State Splitting for Interpreted Regular Tree Grammars

by

Johannes Gontrum

in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computational Linguistics

Department of Linguistics
University of Potsdam

August 18, 2015

Thesis supervisors:

Prof. Dr. Alexander Koller

Dr. Christoph Teichmann

Abstract

State Splitting for Interpreted Regular Tree Grammars

Johannes Gontrum

State splitting is a grammar improvement technique that replaces symbols by multiple new ones. In the process, various new rules are created that cause the grammar to perform more accurately. In this work I investigate previous approaches to state splitting which perform on probabilistic context-free grammars (PCFG). While most of them require a certain treebank or the researchers to select certain information by hand, I concentrate on a completely automatic approach introduced by Petrov et al. (2006). My goal is to apply this algorithm to interpreted regular tree grammars (IRTG). IRTGs can represent various different algebras, while algorithms perform on an underlying regular tree grammar. Hence, my implementation can be used to perform state splitting on grammar formalisms that are more expressive than PCFGs, like linear context-free rewriting systems (if their algebra is defined for an IRTG).

I examine the performance of my implementation and show that it works well on smaller grammars, but has shortcomings on treebank sized data due to data-type imprecision in Java.

Zusammenfassung

State splitting ist ein Verfahren zur Verbesserung von Grammatiken, in dem ein Symbol durch beliebig viele neue ersetzt wird. Dabei einsteht eine Vielzahl an neuen Regeln, die es der Grammatik erlauben, syntaktische Konstruktionen mit spezieller zu beschreiben.

In meiner Bachelorarbeit stelle ich zunächst verschiedene manuelle Ansätze vor, die jedoch eine Baumbank mit bestimmten Annotationen voraussetzen und nicht eigenständig entscheiden können, ob ein Symbol aufgeteilt werden soll oder nicht. Ich konzentriere mich hingegen auf ein Verfahren von Petrov et al. (2006), das unabhängig von Sprache und Domäne *State splitting* durchführt und eigenständig vielversprechende Symbole auswählt. Jeder dieser Ansätze bezieht sich allerdings auf probabilistische kontext-freie Grammatiken (PCFG). Daher stelle ich ein Verfahren für *State splitting* auf interpretierten regulären Baumgrammatiken (IRTG) vor. Dieser Grammatikformalismus erlaubt es, verschiedene Algebren zu repräsentieren, während unabhängig von ihnen Algorithmen auf eine zugrundeliegende reguläre Baumgrammatik angewandt werden. Daher ermöglicht meine Implementierung das Durchführen von *State splitting* für Gramamtikformalismen, die expressiver sind als PCFGs, solange ihre Algebren definiert sind.

Die Anwendung meiner Implementierung auf kleinere IRTGs zeigt ihre Richtigkeit. Es kann jedoch zu Schwierigkeiten bei sehr großen Daten kommen, da die dabei entstehenden sehr kleine Wahrscheinlichkeiten von Java abgerundet werden.

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for another degree at this or any other university.

_____

(Signature)

Berlin, August 18, 2015

Contents

1 Introduction **1**

2 Introduction to State Splitting **2**
2.1 Linguistically Motivated . . . 2
2.2 Automatic Approaches . . . 3
2.3 Splitting and Merging . . . 5
2.4 Results and Evaluation . . . 6
2.4.1 Precision, Recall and F-Score . . . 7
2.4.2 Results . . . 7

3 Interpreted Regular Tree Grammars **8**
3.1 Introduction . . . 8
3.2 Formal Definitions . . . 8
3.2.1 Regular Tree Grammar . . . 8
3.2.2 Algebra . . . 9
3.2.3 Tree Homomorphism . . . 9
3.2.4 Interpretation . . . 9
3.2.5 Example: Interpreting a Derivation Tree . . . 10
3.3 Parsing . . . 10
3.3.1 Decomposition . . . 10
3.3.2 Inverse Homomorphism . . . 11
3.3.3 Intersection . . . 11

4 The Expectation-Maximization Algorithm **12**
4.1 Inside and Outside Probabilities for PCFGs . . . 12
4.1.1 Inside Probabilities . . . 12
4.1.2 Outside Probabilities . . . 12
4.2 An Abstract View on Spans and Trees . . . 13
4.2.1 Tree Automata . . . 13
4.2.2 Spans . . . 14
4.3 Inside and Outside Probabilities for Tree Automata . . . 15
4.3.1 Inside Probabilities . . . 15
4.3.2 Outside Probabilities . . . 15
4.4 Expectation-Maximization . . . 15
4.4.1 PCFGs . . . 15
4.4.2 Tree Automata . . . 16

