XLNet □. The Pareto points are displayed with dashed lines and bold italics. Note that the horizontal axis is not linear: it has been scaled to better fit the BiLSTM points.

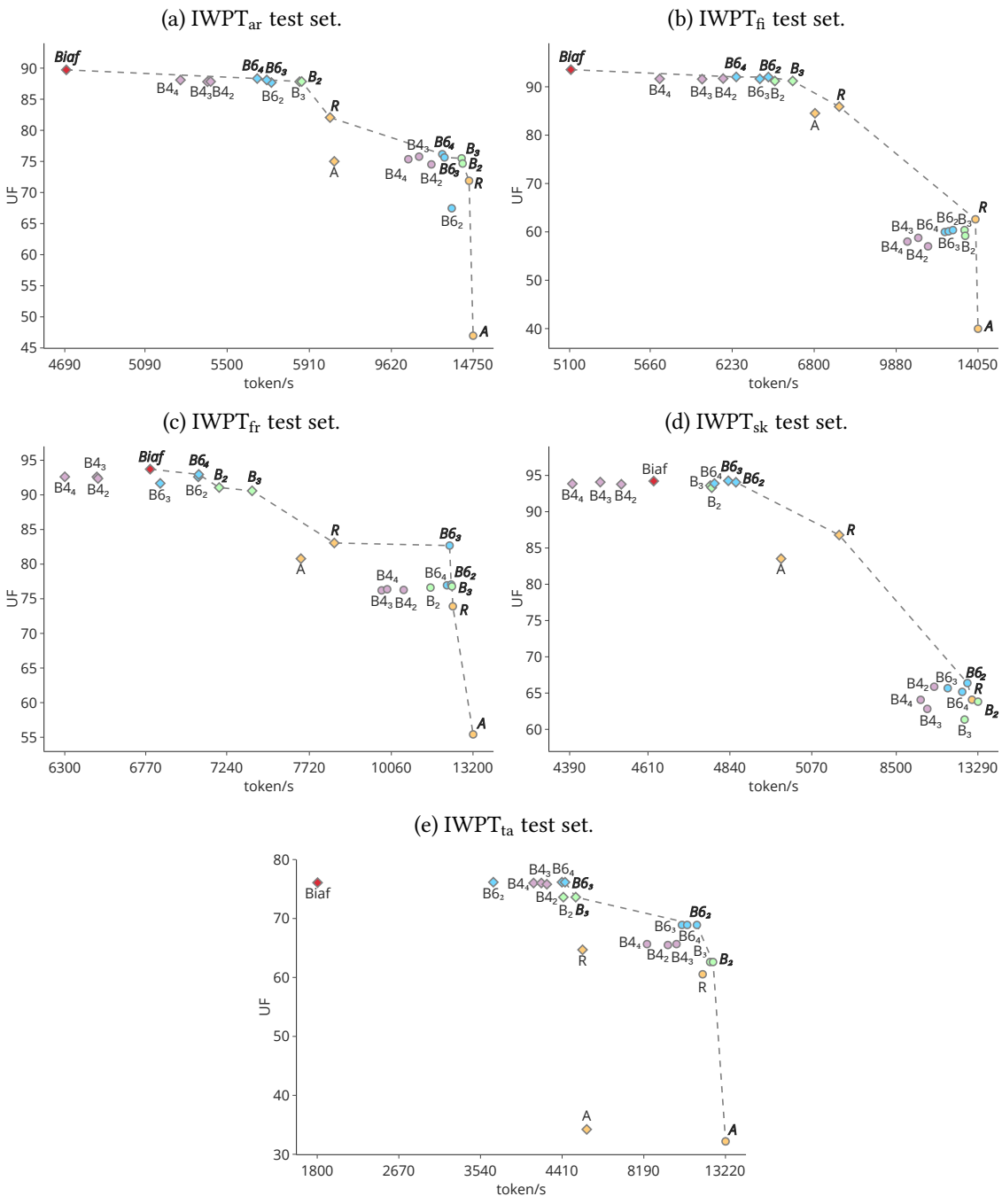


Figure 6.2: Pareto front of performance (UF) against inference speed (tokens per second) for the IWPT test sets. Same notation and legend as in Figure 6.1.

