Universiteit Leiden

Master Computer Science

Automatic Detection of Differences and Similarities in Natural Language Treebanks

Name:	Andrew Caruana
Student ID:	s4014049
Date:	09/07/2025
Specialisation:	Data Science
1st supervisor:	Francesco Bariatti
2nd supervisor:	Matthijs van Leeuwen

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Natural languages are the primary means of communication between people around the world. However, these languages can differ greatly, or not much at all, mainly based on the geography, history and culture of the languages' speakers. Therefore we ask the question, can we automatically find differences and similarities between different languages? Our approach uses a graph miner to mine frequently occurring subtrees within natural language treebanks, and then utilise the Minimum Description Length (MDL) Principle to ensure the meaningful patterns remain and to mitigate the pattern explosion problem. Experiments were conducted on six languages: English, French, Italian, Czech, Icelandic and Arabic. The results demonstrate that the approach works, and various language syntax patterns can be found and associated with one or many languages. Analysis of the results shows that all the languages had a reasonable number of patterns in common, however, Arabic was the language with the most patterns not found in other languages. In addition, English had the most patterns shared with other languages, with only one unique pattern found. Whilst we did have computational limitations, and a lack of analysis from a linguistic point of view, this thesis illustrates the feasibility of combining tree mining with MDL-based model selection to differentiate between languages automatically. Future work may focus on optimising the efficiency of our algorithms, making use of more or different languages and collaborating with linguists for a more thorough analysis.

Contents

Chapter 1

Introduction

This chapter lays the foundation for the entire thesis. It introduces the problem that this work aims to tackle, and gives a concrete description as to how we aim to do so. Finally, we also provide a short overview of the content of the remaining chapters.

1.1 Problem Statement

Language is a means of communication present in the vast majority of people's lives worldwide. Plenty of languages, specifically, natural languages exist with differing levels of relation between these languages, based on the culture and history of the people that spoke them. This means that certain languages tend to be quite similar, and others tend to be quite different. For example, Italian and French are both Romance languages, and therefore one would expect them to share some similar words and grammar. Whereas, if one were to compare the same two languages to Mandarin Chinese, they would likely find much less similarity in comparison.

The issue posed, however, is that these similarities and dissimilarities have to be found manually, by means of linguists who have devoted lots of time and energy to their study. This begs the question: Can similarities and differences between languages be found automatically, and with little to no linguistic background?

There has been work done on language classification, which is the task of identifying a language given a piece of text [1, 2, 3]. However, no work has been done on differentiating languages between each other. Furthermore, when talking about similarities or differences between a language, this means that languages share or do not share certain features, i.e. certain syntactic patterns are present in some languages, but not in others. It might be interesting to mention the differences between descriptive and discriminative approaches. Classification is a discriminative approach, meaning it aims to distinguish data by assigning it to categories. Whereas, with descriptive approaches such as using patterns, we are able to find inherent traits of the data itself.

Pattern mining is already a prominent subfield of computer science; it involves the task of extracting patterns from all kinds of data. This can be data in the form of text [4, 5], graphs [6, 7] or even, and especially popular, transactions [8, 9]. Finding a suitable dataset, therefore, is a crucial first step as it heavily dictates how one would move forward. The Universal Dependencies (UD) [10, 11] dataset fits our requirements perfectly. It is a collection of treebanks for various languages. A treebank is a collection of trees, where each tree within the UD treebanks

represents one sentence in any language. There are also parallel corpora available for different languages for a fair comparison.

Unfortunately, one major issue when it comes to pattern mining is pattern explosion. When mining patterns, one often ends up with too many patterns to realistically use and evaluate. Therefore, one must find a way of reducing the pattern set. One way of doing so is with the Minimum Description Length (MDL) Principle [12, 13]. Simply put, it states that the best model to describe some piece of data is the one that compresses it the best. Using the MDL principle, one can construct a model that captures meaningful patterns from various types of data, whether it be itemsets [14, 15] or even graphs [16, 17, 18]. We aim to implement a similar method, however, using the trees provided by Universal Dependencies instead. In addition, it is worth mentioning that there has been work done on finding the differences and similarities between various databases using MDL and pattern mining [19], however, this has not been done for trees.

Once the MDL Principle has been successfully applied, and the best model has been found, some analysis is done on the model in order to determine if the patterns mined are indeed meaningful and what the relationship between the patterns found in the various languages is.

1.2 Research Objectives

Our aim is to automatically capture the differences and similarities between different languages. More specifically, our research question can be said to be the following. Can syntactic similarities and differences between natural languages be automatically identified by mining frequent subtrees within the Universal Dependencies treebanks using the MDL principle and without the need for extensive linguistic experience? This research question can thus be broken down into a few key steps:

1. **Data Selection:** Within UD, several treebanks are present. Several treebanks are selected that include both closely related languages (such as Italian and French) and languages that differ (such as English and Arabic).

2. **Candidate Pattern Extraction:** Utilise a pattern miner to extract patterns from the data. Where a pattern is a frequently occurring subtree within the original tree itself. These patterns represent frequently occurring syntactic relationships within languages.

3. **Model Construction using the MDL Principle:** Construct an MDL-based model to capture meaningful patterns and combat pattern explosion.

4. **Model Analysis:** Analyse the model and the patterns contained within. This is done by comparing the various language associations that each meaningful pattern has.

1.3 Thesis Structure

The structure of this thesis is as follows. Chapter 2 introduces some fundamental concepts for the comprehension of this thesis. Chapter 3 discusses some related work and highlights some papers that served as inspiration. Chapter 4 provides some details about the Universal Dependencies dataset and gives an explanation as to how the candidate pattern extraction process was done. Furthermore, Chapter 5 delves into how the MDL Principle was implemented, including our construction of a model and our encoding process. Chapter 6 then goes into the various experiments conducted and provides some visualisations of the results. Finally, Chapter 7 discusses some key aspects of the results and closes off this work with some limitations and future work, along with some concluding remarks.

Chapter 2

Preliminaries

This chapter introduces the foundational concepts relevant to this work. In particular, it presents the Minimum Description Length (MDL) Principle, which is crucial for selecting meaningful patterns. Secondly, this chapter provides the formal definition of a tree and a pattern used in later parts of this thesis.

2.1 The Minimum Description Length Principle

The Minimum Description Length (MDL) Principle [13, 12] is a fundamental concept for this thesis. In simple terms, it states that the best model to describe a piece of data is the one that compresses it the most. To put this formally, given a model family $\mathcal{M}$, the best model $M \in \mathcal{M}$ is the one that results in the lowest value for $L(M) + L(D|M)$, where $L(M)$ is the length of the model (in bits) and $L(D|M)$ is the length of the data encoded with M (again, in bits). Specifically, the type of MDL this refers to is known as *two-part MDL*, as the encoding process is done in two steps; the encoding of the model, and then the encoding of the data given the model.

Use of the MDL principle is often paired with model explainability. This is key in certain cases as after finding the best model, one can look and see what the model is comprised of and makes it the best model, unlike popular machine learning models such as neural networks. In addition, due to a core concept of the MDL principle being that the length of the model itself is taken into account, complex models that are prone to overfitting are heavily discouraged.

It is also important to note that despite mentioning terms such as compression and encoding, no actual compression is being done, we are simply interested in knowing how many bits it would take to encode the data if it were to be actually encoded. Furthermore, the MDL principle does not state how to go about "encoding" the data, as this can differ greatly depending on the type of data one is working with. However, there are some frequently used encoding methods that are relevant for this work [13] (all logarithms are base 2 unless specified otherwise):

- An element x such that $x \in X$ where X is a set and x is chosen with uniform probability has a description length of $log(|X|)$ bits [18].
- An integer $n \in \mathbb{N}$ with an unknown upper bound has a description length of $L_{\mathbb{N}}(n)$ bits. The process $L_{\mathbb{N}}(x)$ is known as Universal Integer Encoding, and the specific function used in this work is known as Elias delta encoding [20].

- A sequence of elements S, consisting of items $\langle s_0, s_1, ..., s_{|S|-1} \rangle$ where $s_i \in X$, and X is a set, has a description length of $L_{preq}(X, S)$ bits. This is known as Prequential Plugin Code [12, 18]. It allows for the encoding of sequences of items one by one, without prior knowledge of the probability of each item. The order in which items are presented in the sequence also does not matter. The formula given for this can be seen in Equation 1.1 [18]. Within the formula, Let S^i denote S up to and including the element at position i. Next, we define $usg(x, S)$ to be the number of times the element x appears in S. Lastly, ϵ is an initial value assigned to each element to avoid a count of 0. For most work, including this one, $\epsilon = 0.5$ is used.

$$L_{preq}(X, S) = -\log \left(\prod_{i=0}^{|S|-1} \frac{usg\left(s_i, S^{i-1}\right) + \epsilon}{\sum_{x \in \mathcal{X}} \left[usg\left(x, S^{i-1}\right) + \epsilon\right]} \right) \tag{2.1}$$

2.2 Tree Definition and Mining

Formally, a tree T can be defined as $T = (V, E, V_L, E_L)$ where V is the set of vertices (nodes) in the tree and E is the set of edges, such that each edge connects two nodes $E \subseteq V \times V$. In addition, if T has $|V|$ nodes and $|E|$ edges, then $|E| = |V| - 1$. V_L and E_L are functions that map each node to a label $V_L : V \to \mathcal{V}$ and $E_L : E \to \mathcal{E}$, where $\mathcal{V}$ and $\mathcal{E}$ are the set of node and edge labels, respectively. A label is a particular value attached to each node and each edge. Each tree has a depth d which describes the number of layers present. Therefore, let V_i where i is an integer between 0 and d represent the set of nodes at depth i. There can only be one node at $d = 0$, this being the root node, and thus, making these trees rooted trees. Lastly, let C_v signify the set of nodes that are children of node $v \in V$.