5 State Splitting for IRTGs **18**
5.1 Splitting a Grammar . . . 18
5.1.1 Splitting . . . 18
5.1.2 Adding Randomness . . . 18
5.1.3 Renormalization . . . 18
5.1.4 Tree Homomorphisms . . . 19
5.2 Identifying Important States . . . 19
5.2.1 Recovering Inside and Outside Scores . . . 20

5.2.2 Estimating the Loss in Likelihood . . . 21
5.3 Merging . . . 21

6 Results and Discussion **23**
6.1 Experimental Setup . . . 23
6.1.1 Data . . . 23
6.1.2 Tree Transformation . . . 23
6.1.3 Binarization . . . 24
6.2 Differences to Petrov et al. (2006) . . . 24
6.3 Analysis: Penn Treebank Grammar . . . 25
6.4 Analysis: Small Grammar . . . 26
6.5 Discussion . . . 27
6.6 Future Work . . . 28

7 Conclusion **29**

List of Figures

2.1 An exemplary unmodified tree taken from the Penn Treebank (Marcus et al., 1993). . . . 2
2.2 Rules of a context-free grammar that are read off the tree in figure 2.1. 2
2.3 The phrase "to see if advertising works" first wrongly parsed with a grammar with only parent annotation. The second tree shows the correct result, if additional annotations for the preterminal symbols like "Det" or "N" were used. 3
2.4 Parent annotation is used to weaken the independence assumption of a context-free grammar: The nonterminal symbols in the left tree are annotated with their lexical head noun, while the second tree uses its category. 4
2.5 Demonstration of state splitting for a binary rule, where $X = \{x_1, x_2\}, Y = \{y_1, y_2\}, Z = \{z_1, z_2\}$. The weight of the rule is evenly split over the newly created ones. Additional steps add randomness and redistribute the rule weights using an EM algorithm. 5
2.6 Illustration of two split-merge cycles, resulting in three new symbols instead of four. . . 6
2.7 Calculation of precision, recall and the F_1-score, assuming that P is a set of constituents in the parse and G the constituents in the gold tree. 7

3.1 Example of the derivation and interpretation process in IRTGs. (a) is a RTG, (b) a derivation tree of (a), (c) shows the mappings of a tree homomorphism h for strings and (d) the tree (b) applied to h. 10
3.2 Exemplary decomposition tree of the string "the squirrel meets the fox" with the used nonterminals next to the symbols of Δ. 11
3.3 Inverse homomorphism applied to the tree in Figure 3.2. 11

4.1 Illustration of the calculation of the outside score of J that covers the span (p,q). 13
4.2 The structure of a parse chart. (a) is the original RTG with the mapping of the homomorphism of the string algebra in (b). (c) shows the RTG of the decomposition language of a string input of length 3 and (d) shows a derivation tree of the parse chart with pairs of the RTG in (a) and (c). 14

5.1 Demonstration of state splitting for an exemplary RTG rule. If all symbols are split, a rule with an arity of 3 is split into 16 new rules. 19
5.2 Merging rules for the same nonterminal on the left-hand side. 21
5.3 Merging rules for the same right-hand side. The rules for each split symbol is multiplied by its estimated frequency. 22

6.1 The hierarchy of the splitting of the symbols "ADVP" and "PRP". 24
6.2 The increasing number of rules (blue) on the left axis and added states (red) on the right. 26
6.3 Parse time for test sentences in minutes (blue) on the left and the f-score (green) right. . . 26

List of Tables

2.1 Comparison of the best results reached by the presented papers and the used sections of the Penn Treebank. 7

6.1 Analysis of different tree transformations. Parsing was performed on the binarized grammar, distributed on 20 CPUs. The f-score calculated for the first 100 sentences in Penn Treebank section 22 with length ≤ 20. 23
6.2 Number of latent annotations after six split-merge cycles. 25
6.3 Creating and evaluating an IRTG with state splitting. Time in minutes, parts of the calculations were performed in multithreading on 20 cores. Parsing was performed on the sentences of the section 23 of the Penn Treebank with 20 words or less. 27

Introduction

It is fascinating that it takes our brains only fractions of a second to understand and validate a sentence. But while parsing seems to be such an easy process for us, it is still a difficult task for computers. Even after more than sixty years of research with better and better algorithms, new methods to improve speed and accuracy of natural-language parsers are still discovered.

Since Kasami (1965) and Earley (1970) presented fundamental algorithms, the rising speed and the decreasing cost of computers in the early 1990s set the presuppositions for statistical parsing. Like in a race, teams of researchers competed against each other in finding better parsing algorithms. Soon it became clear that even rising computer power does not solve this problem completely. Hence, from the mid of the 1990s on, researches tried to optimize the grammars that provided the underlying rules for parsing sentences. They started to realize that grammars read from human annotated treebanks are at some points not accurate enough to cover all linguistic phenomena and therefore tried to improve them to reveal these hidden annotations.