Chapter 7
Conclusion

This work introduces a novel approach to graph parsing by formulating it as a sequence-labeling task. Our approach aims to provide a simpler and more efficient alternative to traditional methods [17, 20, 71] while maintaining competitive accuracy. To this end, we proposed two families of SL-based graph linearizations: unbounded and bounded encodings.

Unbounded linearizations (Chapter 3) offer a straightforward representation of graph structures but do not constrain the label set, which may grow dynamically with the length or density of the graph. We introduced two variants within this category: the positional encoding, which comes in absolute and relative variants, and the bracketing encoding, which sequentially encodes graph structures based on their hierarchical dependencies. While these methods preserve expressiveness, their lack of a fixed label set poses challenges in scalability and implementation. To address these limitations, we designed bounded linearizations (Chapter 4) that restrict the number of possible labels. Specifically, we introduced the $4k$-bit and $6k$-bit encodings, where the hyperparameter k controls the coverage of the representation. For each of these encodings, we formally defined the encoding and decoding functions and provided detailed illustrative examples, demonstrating their theoretical soundness and applicability to graph parsing tasks.

To evaluate the effectiveness of our approach, we conducted experiments on two well-known datasets, the SemEval 2015 Task 18 [16] and IWPT 2021 Shared Task [14] datasets, using a biaffine parser [17] as baseline. We experimented with various neural encoder architectures, including BiLSTMs [51], XLM [72] and XLNet [54], to assess the impact of different contextualization strategies. Our empirical results indicate that SL approaches achieve paired performance with the biaffine parser in the SDP datasets, demonstrating their capability in handling structured linguistic graphs. In the IWPT datasets, biaffine exhibited a slight advantage in performance, likely due to its explicit modeling of pairwise relations. However, our efficiency analysis revealed a key advantage of SL-based methods: they consistently outperformed the biaffine parser in inference speed, making them highly suitable for real-time or large-scale parsing applications.

Overall, our work demonstrates that sequence-labeling formulations provide an alternative to traditional graph-based methods in NLP. Although SL methods have been explored for other structured prediction tasks, their adaptation to graph-based parsing is unprecedented. This highlights the potential of SL-based methods to redefine graph processing by offering a more computationally efficient alternative to traditional approaches.

Appendices

Glossary

Bounded A or encoding is said to be *bounded* when the set of labels $\mathcal{L}$ is fixed and it does not depend of the input structure. 2, 3, 5, 19, 20, 30

Coverage Measure of the amount of arcs that are recovered from the original set A after the encoding and decoding transformation. It assesses the similarity between A and $\delta(\varepsilon(A)$.. 20, 27, 51, 52

Decoding (δ) Surjective function that maps a sequence of n labels into the original set of arcs A. 3, 17, 19–21, 26, 27, 30, 31, 51, 59, 62

Deductive system Formal system consisting of a set of axioms and inference rules that define how new statements (theorems) can be derived from given premises. In this work we denote the inference rules as $\frac{\gamma_1 \cdots \gamma_{t-1}}{\gamma_{t+1}}(\gamma_t)$, where $\gamma_1, ..., \gamma_t$ are the set of premises and γ_{t+1} is the conclusion.. 23

Embedding A dense vector representation of a discrete input (such as words) designed to facilitate its use in neural architectures. Embeddings can be static [44] (precomputed from a fixed dictionary), contextual [31] (dynamically generated by a sequence model based on surrounding context), or learnable (optimized during training alongside the target neural model). 7–10, 14

Encoding (ε) Injective function that maps an input set of arcs A of a graph of size n-sized graph into a sequence of n labels. 3, 17–21, 26, 27, 30, 31, 33, 51, 59, 60, 62