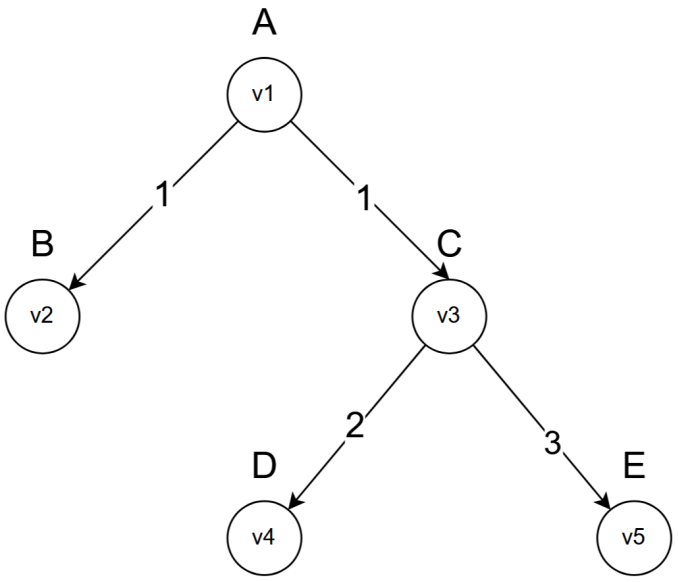


Figure 2.1: Example Tree

An example tree can be seen in Figure 2.1. Within this example: $V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{(v1, v2), (v1, v3), (v3, v4), (v3, v5)\}$, and an example of the label functions being: $V^L(v1) \rightarrow A$ and $E^L(v1, v2) \rightarrow 1$. The root node is $v1$ as it has no incoming edges. Its label is A, and its children $C_{v_1} = \{v_2, v_3\}$. Lastly, the tree has 3 layers, so $d = 2$.

As trees are a data structure, naturally, certain patterns may emerge within the structure of any particular tree. The extraction of these patterns is known as mining and specifically, tree mining. Naturally, since a tree is just a graph with restrictions on structure, any graph mining algorithms will also work on trees. Many methods for graph mining exist [21], all with different properties depending on what the user prioritises when mining, or the type of trees/graphs being used. Furthermore, as patterns are simply repetitions of identical parts of a tree, each pattern is also a tree in itself.

Formally, a pattern $T^P = (V, E, V_L, E_L)$ has the same properties of a tree previously defined. They are frequently occurring subtrees within the original trees themselves. Each pattern has a frequency f that states how often the pattern occurs within all the trees mined, and a support s which indicates how many trees it occurs in. The key difference between frequency and support being that if a pattern occurs twice in the same tree, its frequency would be incremented by 2, but its support will only be incremented by 1.

To give a formal definition of frequency and support: frequency $f(T^P) = \sum_{i=1}^{N} count_{T_i}(T^P)$ where N is the number of trees in the data and where $count_T(T^P)$ gives the number of occurrences of T^P in any given tree T. On the other hand, support can be defined like so: $s(T^P) = |\{i \in \{1, 2, ..., N\} | count_{T_i}(T^P) \geq 1\}|$.

Chapter 3

Related Work

This chapter provides a review on the relevant literature for this thesis. Several works are discussed, firstly, those relating to language classification, then moving on to pattern mining and the various pattern mining algorithms that exist and finally, how the MDL principle has been applied in different ways to tackle the pattern explosion issue.

3.1 Language Classification

Whilst there currently is not any work that also aims to find similarities and differences between languages in our exact manner, there has been work done on language classification. Language classification aims to identify what language some text belongs to given a set of languages to choose from. Botha et al. [1] aimed to classify text as one of the of the 11 official languages of South Africa. They used various classifiers for their work and found that with few words, their models were able to accurately classify what language they belonged to. Keep in mind that all of these methods used words themselves, whilst we only use the structure of various sentences whilst discarding the lemmas.

In addition, Ölvecký [2] aims to also classify text in various Slavic languages (Polish, Czech and Slovak). The author found that their method, which utilised a modified N-gram based approach could achieve up to 95% accuracy in classification. Furthermore, Vatanen et al. [3] also used N-gram models to classify languages from short texts, however, they found issues with their models especially with regards to model size and complexity.

3.2 Text and Graph Mining

Given that our data relates to languages, it is important to first discuss some work done on pattern mining on language data. This often overlaps with Text Mining, as text is nearly always written in some form of language. There has been work done for pattern mining with use of Natural Language Processing (NLP) techniques [5]. The authors' aim was to discover patterns within linguistically pre-processed text. They tested two pattern mining algorithms, however, only one of them, *GenPrefixSpan* [22] produced meaningful patterns. It should also be noted that the authors only tested their work on one language, that being Portuguese.

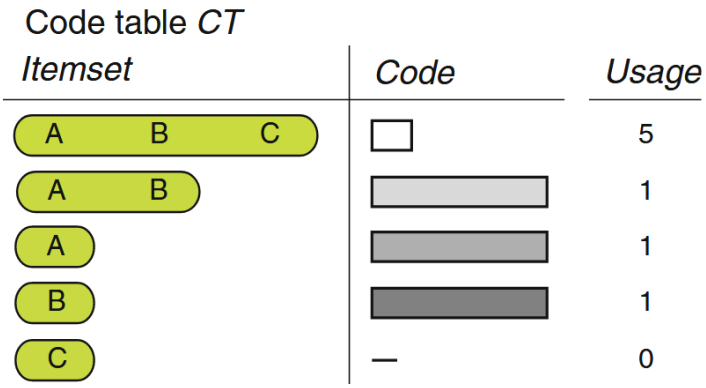


Figure 3.1: KRIMP Code Table. Taken from [14]

Next, regarding graph mining, various graph mining methods have been developed, one of which being gSpan [6]. It is relatively simple in nature and was found to be able to handle large sets of graphs, making it ideal for this work. The authors also found that their methods outperformed an existing method at the time: FSG, by an order of magnitude.

Gaston [7], on the other hand, is also a solid candidate. Its approach aims to split the graph mining process into three steps, going from simplest to most complex. These are, finding frequent paths, then frequent trees and finally cyclic graphs. As one of the steps is extracting trees themselves, it is quite an efficient method for tree mining despite it being a graph miner.

In addition, there are actual rooted tree mining algorithms too, such as TRIPS and TIDES [23], two algorithms which are very adjustable and can work with most kinds of rooted trees. The authors also found that their approaches outperformed state-of-the-art algorithms at the time by several orders of magnitude.

3.3 MDL for Pattern Mining

One of the biggest issues with pattern mining, is that they extract *too many* patterns. This makes analysis quite tedious, computationally expensive, and sometimes infeasible. Thus, the MDL Principle can be applied in order to reduce the pattern set.

3.3.1 KRIMP

One foundational algorithm in this field is KRIMP [14] which deals with mining itemsets and using MDL to deal with pattern explosion. KRIMP makes use of a code table. which consists of Itemsets that are mapped to a particular code. An example of a code table in KRIMP can be seen in Figure 3.1.

KRIMP also introduces two fundamental algorithms for applying the MDL principle to their data. These being the Cover and Search algorithms. The Cover algorithm tries to fit the patterns into the data such that the data can be reconstructed solely from the patterns. On the other hand, the Search aims to construct several models (code tables) and compares them in order to find the one which minimises the total description length. The authors found that

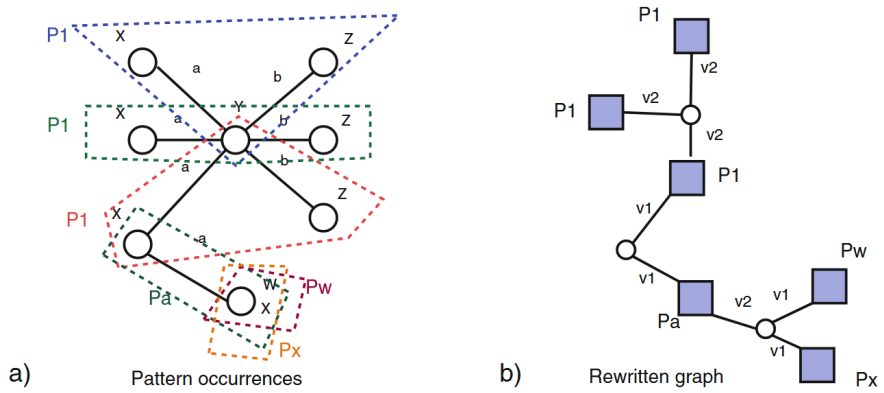


Figure 3.2: Creation of a Rewritten Graph (b) from the pattern occurrences in the original graph (a). Taken from [16].

KRIMP significantly reduced the pattern set by several orders of magnitude, and that the code tables generated were of a high quality.

3.3.2 GraphMDL

GraphMDL [16] aims to adopt KRIMP, however, in place of itemsets, graphs are mined instead. It introduces the concept of a rewritten graph, essentially a reconstructed form of the original graph that shows the patterns present within and how they link to each other. An example of such can be seen in Figure 3.2. The square nodes in the figures represent the patterns, and are known as embedding nodes, the circular ones are known as ports, and are used to highlight which vertex within each pattern is shared in another. The authors found that using this approach significantly reduced the pattern set and found the patterns to be meaningful and representative of the original data.

3.3.3 SLIM

SLIM [15] is another type of algorithm when it comes to pattern mining. It takes a different approach to KRIMP and aims to improve upon KRIMP's drawbacks. SLIM mines pattern sets directly from the data and then are constructed from the bottom up, joining co-occurring patterns to potentially increase the compression rate. The authors found that SLIM mines high quality patterns whilst evaluating far fewer candidates than KRIMP. Furthermore, it is worth noting that a version of GraphMDL called GraphMDL+ [17] exists, which is based on SLIM instead of KRIMP. GraphMDL+ was found to achieve results compared to that of its predecessor in a much faster time.

3.3.4 DIFFNORM

The concept of using the MDL principle to find similarities and differences within data is not a novel one. In their work, Budhathoki and Vreeken develop the DIFFNORM algorithm [19], which is able to find differences and similarities (norms) between patterns in databases. A pattern would then be a tuple of items such as $bread, butter$. This differs compared to an approach like KRIMP [14] and GRAPHMDL [16] as instead of considering itemsets or sets of graphs, DIFFNORM considers sets of itemsets. This is similar to how our data is structured, being in the form of trees in various languages. Think of each language as a set of trees, and since there are multiple languages being considered, multiple sets of trees are needed.

