1 Problem Setting

Dependency parsing is a structure prediction problem in which we take a sentence as the input, and we try to predict a tree to represent the structure of the sentence. Every edge of the tree has a label. For every edge from a word a-> b, b modifies a in some way, which is captured by the label. In the tree, every word has exactly 1 parent. S CoNLL is the most common format to represent the dependency tree. We assignment each word has an index according to its position in the sentence. Since each node has exactly 1 parent, we write the the index of the parent node in the representation along with the label.

Non-Projectivity

One of the key features of a dependency parse tree is whether the tree has crossing arcs or not. The types of algorithms we can use for the problem depends on whether the sentence/language we have has crossing arcs or not. Crossing arcs are fairly uncommon in English, but can be extremely common in other languages, such as Czech. A tree with no crossing arcs is said to a projective

An example of a sentence with crossing arcs is written below. In it, saw is modified by dog, and by yesterday, but dog is in term modified by Terrier(through was). Hence the arc between saw and yesterday is 'crossed’ by the arc between dog and was.

Formal Conditions

The following are the constraints applicable to the structure of a potential dependency parse graph:

For a Graph $G = (V, A)$, with a label set $L = \{l_1, \ldots, l_{|L|}\}$,

- $G$ is (weakly) connected: If $i, j \in V, i \leftrightarrow^* j$
- $G$ is acyclic: If $i \rightarrow j$, then not $j \rightarrow^* i$
2 Styles of dependency parsing

There are 2 main styles of dependency parsing under which most of the methods can be classified.

Transition based

These methods are usually performed using a stack and some simply operations, so they usually simply go left to right through a sentence. These methods are fast and are usually done greedily. They normally don’t give any optimality constraints. This style includes greedy methods and beam search methods. In beam search, you explore multiple hypothesis, not just one. As your beam becomes wider, accuracy tends to go up, however, execution time increases linearly with beam size.

Graph based

These methods are usually much slower, but are guaranteed to find what is the best tree. We can use higher order factorizations in order to increase the model accuracy, however, the higher order models are much slower. Most commonly used graph based algorithms take cubic time.

Both transition based and graph based models now have neural network versions.

3 Graph Based Models

3.1 Arc-Factored Models

These are the simplest of the graph based models. We assume that the score for the tree is simply the summation of the individual scores of all the arcs. Or, \( w(G) = \prod_{(i,j,k) \in G} w^k_{ij} \), where \( w^k_{ij} \) is the weight of creating a dependency from word \( w_i \) to \( w_j \), with a label \( l_k \). This is similar to what is used in POS-tagging, or in sequence modeling. This model does have a strong assumption that each dependency decision is independent, which may or may not be true. The scores are assumed to be independent with each other.

For example, if we have a sentence ”I washed the dishes with detergent”. The arc-factored score for this sentence will be as follows:

\[
\text{Score(sentence)} = \text{Sc}(\text{I} \leftarrow \text{washed}) + \text{Sc}(\text{washed} \to \text{dishes}) + \text{Sc}(\text{washed} \Rightarrow \text{with}) + \text{Sc}(\text{with} \to \text{detergent})
\]

Looking at this sentence, we may feel like washed \( \to \text{with} \) is a good arc. However, if the sentence was ”I washed the dishes with a fancy pink pattern, then dishes \( \to \text{with} \) is the correct arc. With arc-factored models, we have no way to be able to differentiate between these cases. We will look at how to deal with this assumption later on.

Note that there are exponentially many trees possible for any given sentence. In order to design an efficient algorithm, we have to figure out how to solve 2 parts of the problem. (a) Given a tree, how do you score it? (b) How do you find the best scoring tree in the exponential space?
For the second part, we have many options. We can search over just the projective trees, over all trees, or something in between.

### 3.2 Dynamic Programming

If we search over just the projective trees, we can use dynamic programming. Every word and all its dependents form one interval in the input sentence.

**Theorem:** All projective graphs can be written as the combination of two smaller adjacent graphs.

For some particular head word, all of its dependents exist in a span from $i$ to $j$. We can split up the sentence into different spans. We can build up these spans in whichever order we like. For example, we can build up the graph for all words but the last one, and then simply add the last one.

**The naive approach**

The simplest way to approach this problem is to simply do it in a bottom-up manner. Consider a case where we have a span headed by $h$ from $i$ to $l$ and a second span from $l + 1$ to $j$ headed by $h'$, so these spans are adjacent. Then we can make an arc from $h$ to $h'$, and create a bigger span from $i$ to $j$, which is headed by $h$.