Graph Abstract structure conformed by a set of nodes and a set of arcs. In this work, we assume that the term *graph* refers to a *labeled directed graph*, denoted as $G = (W, A)$, conformed by a set of ordered nodes, denoted as $W = (w_1, ..., w_n) \in \mathcal{V}$, and a set of arcs A. Each arc $(h \xrightarrow{r} d)$ is defined by its head ($h \in [0, n]$), dependent ($d \in [1, n]$) and label ($r \in \mathcal{R}$). The set of arcs A fulfills that it does not contain cycles of length one and each arc is unique in terms of its head and dependent position, so $(h \xrightarrow{r} d) \in A$ holds that $h \neq d$ and $\nexists (h \xrightarrow{r'} d) \in A$ such that $r \neq r'$. 17, 18, 20, 26, 27, 30, 32, 33, 51, 59

Graph linearization Specific method that compresses the arc information of a n-sized graph as a sequence of n labels through an encoding function, and performs a inverse operation to recover the arcs from the sequence of labels through a decoding function.. 2–5, 7, 18, 19, 51

Graph Parsing NLP task that aims to extract the paired relationships between the nodes of an input sentence $W = (w_1, ..., w_n)$ as a set of arcs A. 1–4, 7, 13, 17–19, 30

Pareto Front In multi-objective optimization problems, stands for a set of solutions that are non-dominated to each other but are superior to the rest of solutions in the search space [87]. 52, 55, 56

Relaxed-plane Set of arcs with no crossing arcs in the same direction. We say that a set of arcs A is distributed in relaxed planes if its arcs are distributed in mutually exclusive subsets such that each subset (a relaxed plane) does not contain crossing arcs in the same direction.. 27, 30, 48, 51

Sequence Labeling NLP paradigm where complex structures built upon an input sequence, denoted as $(w_1, ..., w_n) \in \mathcal{V}^n$, are represented as a sequence of labels $(\ell_1, ..., \ell_n) \in \mathcal{L}^n$ that matches the size of the input. 2

Unbounded A or encoding is said to be *unbounded* when the set of labels $\mathcal{L}$ is not fixed and might grow indefinitely depending on the nature of the input structure. 2–4, 19, 20

Acronyms

AI Artificial Intelligence. 1, 2, 4

BiLSTM Bidirectional LSTM. 10, 14, 48, 53, 55

DL Deep Learning. 3, 5, 7

FFN Feed Forward Network. 7–10, 14, 17

LLM Large Language Model. 9–11, 13, 14

LSTM Long-Short Term Memory. 9, 10, 61

ML Machine Learning. 8

MLM Masked Language Modeling. 11, 12

NLP Natural Language Processing. 1, 2, 4, 7–13, 17, 57, 60

NLU Natural Language Understanding. 7, 10–12

NSP Next Sentence Prediction. 12

PLM Permutation Language Modeling. 12

RNN Recurrent Neural Network. 9

SL Sequence Labeling. 2–5, 7, 17–19, 46, 48, 51, 52, 54, 57

SoTA State-of-the-art. 2, 3, 5, 9, 10, 12, 13, 15, 46, 48

Symbols

$\delta : \mathcal{L}^n \to \mathcal{A}^n$ Decoding function. 19, 20, 51, 59

λ Empty string. 27

$\mathcal{A}^n$ Set of all possible sets of arcs A for n-sized . 19

$\mathcal{L}$ Set of possible labels of an encoding. 19, 20, 30, 52, 59, 60

$\mathcal{R}$ Set of possible arc labels. 13, 59

$\mathcal{V}$ Vocabulary (set of possible words). 13, 19, 59, 60

$\varepsilon : \mathcal{A}^n \to \mathcal{L}^n$ Encoding function. 19, 20, 51, 59

sort Ordering function. When applied to numbers, it arranges a sequence of numbers in ascending order. When applied to arcs, it sorts a sequence of arcs in ascending order by its leftmost component ($\min\{h, d\}$) and rightmost component (($\max\{h, d\}$) in case two arcs share the same leftmost component. For example $\text{sort}(1 \to 2, 5 \to 6, 2 \to 3, 7 \to 1) = (1 \to 2, 7 \to 1, 2 \to 3, 5 \to 6)$. 20

Bibliography

[1] X. Wang, R. Wang, C. Shi, G. Song, and Q. Li, "Multi-Component Graph Convolutional Collaborative Filtering," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[2] J. Wu, X. Wang, F. Feng, X. He, L. Chen, J. Lian, and X. Xie, "Self-supervised Graph Learning for Recommendation," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, 2021.