Chapter 4

Data & Data Processing

This chapter discusses the principal dataset used, Universal Dependencies (UD) and discusses how the treebanks within are mined for candidate patterns.

4.1 Universal Dependencies

Universal Dependencies [10, 11] is a collection of over 200 treebanks across more than 150 different languages. The project is an open community effort consisting of over 600 contributors. The contributors aim to provide consistently annotated treebanks for several natural languages. The purpose of which is to promote multilingual parser development, cross-lingual learning and parsing research from the point of view of language typology [10].

Within UD, multiple datasets of treebanks are available per language. There are several treebanks present in UD, some of which span multiple languages and are therefore parallel corpora. One such corpus, PUD (Parallel Universal Dependencies), contains trees for the same exact 1000 sentences in multiple languages, where each tree therefore represents a single sentence. The sentences present within PUD were taken from news and Wikipedia documents. The parallel nature of the data ensures consistency in regards to patterns found, as the only similarities and differences that emerge should only do so due to linguistic similarity or dissimilarity, and should not be influenced by the actual text present for each language. For this thesis, only the PUD (Parallel Universal Dependencies) dataset is used.

4.1.1 Tree Structure and Annotation

The trees within UD are rooted trees, an example of which can be found in Figure 4.1, with a more readable version seen in Figure 4.2. Each tree represents a sentence in the original corpus. As can be seen, each node label represents a Part of Speech (POS) tag of a word's syntactical function within the sentence, such as the word *dog* being a noun or *chased* being a verb. Furthermore, each edge label shows the syntactic relation between those two nodes. The link between the aforementioned nodes has the label *nsubj:pass*, which indicates that the noun *dog* is the passive nominal subject of the verb *chased*. These edges therefore represent syntactic dependencies between words. There are a number of POS tags for nodes, which can be seen in Table 4.1. Regarding edge labels, there are far too many to reasonably list here, therefore only a short table, Figure 4.2, is provided in this chapter. The comprehensive list

is shown in Appendix A. It is also worth noting that the annotations in UD are created by multiple human annotators. Despite their best efforts to ensure that annotation is consistent (hence the term universal), some minor underlying variation in labelling is inevitable, despite efforts to maintain consistency.

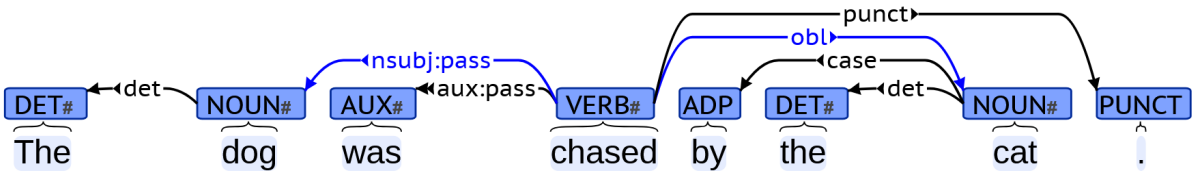


Figure 4.1: Universal Dependencies Tree Example. Taken from [10]

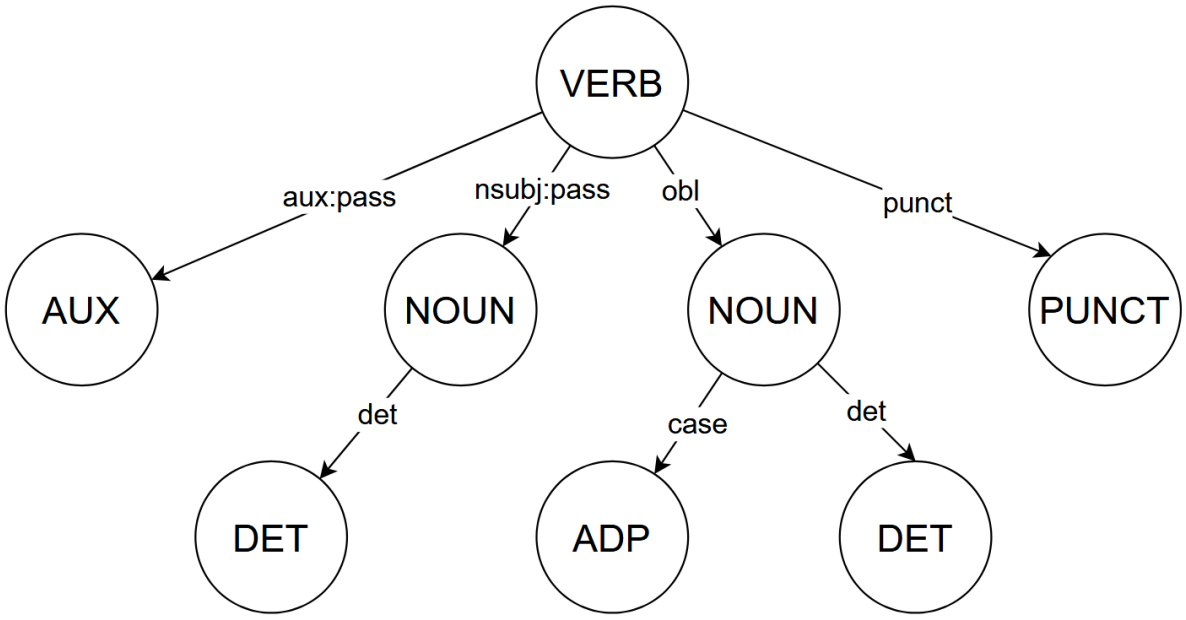


Figure 4.2: UD Tree Example (Figure 4.1) presented in tree-like fashion.

When downloaded, the UD treebanks are in the form of CoNNL-U files which contain descriptive information on each sentence and word in such sentence. These CoNNL-U files are then processed into edge lists (a type of file format commonly used when processing trees or graphs) for use in candidate extraction.

4.2 gSpan

As the data aspect was addressed, the next step is deciding on which pattern mining approach to use. Any tree or graph miner would be suitable, so long as it can extract candidate patterns in a reasonable amount of time. [6] was ultimately selected for its ease of implementation due to an already existing Python library [24], in addition to its quick runtime, where running the algorithm on one treebank would take around 15 minutes. The gSpan algorithm is designed to explore frequently occurring subgraphs from a graph database, whilst tackling the issues of

POS Tag	Linguistic Function
ADJ	adjective
ADP	adposition
ADV	adverb
AUX	auxiliary
CCONJ	coordinating conjunction
DET	determiner
INTJ	interjection
NOUN	noun
NUM	numeral
PART	particle
PRON	pronoun
PROPN	proper noun
PUNCT	punctuation
SCONJ	subordinating conjunction
SYM	symbol
VERB	verb
X	other

Table 4.1: UD POS tags and their linguistic functions

Edge Label	Linguistic Function
cc	coordinating conjunction
cop	copula
aux	auxiliary
det	determiner
case	case marking
nmod	nominal modifier
nsubj	nominal subject
obj	object
punct	punctuation

Table 4.2: UD dependency edge labels and their linguistic functions (A comprehensive list can be found in Appendix A)

candidate generation and subgraph isomorphism. With both of these being traditionally, very costly to overcome.

The algorithm makes use of DFS (Depth First Search) codes, a string of text that represents a subgraph's structure, and which is generated by means of a DFS traversal. Normally, for any given graph, multiple DFS traversals are possible, and thus multiple DFS codes can be generated. The gSpan algorithm, however, accounts for this by imposing a specific lexicographic order for graph traversal, such that isomorphic graphs are not generated more than once. This results in each subgraph having a minimum DFS code, which can also serve as a unique identifier.

gSpan organises its search space into a lexicographic DFS tree, where each node corresponds to a subgraph, which in itself is represented by a DFS code. Subgraph patterns are then extended recursively along the rightmost path of the DFS traversal. Furthermore, each extension can add either a forward edge or a backwards edge, allowing for complex cycles to be represented. Although, in our case, these complex graph structures are naturally, not present. After generating a candidate pattern by extending the current subgraph along the rightmost path, gSpan determines the support of the candidate by counting in how many graphs of the input dataset it occurs as a subgraph. If this count meets or exceeds the minimum support threshold, the candidate is considered frequent and is also eligible for further extension.

The main advantages of gSpan are its relative simplicity and speed, in addition to its capabilities of mining patterns in large datasets. Although the algorithm is by no means new, and there have been faster algorithms developed, for the purposes of this work, it is fast enough.

4.3 Candidate Extraction

As previously mentioned, a Python library [24] existed that had already implemented the gSpan library. Although the algorithmic base was there, some testing and modifications had to be done in order to get it fully working. Firstly, the library did not work out of the box, and some pieces of code had to be updated in order to work at all. In addition, there were some concerns that it would not work with directed graphs, as the GitHub repository stated that it had not been exhaustedly tested on them. However, after conducting some testing ourselves on a small artificial and a reduced version of some UD treebanks, no issues were found with mining patterns in directed trees. The modified library can be found here[1].

In addition to bug fixes, the library was modified to keep track of frequency as well as support, as we are still interested in the overall frequency of graphs. It should also be noted that the minimum support threshold for mining was set to 5 for mining all language trees. As each language contains 1000 trees, this is a threshold of 0.5%.

[1]Modified gSpan repository: https://github.com/druxu/gspan

Chapter 5

Implementation of the MDL Principle

This chapter discusses the main contribution of this thesis. It delves into some foundations for this work, including some definitions and the cover algorithm, the encoding process used and finally, the search algorithm.