An example for these insufficient annotations is the symbol "NP" that describes all nominal phrases. However, linguistics teach us that there are many different kinds of nominal phrases like a subject or object phrase. Instead of using one symbol that tries to cover both cases, it is more suitable to use one distinct symbol for all subjects and one for all objects. This process is called state splitting. The first attempts in this direction relied on tree annotations and required the researches to select symbols for the splitting process based on their linguistic knowledge. Around 2005 however, a new technique was introduced that automatically chooses these symbols. This breaking new idea allowed the application on any treebanks, since it is not constrained by a specific annotation scheme. Most interestingly, the symbols created in the splitting process can be interpreted linguistically, showing that hidden linguistic structures have been revealed indeed.

In my bachelors thesis I will first present the history of discovering these hidden annotations and subsequently examine the algorithm for automatic state splitting. My main goal is to abstract from the implementation for context-free grammar and to apply the techniques to the more general grammar formalism *interpreted regular tree grammars* by Koller and Kuhlmann (2011). The advantage of IRTGs is their ability to represent a wide variety of algebras, from strings and trees to linear context-free rewriting systems and even semantic representations. Multiple different algebras can then be combined in a single IRTG to be used as a synchronous grammar or as a transducer that translates between them. For this purpose an underlying regular tree grammar is used on which algorithms are performed independently of the algebras.

After defining IRTGs, I will take a closer look at the expectation-maximization algorithm that is needed to reveal the hidden information in a grammar and explain how it can be defined to work on IRTGs. Eventually I will introduce in detail how automatic state splitting can be performed on IRTGs and therefore improve grammars for all algebras. My work is integrated in *Alto*, the official implementation of IRTGs.

Introduction to State Splitting

2.1 Linguistically Motivated

In the beginning of the 1990s, the decreasing cost for higher computer power opened up new possibilities for researchers. Tasks like statistical parsing that could not be applied on a bigger scale before now became possible. In that time several research teams competed in building faster and more accurate parsers than the other teams. The Penn Treebank (Marcus et al., 1993) plays to this day an important role in standardizing the testing of parsers. It contains 24 sections of text taken from the Wall Street Journal that has been syntactically annotated. These syntactic trees are the foundation of learning rules for phrase structure grammars and evaluating the output of a parser to human-made annotations.

Apart from developing new heuristics and sophisticated data structures for the parsing algorithm itself, researchers also focused on refining the grammar. This was a logical step, since at some point the improvements in the parsers themselves were only marginal. One of the first approaches in this direction was made by Charniak (1996) who discovered that some of the annotations in the Penn Treebank were not suitable for grammars. As an example, in Figure 2.1 the "-NONE-" category was used to display traces of movements within a tree, but it does not carry beneficial information for a grammar. A rule "NP → QP -NONE-" would never be used, because the symbol "-NONE-" does not describe any visible part of the sentence.

While Charniak (1996) focused on removing distracting annotations that would worsen a grammar, Johnson (1998) took an approach from a different perspective: He added new annotations to enrich the information within the trees. Most syntactic parsers work with context-free grammars and they work fine to describe most common human languages. Conveniently, CFGs come with a parsing complexity of n^3 (Kasami (1965); Younger (1967); Cocke (1970)) in respect to the length of the input sentence. But as their name suggests, CFG rules are utterly unaware of the context they are used in. Take Figure 2.1 for example: You can read the grammar of the tree and find three possible ways to construct a NP (Figure 2.2). The NP leading to DT and NN can occur in the same places as one that describes QP and -NONE-. From another perspective, if I write the rule "S → NP VP", all NPs could be the first child, even if in this context I would prefer a NP that covers the subject of a sentence.

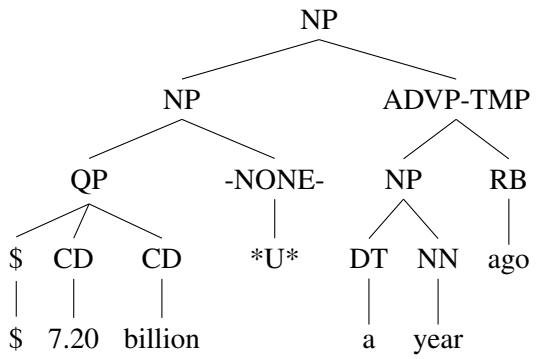

Figure 2.1: An exemplary unmodified tree taken from the Penn Treebank (Marcus et al., 1993).