[3] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional Networks on Graphs for Learning Molecular Fingerprints," in *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2015.

[4] C. Hetang, H. Xue, C. Le, T. Yue, W. Wang, and Y. He, "Segment Anything Model for Road Network Graph Extraction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2024.

[5] D. Jurafsky and J. H. Martin, *Speech and language processing*, 2nd ed. Prentice Hall, Pearson Education International, 2009.

[6] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, 2009.

[7] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[8] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," in *International Conference on Learning Representations*, 2018.

[9] Y. Zhang, X. Yu, Z. Cui, S. Wu, Z. Wen, and L. Wang, "Every Document Owns Its Structure: Inductive Text Classification via Graph Neural Networks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[10] J. Bastings, I. Titov, W. Aziz, D. Marcheggiani, and K. Sima'an, "Graph Convolutional Encoders for Syntax-aware Neural Machine Translation," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2017.

[11] M. Yasunaga, H. Ren, A. Bosselut, P. Liang, and J. Leskovec, "QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 2021.

[12] N. Xue, H. T. Ng, S. Pradhan, R. Prasad, C. Bryant, and A. Rutherford, "The CoNLL-2015 Shared Task on Shallow Discourse Parsing," in *Proceedings of the Nineteenth Conference on Computational Natural Language Learning - Shared Task*. Association for Computational Linguistics, 2015.

[13] R. Xia and Z. Ding, "Emotion-Cause Pair Extraction: A New Task to Emotion Analysis in Texts," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019.

[14] S. Oepen, K. Sagae, R. Tsarfaty, G. Bouma, D. Seddah, and D. Zeman, Eds., *Proceedings of the 17th International Conference on Parsing Technologies and the IWPT 2021 Shared Task on Parsing into Enhanced Universal Dependencies (IWPT 2021)*. Association for Computational Linguistics, 2021.

[15] J. Barnes, L. Oberlaender, E. Troiano, A. Kutuzov, J. Buchmann, R. Agerri, L. Ø vrelid, and E. Velldal, "SemEval 2022 Task 10: Structured Sentiment Analysis," in *Proceedings of the 16th International Workshop on Semantic Evaluation (SemEval-2022)*. Association for Computational Linguistics, 2022.

[16] S. Oepen, M. Kuhlmann, Y. Miyao, D. Zeman, S. Cinková, D. Flickinger, J. Hajič, and Z. Urešová, "SemEval 2015 Task 18: Broad-Coverage Semantic Dependency Parsing," in *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*. Association for Computational Linguistics, 2015.

[17] T. Dozat and C. D. Manning, "Simpler but More Accurate Semantic Dependency Parsing," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, I. Gurevych and Y. Miyao, Eds. Association for Computational Linguistics, 2018.

[18] X. Wang, J. Huang, and K. Tu, "Second-Order Semantic Dependency Parsing with End-to-End Neural Networks," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019.

[19] D. Gildea, G. Satta, and X. Peng, "Cache Transition Systems for Graph Parsing," *Computational Linguistics*, 2018.

[20] D. Fernández-González and C. Gómez-Rodrí guez, "Transition-based Semantic Dependency Parsing with Pointer Networks," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[21] N. Kitaev and D. Klein, "Tetra-Tagging: Word-Synchronous Parsing with Linear-Time Inference," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[22] C. Gómez-Rodrí guez and D. Vilares, "Constituent Parsing as Sequence Labeling," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2018.

[23] M. Strzyz, D. Vilares, and C. Gómez-Rodrí guez, "Viable Dependency Parsing as Sequence Labeling," 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)*. Association for Computational Linguistics, 2019.

[24] A. Amini, T. Liu, and R. Cotterell, "Hexatagging: Projective Dependency Parsing as Tagging," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 2023.