5.1 MDL Groundwork

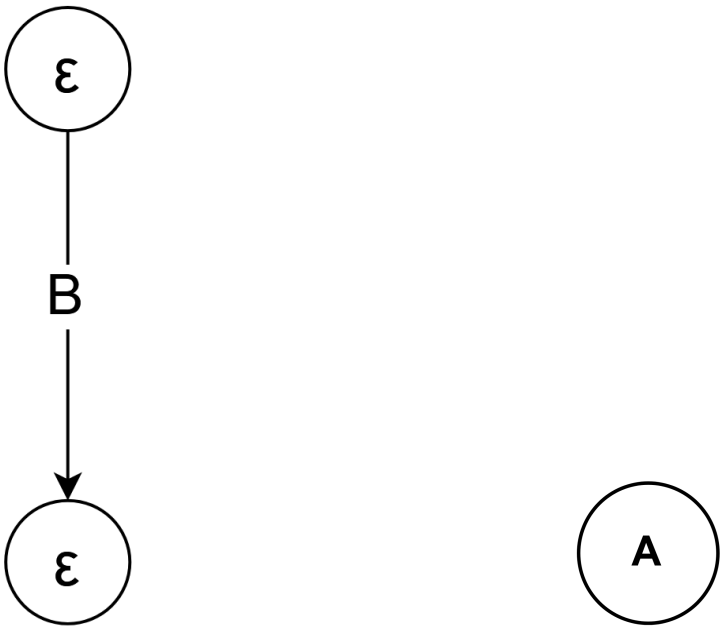
This section provides some core definitions and concepts that the rest of this work is built upon. Firstly, we discuss our definition of a model, and singleton patterns. We then introduce the concept of a rewritten tree, and finally, delve into our Cover algorithm.

5.1.1 Definitions

Model Definition: There are two types of models we need to define. The first one being the global model M. It contains a collection of local models m, which pertain to a particular set of languages L and a set of patterns associated with those languages T^P. Furthermore, let m_L refer to a specific local model that contains all the patterns associated with language set L and let $\mathcal{L}$ refer to all languages present within a global model M. It is worth noting that there exists a model for each combination of languages within $\mathcal{L}$ Moreover, the local model that contains all patterns pertaining to all languages present in the global model is known as the common model, $m_{\mathcal{L}}$. This model also contains all singleton patterns. Lastly, $\mathcal{V}$ and $\mathcal{E}$ represent the set of node and edge labels, respectively, within all patterns present in the data.

Singleton Pattern: A singleton pattern is a specific type of pattern introduced such that the cover algorithm can encode the entire original graph, as the MDL principle only deals with lossless compression. There are two types of singleton patterns present, these being node singleton patterns and edge singleton patterns. An example of which can be seen in Figure 5.1. In this figure is an example for a node singleton with label *A* and an edge singleton example with label *B*. The nodes in the edge singleton are labelled with ϵ. This is an artificial label manually added into $\mathcal{V}$ to represent any node label. As this pattern is meant to cover an edge, using a placeholder value for the node labels allows us to store much fewer edge singleton patterns, and therefore, leads to a more efficient encoding process. Note that this is the only

case where the label ϵ is used.



(a) Edge Singleton Pattern (b) Node Singleton Pattern

Figure 5.1: Examples of singleton patterns: (a) Edge singleton, (b) Node singleton.

5.1.2 Rewritten Trees

A rewritten tree is a reconstructed tree that shows the patterns present within the original tree and how they link to each other. However, before we present the formal definition of a rewritten tree, it is important to discuss the inspiration drawn from GraphMDL with their rewritten graphs. A rewritten graph is a a reconstructed version of an original graph, that shows the presence of patterns within and how they link to each other. A visual example can be seen in Figure 3.2. This type of graph is constructed after the cover algorithm is computed. To recapitulate slightly, there are two types of nodes present in a rewritten graph, port nodes and embedding nodes. Embedding nodes represent the patterns themselves, and ports represent the nodes shared between patterns.

Rewritten Tree Definition: The rewritten tree is exactly the same as GraphMDL's rewritten graph, instead of graphs, trees are used. Thus, a rewritten tree can be defined like so: $T^R = (V^R_{emb}, V^R_{port}, E^R, V^R_L, E^R_L)$. V^R_{emb} and V^R_{port} are the sets of embeddings and port vertices present within the tree. Furthermore, E^R is the set of edges and V^R_L and E^R_L are the sets of node and edge labels, respectively. In this context, the node labels are pattern IDs, thus mapping patterns to the vertices of the rewritten tree. Lastly, the edge labels are between embedding and port vertices, and indicate the port vertex in the original tree that is shared between the patterns.

5.1.3 The Cover Algorithm

This algorithm's purpose is to check which patterns out of several candidates can be used to represent the original data and create a corresponding rewritten tree. It checks each pattern and sees if it occurs in the data, if it does, mark the corresponding nodes and edges as covered and repeat. If, at the end of the process, there are still some uncovered nodes and edges, singleton patterns are created in order to cover them. This algorithm can be seen in Algorithm 1. Lastly, it is important to stress that patterns are considered by order of size (the number of nodes in a pattern) in descending order, such that to allow larger patterns to cover more of the original graph.

Algorithm 1 Cover Algorithm

Require: A list of trees $\mathcal{T}$, and a model M containing patterns
Ensure: Each tree is covered using existing patterns or new singleton patterns

```
Sort patterns in M by size (number of vertices), largest first
Initialise a counter for singleton pattern IDs
Initialise a list of singleton patterns
for each tree T in 𝒯 do
    Reset the set of used edges in T
    Select patterns compatible with T's language
    for each pattern p in the selection do
        if p can be mapped onto T without overlapping covered edges then
            Cover the matched subgraph in T with p
            Increase p's usage count
        end if
    end for
    Identify any uncovered edges and nodes in T
    while uncovered edges remain in T do
        Increment usage of relevant edge singletons
        Mark edges as covered
    end while
    while uncovered nodes remain in T do
        Increment usage of relevant node singletons
        Mark nodes as covered
    end while
end for
```

5.2 Encoding Process

The encoding process is made up of two main parts. The encoding of the model M and the encoding of the data D given model M such that $L = L(M) + L(D|M)$, where L is the total length of both.

5.2.1 Model Encoding

To calculate the length a global model M, we must sum over the length of the local models within using universal integer encoding, as we do not know the size of these local models. For each local model, we encode the number of patterns within, again, using universal integer encoding and we also encode each pattern itself. The equation for which can be seen in Equation 5.1.

$$L(M) = \sum_{m \in M}^{|M|} \left(L_{\mathbb{N}}(|m|) + \sum_{T^P \in m}^{|m|} L(T^P) \right) \tag{5.1}$$

$$L(T^P) = \sum_{i=0}^{d} L_{\mathbb{N}}(|V_{i+1}|) + \sum_{v \in V_i}^{|V_i|} \log(|V_{i+1}| + 1) + |V| \log(|\mathcal{V}|) + |E| \log(|\mathcal{E}|) \tag{5.2}$$

The process for encoding a pattern can be seen in Equation 5.2. Firstly, we encode the structure of the tree. This is done by summing over each layer i in the tree. For each layer $i + 1$, we encode the number of nodes at that layer using a universal integer encoding, as the total number of nodes is unknown. Since the encoding process continues layer by layer until a layer contains no nodes, the depth of the tree is determined implicitly without requiring a separate encoding of depth. Furthermore, as the trees are rooted, there is no need to encode the existence of the root node itself.

Next, for each node in layer i, we encode which node in layer $i + 1$ is its child. This step connects the nodes encoded in the previous step by specifying the parent-child relationships within the tree, therefore encoding the entire tree's structure. Furthermore, to avoid $\log(0)$, normalization is performed by adding 1 inside the logarithm.

The second part of this equation: $V \log(|\mathcal{V}|) + E \log(|\mathcal{E}|)$ deals with encoding the node and edge labels. To encode the specific label used for each node and edge, we must first encode the size of the set of all possible node labels $|\mathcal{V}|$ and $\mathcal{E}$. Since there are $|V|$ nodes and $|E|$ edges, we multiply these two logarithms by $|V|$ and $|E|$, respectively.

5.2.2 Data Encoding

To encode the data D, which contains treebanks of several languages such that each treebank $d \in D$ representing language l. The equation for which can be seen in Equation 5.3.

$$L(D|M) = \sum_{d \in D}^{|D|} L(d|M^l) \tag{5.3}$$

Where $M_l = \bigcup \{m_L \in M \mid l \in L\}$. In layman's terms m_l is the union of local models whose language set contains language l. Following this, the data can be encoded as follows:

$$
\begin{aligned}
L(d|M_l) = \sum_{T \in d}^{|d|} \left(1 + \sum_{v \in V}^{|V|} L_{\mathbb{N}}(|C_v|) \right) \\
+ L_{preq}(T^P \in M^l, \langle \cup_{T \in d} V_L \in T \rangle) \\
+ \sum_{T^P \in M^l} L_{preq}(V \in T^P, \langle \cup_{T \in d} e \in E_L^R \rangle)
\end{aligned} \tag{5.4}
$$

This encoding can be split into two main parts. The initial summations on the first line of the equation encode the structure of the rewritten tree. For each tree present within a given language, encode whether the root is a port or not, which is represented as the 1 in the equation. This information is only needed for the root node, as port vertices can only be connected to root vertices and vice versa, therefore, knowing whether the root is a port or an embedding node is enough to know the type of all the nodes. Next, for each child node, encode the length of its children, and repeat until the number of children for each node has been encoded.

The second part of the equation deals with encoding the node and edge labels respectively, where Prequential Plugin codes are used for both. Firstly, to encode the node labels, which are simply pattern identifiers, we provide a set of all patterns relevant for the rewritten tree's language l. The sequence provided is a union of all the node labels seen amongst all rewritten trees in d. Lastly, to encode the edge labels, which indicate which vertex in any pattern is a port. To do this, we sum over each relevant pattern and for each one, we compute the Prequential Plugin code, providing the set of nodes for each pattern and a union of all edge labels seen amongst the rewritten graphs. This method ensures that the most frequently occurring labels are encoded optimally.

5.3 The Search Algorithm

The final major algorithm present is the Search algorithm. It is responsible for generating many candidate models and iteratively adding patterns (mined by gSpan) and computing the total description length each time a pattern is added. If the length goes down after a pattern is added, the model is labelled as the best model and the process repeated until no more patterns are left.

The basic outline for our algorithm can be described as follows. It takes the candidate patterns, the tree data and the list of languages as inputs. It initialises an empty best global model and covers the data with only singleton patterns. This gives us a maximum value for the best description length. Next, candidate patterns are evaluated one by one. The algorithm attempts to add each candidate pattern to every local model in the current best global model, and the addition that results in the best reduction is finalised. If no reduction is possible for that candidate pattern, then that candidate is skipped. This process is then repeated until all patterns have been evaluated. After all candidate patterns have been checked, the best model and best length are returned, along with the evaluation history.