Spans are represented using triangles, with the top vertex indicating the head, the left vertex corresponding to the start, or $i$, and the right vertex the end, or $j$.

In order to get the best tree, we have to search over all the possible split point $l$, and search over all the possible heads $h'$. This leads to $O(n^5)$ complexity ($n^3 \cdot n^2$), but will always give the optimal output.

Consider the sentence "The dog ate smelly cheese.". Assume we have already build up a subtree from "The" to "dog" rooted at "dog" and a second subtree from "ate" to "cheese" rooted at "ate". We will try to create a larger tree rooted at "ate", which the split-point between "dog" and "ate", for which we will merge the 2 subtrees, add an arc from "ate" to "dog". In practice, we would consider all possibilities. For example, we would also consider splitting between "the" and "dog". However, all other possibilities would lead to a lower scoring tree.

**Eisner Algorithm**

In order to make it faster, we utilize the definition of projectivity.

Consider the sentence "The dog with spots really likes trees". If the arc from "likes" to "dog" is in the tree, we know that nothing can cross that arc. So everything between "likes" to "dog" has to be entirely self contained.

Imagine a case in which we have two adjacent spans. We can split both spans down the middle, according to the root position in the span. Now we have 4 triangles instead of 2. Since we cannot have crossing arcs, we know the left part of $h$, and the right part of $h'$ have to be completely independent. So we only need to consider the merger of the right side of $h$ and the left side of $h'$. This will lead to a significant decrease in the number of indices over which we have to search. Once we join the inner two triangles into a trapezoid, we can add the outer two triangles we had earlier to form the complete span from $i$ to $j$.

This algorithm offers significant benefits over the previous model. It decreases the space complexity from $O(n^3)$ to $O(n^2)$, and the time complexity from $O(n^5)$ to $O(n^3)$, since we only have to pick one split point. The number of stages involved has increased from 1 to 3: Split the spans, merge the inner two triangles, combine into one larger span.
4 Maximum Spanning Tree

4.1 Non Projective Maximum Spanning Tree

Given an arc-factored model, finding the highest scoring dependency tree $y$ for a sentence $x$ is a graph-theoretic problem of finding a weighted maximum spanning tree in $G_x$. The Eisner algorithm only deals with Projective Maximum Spanning Tree Problem, without the Projectivity constraint. For solving maximum spanning tree with projectivity constraint we require a new algorithm.

4.2 Chu-Liu-Edmunds Algorithm

Its a greedy recursive algorithm where we start building a graph for every node from it’s highest incoming arc, except the root node. If the graph formed is acyclic then it is the Maximum Spanning Tree, otherwise the algorithm identifies the cycle and contracts the cycle into a single node. As the weight of the MST of the contracted graph is equal to the weight of the original graph, MST calls itself on the smaller graph recursively for recalculating arc-weights into and out of cycle.

The objective of the algorithm is to build a MST Graph $G$:

$$G = \arg\max_{G \in \mathcal{T}(G_x)} \sum_{(i,j,k) \in G} w^k_{ij}$$

where arc-weights are the linear combinations of features of the arc and their corresponding weight vector, arc-weight $w_{ij} = \arg\max_{k} w^k_{ij}$ from node $i$ to $j$ for all $k$ intermediate nodes as root.

The naive implementation of the algorithm is as follows:
1. Greedily pick an incoming edge for each node
2. If the final graph formed is acyclic, return MST
3. Else, the graph contains a cycle, remove the cycle by choosing an incoming edge to the cycle and accordingly removing the edge from the cycle, rendering it acyclic.

Every recursive call takes $O(n^2)$, time for $O(n)$ recursive calls, which makes its time complexity $O(n^3)$. In fact, using appropriate data structures, it is possible to get the time complexity down to $O(n^2)$.

Figure 2: An example of a Complex Graph reduction into MST using Chu-Liu Edmonds

![Diagram](image-url)
Arc-Weights and Feature ideas

Arc weights are a linear combination of the features of the arc, \( f \), and a corresponding weight vector \( w \). We take this to the exponent in order to simplify the math. The equation can be written as

\[
 w_{ij}^k = e^{w \cdot f(i,j,k)}
\]

Some ideas for possible features are:

- The words themselves, i.e. identities of the words \( w_i \) and \( w_j \) and the label \( l_k \)
- Part-of-speech tags of the words \( w_i \) and \( w_j \) and the label \( l_k \)
- Part-of-speech of the context, i.e. the words surrounding and between \( w_i \) and \( w_j \)
- Number of words between \( w_i \) and \( w_j \), and their orientation
- A combination of the above

Note that feature selection is important in the linear setting, not in the cases we have a neural network setting. Also, in the linear setting, these are often trained with a Structured Perceptron.