[25] C. Gómez-Rodrí guez, D. Roca, and D. Vilares, "4 and 7-bit Labeling for Projective and Non-Projective Dependency Trees," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023.

[26] A. Ezquerro, D. Vilares, and C. Gómez-Rodríguez, "Dependency Graph Parsing as Sequence Labeling," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024.

[27] A. Ivanova, S. Oepen, L. Øvrelid, and D. Flickinger, "Who Did What to Whom? A Contrastive Study of Syntacto-Semantic Dependencies," in *Proceedings of the Sixth Linguistic Annotation Workshop*. Association for Computational Linguistics, 2012.

[28] Y. Miyao, T. Ninomiya, and J. Tsujii, "Corpus-Oriented Grammar Development for Acquiring a Head-Driven Phrase Structure Grammar from the Penn Treebank," in *Natural Language Processing – IJCNLP 2004*. Springer Berlin Heidelberg, 2005.

[29] J. Hajič, E. Hajičová, J. Panevová, P. Sgall, O. Bojar, S. Cinková, E. Fučíková, M. Mikulová, P. Pajas, J. Popelka, J. Semecký, J. Šindlerová, J. Štěpánek, J. Toman, Z. Urešová, and Z. Žabokrtský, "Announcing Prague Czech-English Dependency Treebank 2.0," in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. European Language Resources Association (ELRA), 2012.

[30] G. Bouma, D. Seddah, and D. Zeman, "From Raw Text to Enhanced Universal Dependencies: The Parsing Shared Task at IWPT 2021," in *Proceedings of the 17th International Conference on Parsing Technologies and the IWPT 2021 Shared Task on Parsing into Enhanced Universal Dependencies (IWPT 2021)*. Association for Computational Linguistics, 2021.

[31] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "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). Association for Computational Linguistics, 2019.

[32] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds. Curran Associates, Inc., 2020.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[34] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015.

[35] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.

[37] F. Rosenblatt, "The Perceptron: A perceiving and recognizing automaton," Project PARA, Cornell Aeronautical Laboratory, Tech. Rep., 1957.

[38] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, 1991.

[39] S. ichi Amari, "Backpropagation and stochastic gradient descent method," *Neurocomputing*, 1993.

[40] A. Vinokourov, N. Cristianini, and J. Shawe-Taylor, "Inferring a Semantic Representation of Text via Cross-Language Correlation Analysis," in *Advances in Neural Information Processing Systems*, vol. 15. MIT Press, 2002.

[41] P. Rodríguez, M. A. Bautista, J. Gonzàlez, and S. Escalera, "Beyond one-hot encoding: Lower dimensional target embedding," *Image and Vision Computing*, 2018.

[42] F. Morin and Y. Bengio, "Hierarchical Probabilistic Neural Network Language Model," in *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, vol. R5. PMLR, 2005, pp. 246–252.

[43] A. Mnih and G. E. Hinton, "A Scalable Hierarchical Distributed Language Model," in *Advances in Neural Information Processing Systems*, vol. 21. Curran Associates, Inc., 2008.

[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," 2013.

[45] J. Pennington, R. Socher, and C. Manning, "GloVe: Global Vectors for Word Representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014.

[46] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," 2020.

[47] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and A. Stent, "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)*. Association for Computational Linguistics, 2018, pp. 2227–2237.

[48] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*, 1987.

[49] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 28, no. 3. PMLR, 2013.

[50] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, 1997.

[51] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, 1997.

[52] T. Dozat and C. D. Manning, "Deep Biaffine Attention for Neural Dependency Parsing," in *International Conference on Learning Representations*, 2017.

[53] B. Bohnet, R. McDonald, G. Simões, D. Andor, E. Pitler, and J. Maynez, "Morphosyntactic Tagging with a Meta-BiLSTM Model over Context Sensitive Token Encodings," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018.

[54] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," in *Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[55] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. (2018) Improving language understanding by generative pre-training.

[56] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. (2019) Language Models are Unsupervised Multitask Learners.