Whilst the approach just described works, it has some practicality issues. Firstly, as each candidate global model created contains several local models, which model to add a pattern to is a legitimate question. Therefore, the search algorithm exhaustively tries to add the pattern to each model and checks the total description length. If one were trying this algorithm with 6 languages, this means 63 models need to be tried for each pattern, where there can be over 5000 patterns for that amount of languages. The formula for this being $2^n - 1$ where n is the number of items in the set. Naturally, this is infeasible to compute naively, and therefore, we made use of parallelisation.

Firstly, the process of trying to add each candidate pattern to each local model is done completely in parallel, as one can imagine. However, in addition to this, a batch size was introduced which grows quite slowly, but steeply, in order to process patterns in larger sizes (and in parallel). The rate at which the batch size grows is controlled by the number of candidate patterns there are. The more patterns there are, the slower it grows. Initially, it pays us to compute each pattern one at a time if it is likely to get added to the model, however, if towards the tail end, each candidate pattern is getting rejected, it would be more efficient to process them in batches, such that if no pattern in the batch is accepted, they can all be discarded. This is the case due to the order that candidate patterns are fed into the search algorithm, being frequency based in descending order. Unfortunately, however, the computation time is still incredibly high, and most patterns were found within the first few hours as it follows an exponential decay. Thus, a maximum time limit of 6 hours was set, after which, the algorithm will wait for the current batch to finish processing and then return the current best model and best length. The pseudocode for our algorithm can be seen in Algorithm 2.

Algorithm 2 Search Algorithm

Require: Candidate patterns $\mathcal{P}$, data D, list of languages $\mathcal{L}$
Ensure: An improved global model with minimal total description length

Initialise the best model M_{best} with no patterns
Cover trees with M_{best} and compute initial description length L_{best}
Initialise the growth rate to $\frac{1}{|\mathcal{P}|}$
Initialise the batch size at 0, and have it grow according to the previously set rate
while there are remaining patterns **do**
    Select a batch of patterns based on the batching strategy
    Results $\leftarrow$ [ ]
    for all pattern p in batch (in parallel) **do**
        for all language subsets $\mathcal{S} \subseteq \mathcal{L}$ where p is applicable (in parallel) **do**
            Create a copy of M_{best} and insert p into it for language set $\mathcal{S}$
            Cover trees with this model, compute new description length L
            results $\leftarrow$ results $\cup(L, M, p\mathcal{S})$
        end for
    end for
    Select the result with minimal length $L_{\min}$
    if $L_{\min} < L_{\text{best}}$ **then**
        Update $M_{\text{best}} \leftarrow M$ and $L_{\text{best}} \leftarrow L_{\min}$
        Remove the selected pattern p from $\mathcal{P}$
    else
        Remove all batch patterns from $\mathcal{P}$
    end if
    if time limit exceeded **then**
        break
    end if
end while
return best model M_{best}, length L_{best}, and evaluation history

Chapter 6

Experimental Evaluation

This chapter provides details on the experiments done, which specific treebanks were used throughout said experiments, and also discusses the results of those experiments. Note that all experiments were done using Python, and on a Linux server running Rocky OS. The server has 512GB of RAM and 128 AMD Epyc 7702 CPU at 2GHz with 256 threads. Lastly, the GitHub repository can be found here. [1]

When running experiments, the first task was to determine which treebanks (languages) to include. Ideally, we would like to include as many as possible, however, as running the Search algorithm is quite computationally intensive and time consuming, this is not feasible. Initially, we selected three languages to run some basic experiments on. These being English, French and Italian, which were picked as we can understand these languages, and therefore if needed, we can refer back to the original text for further context. For the following experiments and until specified otherwise, these are the only languages that are being used, and is hereby referred to as the *small set*.

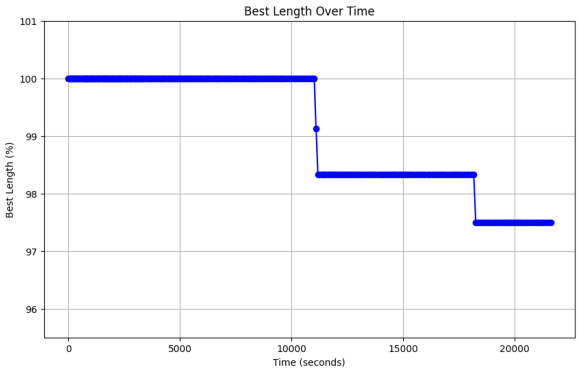
6.1 Candidate Pattern Order

The first experiment conducted was on the small set to determine the optimal order for feeding candidate patterns into the Search algorithm. Naturally, the order that patterns are fed in greatly affects the resulting best model, as it could be that a non-optimal pattern is added to the model, which then prevents other, potentially better, candidate patterns from being added. Three experiments were conducted, feeding the patterns in three different orders: frequency (descending), number of vertices (descending) and frequency (ascending). Intuitively, feeding the most frequent patterns first makes the most sense, as they would be often need to be used to cover the original data. This is also the same order that was used in KRIMP [14]. If some patterns share the same value for their respective criterion, they are then sorted by the other unused criterion in descending order. If they match on both criteria, they are then sorted by canonical code, so that the order they are fed in remains consistent between runs.

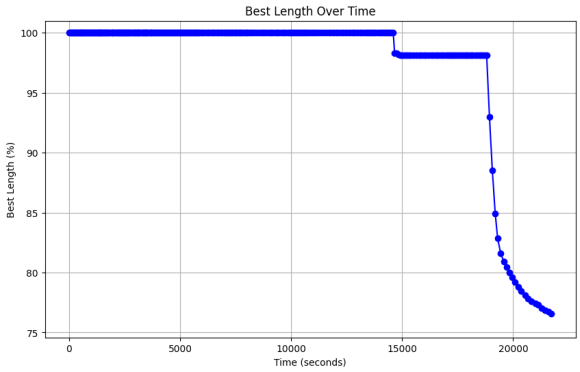
The best description lengths achieved for each order can be seen in Figure 6.1. As shown by the figure, starting the search algorithm with the most frequent patterns first by far yielded

[1] Thesis Repository: https://github.com/druxu/masters_thesis

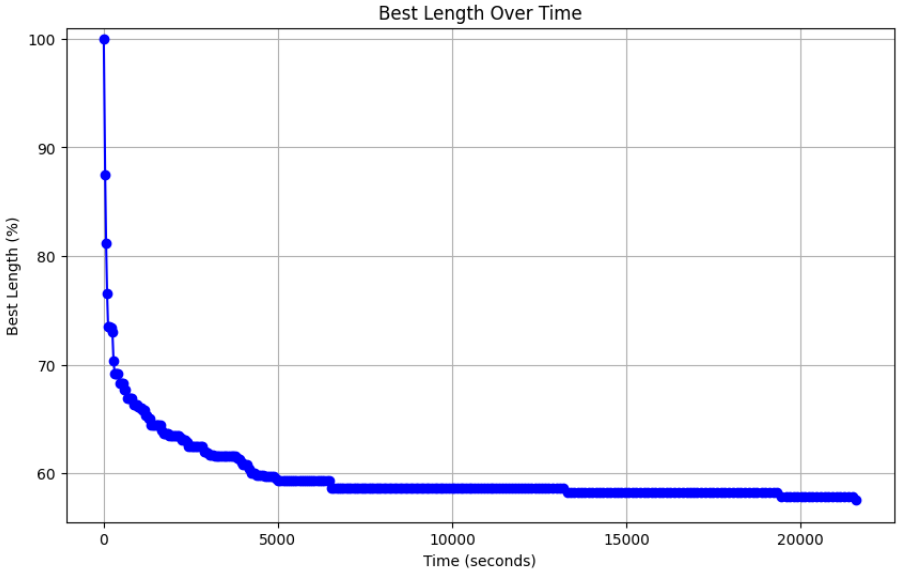
the best compression percentage of the original data with the best length achieved being just under 60% of the original length. Whilst (c) reaches a good description length rather quickly, and the others did not achieve good results at all, they may eventually reach a good description length if left longer than 6 hours. However, since (c) was the quickest, this order will be used for future experiments.



(a) Best length over time (Lowest Frequency First)



(b) Best length over time (Highest Number of Vertices First)



(c) Best length over time (Highest Frequency First)

Figure 6.1: Evolution of model length over time using different pattern ordering strategies. The best length achieved is relative to the description length of the initial empty model, which is the same for all subfigures. Note that the Y-scales differ per subfigure. For ease of reading, 5000 seconds $\approx$ 1.4 hours

6.2 Exclusion of Multi-Language Sets

One question we proposed is whether or not it is worth having local models representing multiple language combinations, as it greatly increases computation time. By default, we used these

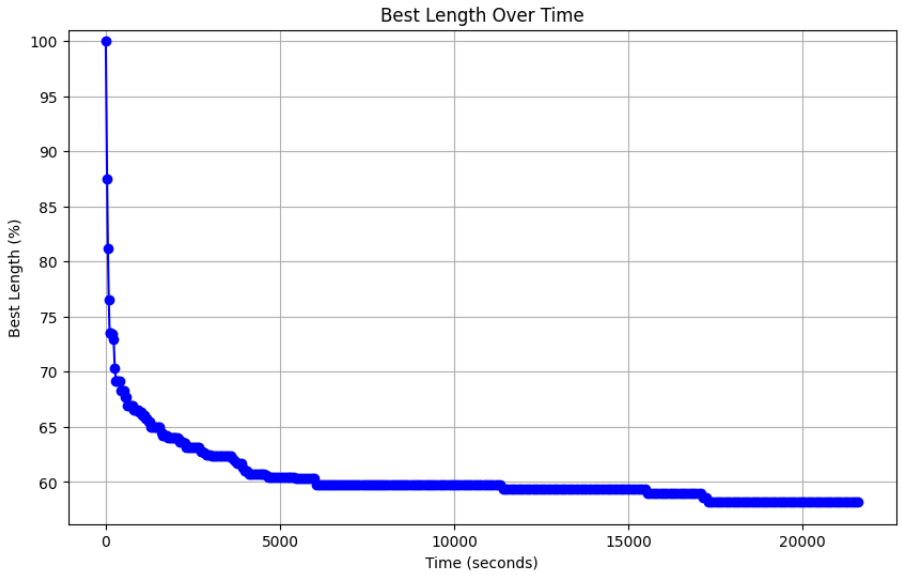


Figure 6.2: Best length over time (No Multi-Language Sets) (5000 seconds ≈ 1.4 hours)

MULTI-LANGUAGE SETS, however, we decided to disable them and run the Search algorithm to see how it might affect the resulting best description length. Keep in mind that the common model is still in use, as it is also used to store the singleton patterns.