![Figure 3: An example of possible features for the arc from 'went' to 'As' for the sentence 'As McGwire neared, fans went wild.'](image)

5 Transition Based Dependency Parsing

5.1 Arc-Standard

Transition based dependency parsing processes the sentences from left to right. One of the transition based dependency parsing strategy is Arc-Standard. At every step, it maintains a stack which contains words that
are being processed and a buffer of remaining words. Initially the stack is empty and the buffer contains
entire list of words in the sentence. At each step, one of the following operations is applied to the stack:

1. Shift: This removes the first word from the buffer and pushes it on the top of the stack.
2. Left Arc: This applies a dependency arc from the word at top of the stack to the next word in stack.
   These two words are combined into a single element in the stack.
3. Right Arc: This applies a dependency arc to the word at top of the stack from the next word in stack.
   These two words are combined into a single element in the stack.
The transitions are applied till the buffer becomes empty.

Consider the following example sentence:
I booked a flight to Lisbon.
Initially the buffer contains all the words. The stack will be empty.
Buffer:
I booked a flight to Lisbon
Stack:

Since the stack is empty, the only operation that can be performed is shift. Hence I is pushed to the stack.
Buffer:
booked a flight to Lisbon
Stack:
I

Since the stack has just one element, the only operation that can be performed is shift.
Buffer:
a flight to Lisbon
Stack:
booked
I

At this step, there are 3 choices: shift, left arc, right arc.
Applying left arc operation:
Buffer:
a flight to Lisbon
Stack:
I ← booked

This process is repeated till the buffer becomes empty.
Since each step can have just three operations, this problem is reduced to a classification problem at each
step. The features to this classification problem can be:
1. Top of the stack word
2. Top of the stack POS
3. Buffer front word
4. Child of the top of stack

SVM or Structured Perceptron can be used for this classification. The hyperparameters include:
1. Regularization
2. Loss function
3. Hand-crafted features
5.2 Neural Network Transition Based Parser

In place of a linear model, a neural network can be used to solve this classification problem. The features (POS tags, words, arc-labels) are converted into atomic inputs using one hot representation. There is one embedding matrix for each feature type. The embedding matrix is used to convert the atomic inputs to vector representation. This forms the embedding layer which is connected to a fully connected hidden layer. Finally, the softmax classifier outputs which action should be performed.

The network can be further improved by changing the size of hidden layer or adding another hidden layer. This algorithm is a greedy algorithm. Weiss et al. suggested combining the representations of different layers to form features for a structured perceptron.

The hyperparameters for this neural network can be regularization, loss function, dimensions, activation function, initialization, adagrad, dropout, mini-batch size, learning rate schedule, momentum, stopping time, parameter averaging.

5.2.1 Effect of Embedding Dimensions

The results are better for larger word embedding dimensions.

![Figure 4: Effect of Embedding Dimensions](image)

5.2.2 How important is Lookahead

If the neural network is allowed to see what is coming in from the buffer (lookahead), the accuracy increases. Without lookahead the accuracy is quite bad. A single lookahead increases the accuracy by a huge amount and it increases by a small amount with increase in the lookahead which gives an accuracy close to the LSTM model.

5.2.3 Beam Search with Local Model

Beam Search maintains the best k hypothesis at each step. The accuracy gain obtained with beam search is not much high. The accuracy for beam search can be increased using early updates. During the training, if the actual path falls out of the beam, the parsing is stopped and the weights are updated only upto that point.
Thus for local model every decision cannot be penalized for pushing the result too far from the true result. But global model is able to penalize or assign credit to each decision. The global model outperforms the local and beam search model.

6 NN Graph-based Parsing

In graph based parsing, the score of tree is calculated as summation of scores of the edges. Neural Networks can be used to calculate the score of each edge. The words are passed through a Bidirectional LSTM, which is used to represent all the words. These representations are used to score the edge. This method provides quite high accuracies (UAS=95.74, LAS=94.08).

7 Universal Dependencies

Universal Dependencies give same kind of representation across 8 different languages. The unlabeled attachment scores for different languages are given in the figure below.