[57] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "LLaMA: Open and Efficient Foundation Language Models," 2023.

[58] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open Foundation and Fine-Tuned Chat Models," 2023.

[59] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou, T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco,

A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk, S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, "The Llama 3 Herd of Models," 2024.

[60] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7B," 2023.

[61] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A.

Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mixtral of Experts," 2024.

[62] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[63] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, 2020.

[64] A. Ezquerro, C. Gómez-Rodríguez, and D. Vilares, "On the Challenges of Fully Incremental Neural Dependency Parsing," in *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 2023.

[65] ——, "From Partial to Strictly Incremental Constituent Parsing," in *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 2024.

[66] W. Foundation. Wikimedia Downloads.

[67] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning Books and Movies: Towards Story-Like Visual Explanations by Watching Movies and Reading Books," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[68] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Association for Computational Linguistics, 2018.

[69] J. Zhou and H. Zhao, "Head-Driven Phrase Structure Grammar Parsing on Penn Treebank," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019.

[70] Y. Zhang, H. Zhou, and Z. Li, "Fast and Accurate Neural CRF Constituency Parsing," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. International Joint Conferences on Artificial Intelligence Organization, 2020.

[71] X. Wang and K. Tu, "Second-Order Neural Dependency Parsing with Message Passing and End-to-End Training," in *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, 2020.

[72] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, "Unsupervised Cross-lingual Representation Learning at Scale," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020.

[73] G. Wenzek, M.-A. Lachaux, A. Conneau, V. Chaudhary, F. Guzmán, A. Joulin, and E. Grave, "CCNet: Extracting High Quality Monolingual Datasets from Web Crawl Data," in *Proceedings of the Twelfth Language Resources and Evaluation Conference*. European Language Resources Association, 2020.

[74] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, "Transformer-XL: Attentive Language Models beyond a Fixed-Length Context," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2019.

[75] J. Nivre, J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret, "The CoNLL 2007 Shared Task on Dependency Parsing," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, 2007.

[76] M. Strzyz, D. Vilares, and C. Gómez-Rodríguez, "Bracketing Encodings for 2-Planar Dependency Parsing," in *Proceedings of the 28th International Conference on Computational Linguistics*. International Committee on Computational Linguistics, 2020.

[77] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu, "Automated Concatenation of Embeddings for Structured Prediction," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 2021.

[78] M. Covington, "A Fundamental Algorithm for Dependency Parsing," in *Proceedings of the 39th Annual ACM Southeast Cnference*. Association for Computing Machinery, 2001.

[79] J. Nivre, D. Zeman, F. Ginter, and F. Tyers, "Universal Dependencies," in *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Tutorial Abstracts*. Association for Computational Linguistics, 2017.

[80] H. Shavarani and A. Sarkar, "SpEL: Structured Prediction for Entity Linking," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2023.

[81] A. Ramponi, R. van der Goot, R. Lombardo, and B. Plank, "Biomedical Event Extraction as Sequence Labeling," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

[82] S. Zhou and T. Qian, "On the Strength of Sequence Labeling and Generative Models for Aspect Sentiment Triplet Extraction," in *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, 2023.

[83] Z. Huang, P. Cao, J. Zhao, and K. Liu, "DiffusionSL: Sequence Labeling via Tag Diffusion Process," in *Findings of the Association for Computational Linguistics: EMNLP 2023*. Association for Computational Linguistics, 2023.

[84] Y. Tong, G. Chen, G. Zheng, R. Li, and J. Dazhi, "When Generative Adversarial Networks Meet Sequence Labeling Challenges," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2024.

[85] A. L. Maas, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the International Conference on Machine Learning*, 2013.

[86] I. Loshchilov and F. Hutter, "hrefhttps://openreview.net/forum?id=Bkg6RiCqY7Decoupled Weight Decay Regularization," in *International Conference on Learning Representations*, 2019.

[87] E. Zitzler, J. Knowles, and L. Thiele, *Quality Assessment of Pareto Set Approximations*. Springer Berlin Heidelberg, 2008.