In Figure 6.2, we can see how the description length decreases over time. It is comparable to Figure 6.1c, as the only difference between them being the inclusion of multi-language sets. There does not seem to be much of a difference at all between the best description lengths achieved. That being said, ultimately we would lose some interpretability of results if multi-language sets were to be excluded. As the inclusion of multi-language sets had seemingly no effect on the results, despite the much more intensive computation process, it can be said that the parallelisation of the Search algorithm is effective. Due to these observations, multi-language sets will be used in all future experiments.

6.3 More Languages

Naturally, the next logical step in the experimenting process is to try the algorithm with more languages and to see what the best model looks like. Three more languages were added to the existing set of English, French and Italian, these being: Czech, Icelandic and Arabic. Czech was chosen as it is still an Indo-European language, but is much further related to the current three as it is in the Slavic family. Icelandic was chosen as it is a part of the Germanic language family, like English, so it might be interesting to see if any patterns are shared between them. Lastly, Arabic was chosen to be a "dark horse". As it is a Semitic language, one would assume there would not be many commonalities between it and the other languages. The best length achieved over time for these languages can be seen in Figure 6.3. A maximum compression of just under 65% was achieved, with it likely being able to achieve a better one if more time was allotted.

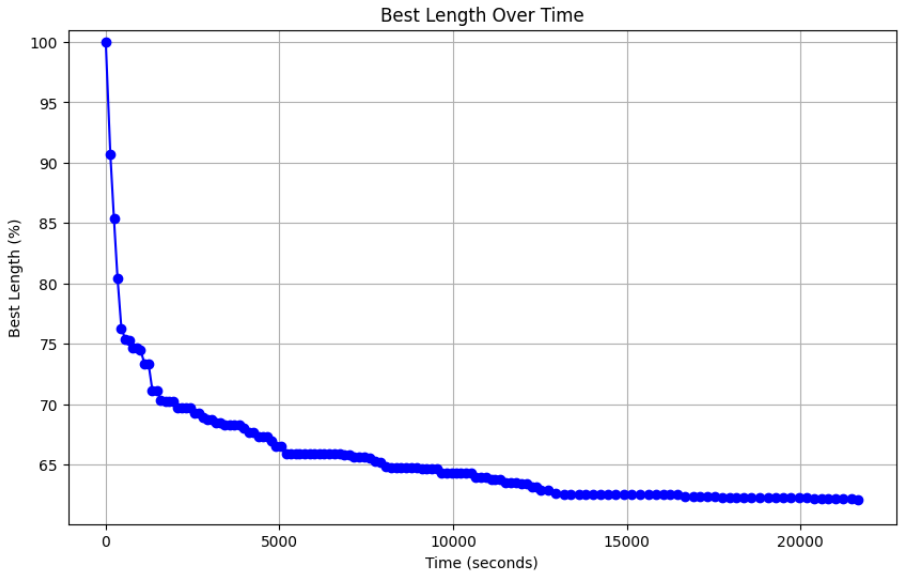


Figure 6.3: Best Length over Time (6 Languages) (5000 seconds ≈ 1.4 hours)

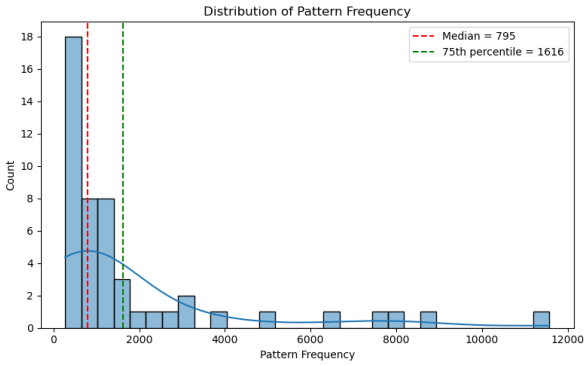
In addition to showing the best length over time, Table 6.1 shows the 10 most frequent patterns present within the best model. Within the table, it can be seen that most of these frequent patterns were added to the common model, which is to be expected. However, there were a few patterns that were added to different local models instead. Interestingly enough, the eighth most frequent pattern was added to the model for Arabic and Icelandic, which was unexpected as one would not necessarily think these two languages would have such a high frequency pattern in common. However, just because a pattern is found in a certain local model it does not necessarily mean that that pattern is only found in those languages pertaining to that local model. This is because the Search algorithm automatically determines which local model a pattern gets added to. Furthermore, one can see that the frequency does decrease a fair bit from the first to the last position, indicating that there are a small number of highly frequent patterns that could explain the exponential decay seen in Figure 6.3.

Node 1	**Node 2**	**Edge**	**Frequency**	**Support**	**Model**
NOUN	ADP	case	11572	4963	Common
NOUN	DET	det	8823	3196	Czech, English, French, Italian
VERB	PUNCT	punct	7836	4973	Common
NOUN	ADJ	amod	7596	4063	Common
NOUN	NOUN	nmod	6592	3436	Arabic, Czech, Icelandic
VERB	NOUN	obl	5093	3396	Arabic, English, French, Icelandic, Italian
VERB	NOUN	obj	3781	2941	Common
VERB	NOUN	nsubj	3235	2760	Arabic, Icelandic
NOUN	PUNCT	punct	3105	1941	Common
PROPN	ADP	case	2736	1980	Common

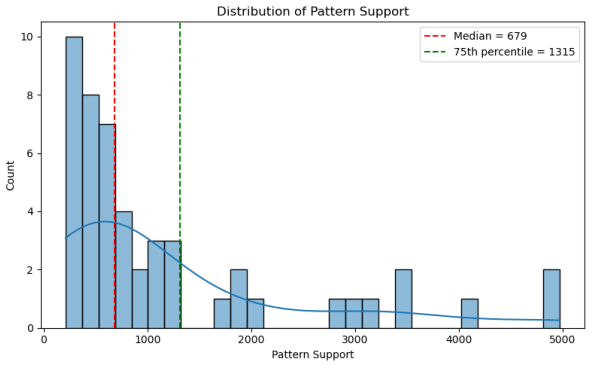
Table 6.1: Top 10 Most Frequent Patterns. All of these patterns only had 2 vertices. The Node and Edge columns show the labels node and edge labels respectively in each pattern. The Model column indicates which local model that pattern is present in.

6.3.1 Pattern Frequency, Support and Size Distributions

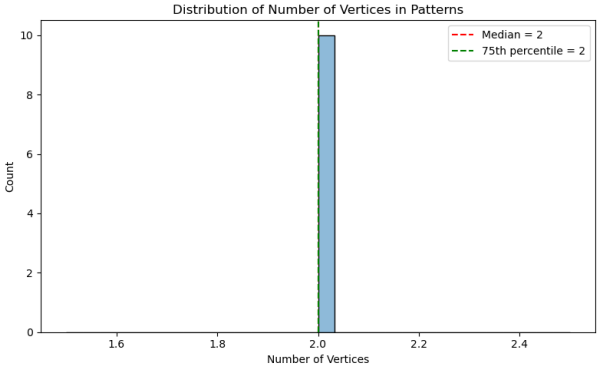
As a type of sanity check, to ensure that our Search algorithm correctly captures important, frequently occurring subtrees, we plotted the distributions for frequency, support and the number of vertices within each pattern in the best model found. In Figure 6.4, one can see these distributions. The frequency and support distributions are suitable, where a few patterns are extremely frequent (over 5000), and most are reasonably frequent. The median support was 679, as there are 6000 total trees, this means the median support was around 10% of the total. Interestingly enough, each pattern accepted into the global model only had 2 vertices. This was quite peculiar, and warranted further analysis.



(a) Pattern Frequency Distribution



(b) Pattern Support Distribution



(c) Number of Vertices in Patterns Distribution

Figure 6.4: Distribution statistics of selected patterns across frequency, support, and number of vertices.

One hypothesis as to why only patterns with 2 nodes were being selected, is that they happen to be the most frequent and thus, the algorithm selects them first. Following the selection of these few, highly frequent patterns, it could be that the remaining patterns that reduce the description length further also happen to contain just 2 vertices due to the patterns previously selected. We took a look at all the candidate patterns mined and plotted their size against their frequency. This can be seen in Figure 6.5. The pattern's size (number of vertices) is plotted on the y-axis, and the mean frequency on the x-axis. Mean frequency was plotted as each dot represents a collection of patterns that all share the same number of vertices. This was done as otherwise there would be thousands of dots overlapping each other, and would

therefore be difficult to interpret. The size of each dot also represents a value, this being how many patterns can be found with that size. This value scales linearly with the area of each dot. The plot shows that indeed, there are a reasonable amount of highly frequent patterns of size 2. Although there are more patterns of sizes 3 and 4, they are much less frequent on average, which then explains why all patterns selected only have 2 vertices.

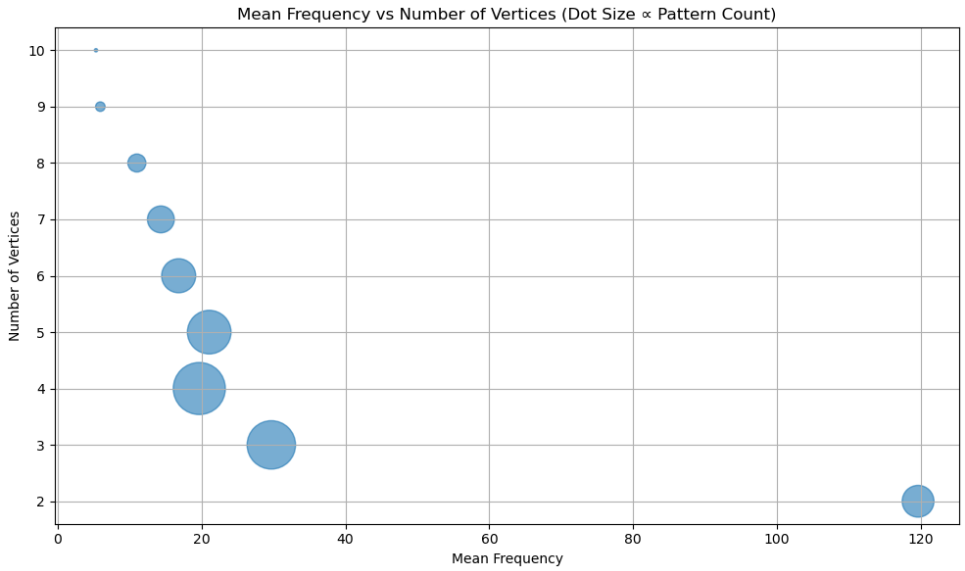


Figure 6.5: Candidate Pattern Size vs Mean Frequency

6.3.2 UpSet Plot

We then made use of an UpSet plot, a type of plot which allows one to visualise set intersections on a large scale in a more coherent way than a Venn diagram. This plot can be seen in Figure 6.6. Each black dot or black dot combination represents a set of languages. The bar on top of each set combination shows the size of the set, i.e. how many patterns are present within, and the bar on the left of each language represents how many patterns were added that correspond to that language across all the local models. Note that only non-empty local models are shown in the plot.

The most populous local model is the common model; unsurprisingly, since the most frequent patterns are fed first, these most frequent patterns are likely to be present in many if not all languages. The second biggest local model is the one with just Arabic, which aligns with expectations, since it is the one most distantly related to the other languages. Other notable combinations include French, Italian and English, and Czech, Italian, English and Icelandic, both with a total of three patterns. The global model as a whole has 49 patterns total. English, Italian and French also have the least number of patterns in their individual sets, meaning that most patterns present in these languages are also present in others. Note for this plot, the singleton patterns have been removed as they added clutter to the plot due to the sheer number of them (82). However, the plot including the singletons can still be seen in Figure 6.7.

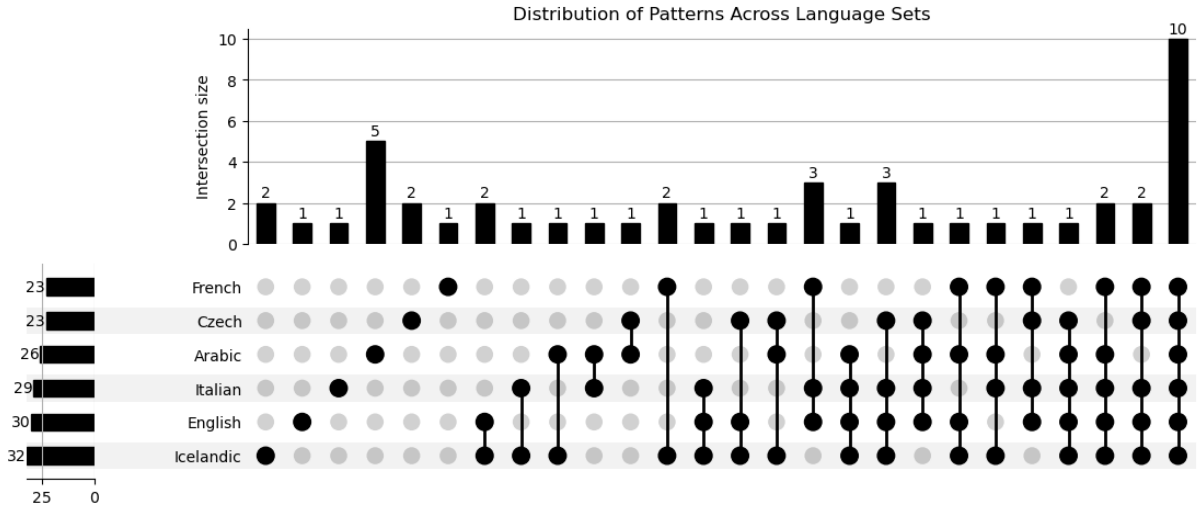


Figure 6.6: UpSet Plot for all Non-Empty Language Sets. (Singletons Excluded)

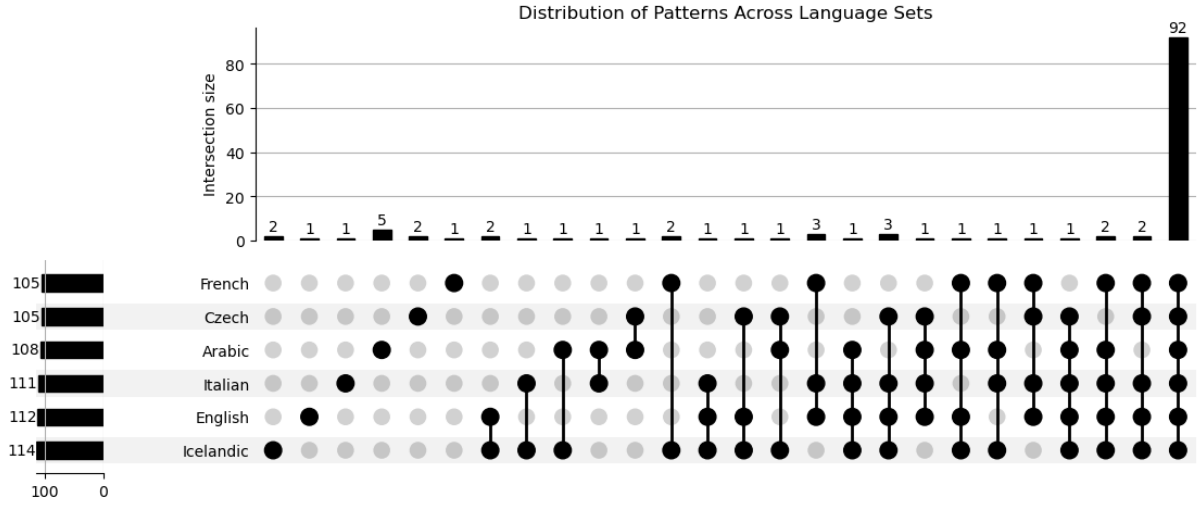


Figure 6.7: UpSet Plot for all Non-Empty Language Sets. (Singletons included)

6.3.3 Jaccard Similarity

In addition to the UpSet plot, we wanted a way to highlight which language pairs were the most similar. Whilst this information can be deduced from the UpSet plot, we decided to plot a Jaccard Similarity score heatmap, as seen in Figure 6.8. Jaccard Similarity is a statistic that calculates the similarity between sets. The score ranges from 0 to 1, where 0 means the two sets are completely different, and 1 meaning the sets are identical. The equation for Jaccard Similarity can be seen in Equation 6.1. It is computed by dividing the size intersection between sets A and B with the union of those same two sets.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.1}$$

By far, the language that had the highest similarity score with the other 5 languages was English, which is unsurprising given the prominence and influence other languages have had

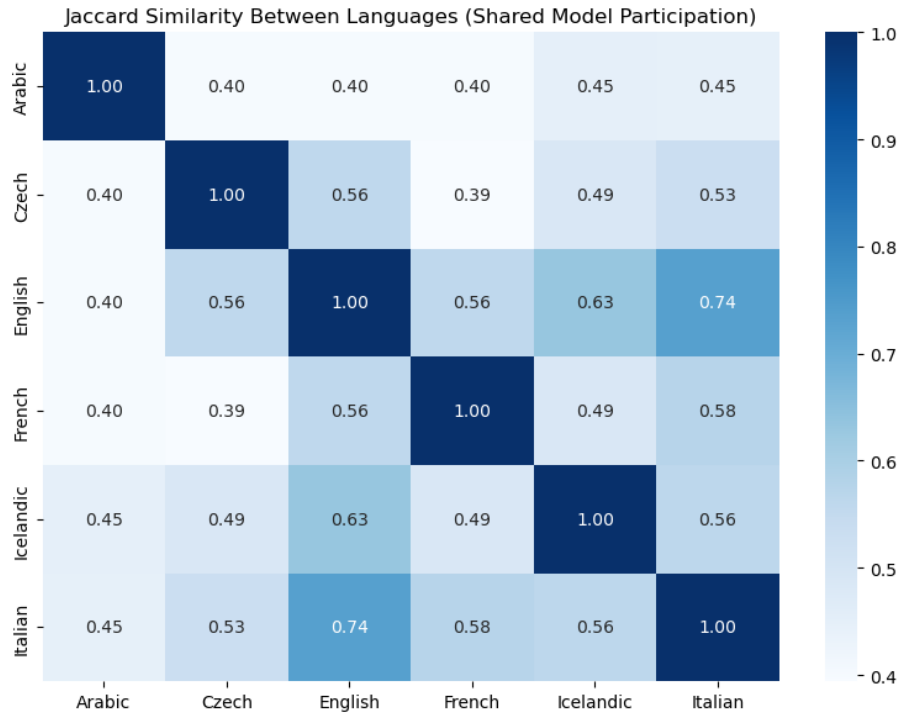


Figure 6.8: Jaccard Similarity Score between Languages

on English and vice versa. On the other hand, the language that was the most dissimilar to the others was Arabic, which is also to be expected. The language pair with the highest similarity was English and Italian, which is quite an interesting observation as intuitively, one might expect French and Italian or French and English to be higher as both French and Italian are Romance languages and French historically had lots of influence on the English language.

Chapter 7

Discussion and Conclusions

This chapter brings this thesis to a close with a discussion about the results, and whether the aim for this work has been successfully accomplished. Furthermore, we delve into some limitations and potential future work, before some brief concluding remarks.

7.1 Discussion

In the previous chapter, we showed some experiments done and the results obtained from such experiments. The analysis performed shows that the methods implemented do allow us to find meaningful patterns for each of the languages that highlight the differences and similarities between them. The results achieved were more or less expected, however, there are still some talking points in need of discussion.

Limited Pattern Size and Time Constraints

Most notably, the selection of patterns with only two vertices was unexpected. Granted, we found that patterns containing just 2 nodes were by far the most frequent, it is still a bit peculiar that no patterns with three or more vertices were found. In Figure 6.5 we can also see that there are many patterns with 3, 4 and 5 vertices that may have been interesting to explore. It is possible that the 6 hour time limit had an effect on this, as perhaps, more patterns would have been added had the 6 hour restriction been raised or even removed completely. An interesting add-on to this thought would be that perhaps if sorting by pattern size in ascending order might have shown similar results to sorting by most frequent due to the patterns accepted into the model only having 2 vertices.

Compression Ratios and Redundancy in Syntactic Structures

One other important point to mention is the compression ratios achieved by our algorithm. The compression percentage achieved (35-40%) shows that there is a substantial amount of redundancy in the syntactic structure of the original data, as the biggest reductions in description length are found in the first few patterns added to the model. Especially since reasonably good compression ratios are achieved in such a short time, with the graphs all following a

rough exponential decay.

Minimum Support Threshold

Furthermore, we mentioned the use of a minimum support threshold for pattern mining. We set this value to 5 as initially, we thought having as low a value as possible for this would be better, due to there being more candidate patterns to choose from. However, the patterns actually selected for the model tended to be rather frequent, so perhaps, it might have been a better idea to raise this minimum support threshold to reduce the number of patterns we need to process in the Search algorithm, as this was the main computational bottleneck of this work.

Necessity of the MDL Principle

Lastly, whilst we did manage to find patterns that allow us to differentiate between various languages using this method, the question has to be asked whether there might have been a simpler way to go about it, and whether the MDL principle was even needed. In hindsight, the 5000 or so patterns present in the 6 languages analysed might not have been too difficult to analyse in its entirety, without the need of reducing the pattern set. However, this may be a different story if many more languages were to be used.

7.2 Limitations and Future Work

There are several key limitations to keep in mind that we had to juggle throughout this thesis. However, there are two primary ones that are important to consider.

Firstly, computation power was a massive limitation on what we could do. We already discussed the 6 hour time limit for the Search algorithm, which, while we found that to be suitable, ideally the Search and Cover algorithms could be made more efficient, and could be run for longer periods of time to potentially find more patterns.

Secondly comes analysis. Whilst we aimed to find differences and similarities in languages without the need of a linguist, we found that there is limited analysis that can be done without prior linguistic knowledge. The relations between languages can be seen by means of the visualisations provided in Chapter 6, however, analysis of the patterns themselves was not done, as we are computer scientists.

In the future, the Search and Cover algorithms could be overhauled to be more efficient, and we could pass the resulting patterns to a linguist for their own input on the results. Not to mention, it might be interesting to make some additions, such as using different languages, or perhaps even non-parallel corpora as well as parallel corpora to see whether that has an effect on the results or not.

7.3 Conclusion

As the aim of this thesis was to find a way to automatically differentiate between various languages, it can be said that this was accomplished with some caveats. We successfully mined the Universal Dependency treebanks for patterns using gSpan, built a model and selected the patterns using the MDL principle and performed some experimentation and analysis on the resulting best model. Within this best model, we ultimately did find ways of analysing the patterns and found interesting observations regarding the similarities and differences between languages. However, we were limited by the fact that analysis on the individual patterns is difficult due to a lack of linguistic knowledge.

Bibliography

[1] G. Botha, V. Zimu, and E. Barnard, "Text-based language identification for south african languages," *SAIEE Africa Research Journal*, vol. 98, no. 4, pp. 141–146, 2021.

[2] T. Ölvecký, "N-gram based statistics aimed at language identification," *IIT. SRC. Bratislava*, pp. 1–17, 2005.

[3] T. Vatanen, J. J. Väyrynen, and S. Virpioja, "Language identification of short text segments with n-gram models.," in *LREC*, 2010.

[4] N. Zhong, Y. Li, and S.-T. Wu, "Effective pattern discovery for text mining," *IEEE transactions on knowledge and data engineering*, vol. 24, no. 1, pp. 30–44, 2010.

[5] A. C. Mendes and C. Antunes, "Pattern mining with natural language processing: An exploratory approach," in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pp. 266–279, Springer, 2009.

[6] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pp. 721–724, IEEE, 2002.

[7] S. Nijssen and J. N. Kok, "The gaston tool for frequent subgraph mining," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 1, pp. 77–87, 2005.

[8] H. Yao and H. J. Hamilton, "Mining itemset utilities from transaction databases," *Data & Knowledge Engineering*, vol. 59, no. 3, pp. 603–626, 2006.

[9] J. S. Park, M.-S. Chen, and P. S. Yu, "Using a hash-based method with transaction trimming for mining association rules," *IEEE transactions on knowledge and data engineering*, vol. 9, no. 5, pp. 813–825, 2002.

[10] "Universal Dependencies." `https://universaldependencies.org`, 2025. Accessed: 2025-06-11.

[11] J. Nivre, M.-C. De Marneffe, F. Ginter, J. Hajič, C. D. Manning, S. Pyysalo, S. Schuster, F. Tyers, and D. Zeman, "Universal dependencies v2: An evergrowing multilingual treebank collection," *arXiv preprint arXiv:2004.10643*, 2020.

[12] P. D. Grünwald, *The minimum description length principle*. MIT press, 2007.

[13] T. C. Lee, "An introduction to coding theory and the two-part minimum description length principle," *International statistical review*, vol. 69, no. 2, pp. 169–183, 2001.

[14] J. Vreeken, M. Van Leeuwen, and A. Siebes, "KRIMP: mining itemsets that compress," *Data Mining and Knowledge Discovery*, vol. 23, pp. 169–214, 2011.

[15] K. Smets and J. Vreeken, "SLIM: Directly mining descriptive patterns," in *Proceedings of the 2012 SIAM international conference on data mining*, pp. 236–247, SIAM, 2012.

[16] F. Bariatti, P. Cellier, and S. Ferré, "GraphMDL: Graph pattern selection based on minimum description length," in *Advances in Intelligent Data Analysis XVIII: 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27–29, 2020, Proceedings 18*, pp. 54–66, Springer, 2020.

[17] F. Bariatti, P. Cellier, and S. Ferré, "GraphMDL+ interleaving the generation and mdl-based selection of graph patterns," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 355–363, 2021.

[18] F. Bariatti, P. Cellier, and S. Ferré, "KG-MDL: Mining graph patterns in knowledge graphs with the mdl principle," *arXiv preprint arXiv:2309.12908*, 2023.

[19] K. Budhathoki and J. Vreeken, "The difference and the norm—characterising similarities and differences between databases," in *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2015, Porto, Portugal, September 7-11, 2015, Proceedings, Part II 15*, pp. 206–223, Springer, 2015.

[20] P. Elias, "Universal codeword sets and representations of the integers," *IEEE transactions on information theory*, vol. 21, no. 2, pp. 194–203, 2003.

[21] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *The Knowledge Engineering Review*, vol. 28, no. 1, pp. 75–105, 2013.

[22] C. Antunes, "Pattern mining over nominal event sequences using constraint relaxations," *Unpublished doctoral dissertation, Instituto Superior Técnico, Lisboa*, 2005.

[23] S. Tatikonda, S. Parthasarathy, and T. Kurc, "TRIPS and TIDES: new algorithms for tree mining," in *Proceedings of the 15th ACM international conference on Information and knowledge management*, pp. 455–464, 2006.

[24] Yu, Tian, "gSpan-mining: A Python implementation of gSpan." `https://pypi.org/project/gspan-mining/`, 2020. Accessed: 2025-06-17.

Appendix A

Universal Dependencies Edge Labels

This table is a comprehensive list of all the possible edge labels found within Universal Dependencies [10, 11].

Table A.1: UD Edge Labels and Their Syntactic Functions

Edge Label	**Syntactic Relation**
acl	clausal modifier of a noun
acl:relcl	relative clause modifier
advcl	adverbial clause modifier
advmod	adverbial modifier
advmod:emph	emphasising word, intensifier
advmod:lmod	locative adverbial modifier
amod	adjectival modifier
appos	appositional modifier
aux	auxiliary
aux:pass	passive auxiliary
case	case marking
cc	coordinating conjunction
cc:preconj	preconjunction
ccomp	clausal complement
clf	classifier
compound	compound
compound:lvc	light verb construction
compound:prt	phrasal verb particle
compound:redup	reduplicated compounds
compound:svc	serial verb compounds
conj	conjunction
cop	copula
csubj	clausal subject
csubj:outer	outer clause clausal subject

Edge Label	Syntactic Relation
csubj:pass	clausal passive subject
dep	unspecified dependency
det	determiner
det:numgov	pronominal quantifier governing the case of a noun
det:nummod	pronominal quantifier agreeing in case with the noun
det:poss	possessive determiner
discourse	discourse element
dislocated	dislocated elements
expl	expletive
expl:impers	impersonal expletive
expl:pass	reflexive pronoun used in reflexive passive
expl:pv	reflexive clitic with an inherently reflexive verb
fixed	fixed multiword expression
flat	flat expression
flat:foreign	foreign words
flat:name	names
goeswith	goes with
iobj	indirect object
list	list
marker	marker
nmod	nominal modifier
nmod:poss	possessive nominal modifier
nmod:tmod	temporal modifier
nsubj	nominal subject
nsubj:outer	outer clause nominal subject
nsubj:pass	passive nominal subject
nummod	numeric modifier
nummod:gov	numeric modifier governing the case of a noun
obj	object
obl	oblique nominal
obl:agent	oblique agent in passive construction
obl:arg	oblique argument
obl:lmod	locative modifier
oblq:tmod	temporal modifier
orphan	orphan
paratixis	parataxis
punct	punctuation
reparandum	overridden disfluency
root	root

Edge Label	Syntactic Relation
vocative	vocative
xcomp	open clausal complement