NP → NP ADVP-TMP
NP → QP -NONE-
QP → $ CD CD
ADVP-TMP → NP RB
NP → DT NN
$ → $
CD → 7.20 | billion
-NONE- → *U*
DT → a
NN → year
RB → ago

Figure 2.2: Rules of a context-free grammar that are read off the tree in figure 2.1.

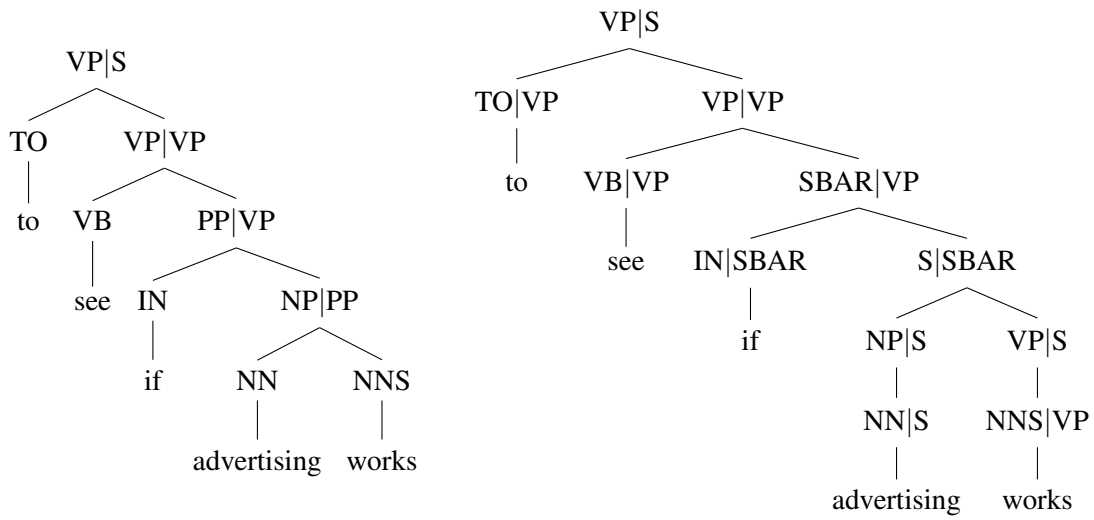

Figure 2.3: The phrase "to see if advertising works" first wrongly parsed with a grammar with only parent annotation. The second tree shows the correct result, if additional annotations for the preterminal symbols like "Det" or "N" were used.

To weaken the independence assumptions of a CFG, Johnson (1998) suggests to add information about their parent to the node label. This way, rules become more aware of their position within a tree. The left tree in Figure 2.3 shows a tree with parent annotation, where the parent is attached to the right of each node. A disadvantage is that the amount of rules in the grammar increases dramatically, so he decided to change only VPs and NPs. Another way to regard this approach is the splitting of the symbol "NP" into several other symbols, "NP|S" and "NP|VP" in this case.

A few years later, Klein and Manning (2003) take on the idea of splitting symbols based on linguistic motivation. Their main idea is to prove that given some simple tree transformations, an unlexicalized grammar can perform as well as end-1990 state of the art parsers with lexicalized grammars. Unlexicalized grammars do not contain any words, but use their part-of-speech tags as the alphabet. This method makes the resulting trees less accurate, because they strip away information about the word. For example prepositions (of, in, from) and complementizers (that, for) are tagged with the same POS tag "IN". Linguistically speaking, one could argue that you expect other structures arise from a preposition than a complementizer - an information that we lose by purely relying on an unlexicalized grammar. On the other hand, POS driven grammars are much more general and are not running into a sparse-data problem if the input sentence contains an unseen word.[1] To walk the middle way between a very specialized lexical grammar and a too general unlexicalized one, they decided to split the states for certain POS tags by adding the parent state to the label. They discovered that splitting IN six times results in a notable increase of f-score.

Even if Klein and Manning (2003) discovered that certain POS tags are too coarse by covering different categories of words, they still select these cases by hand based on their linguistic knowledge. Hence there might be other tags or states that should or should not be split by parent annotation and that are not processed by their method.

2.2 Automatic Approaches

In 2005, Matsuzaki et al. (2005) and Prescher (2005) simultaneously tried to approach grammar refinement from an automatic perspective. Both wanted to reveal *latent* or hidden information for the symbols. This can be explained as knowing that a nominal phrase is a symbol that in reality describes very distinct phrases. A NP can be used as a subject, an object in general or even a direct or indirect object. As one

[1] Sophisticated lexicalized parsers have other solutions to the sparse-data problem, so that a single unknown word would not render the whole sentence invalid.

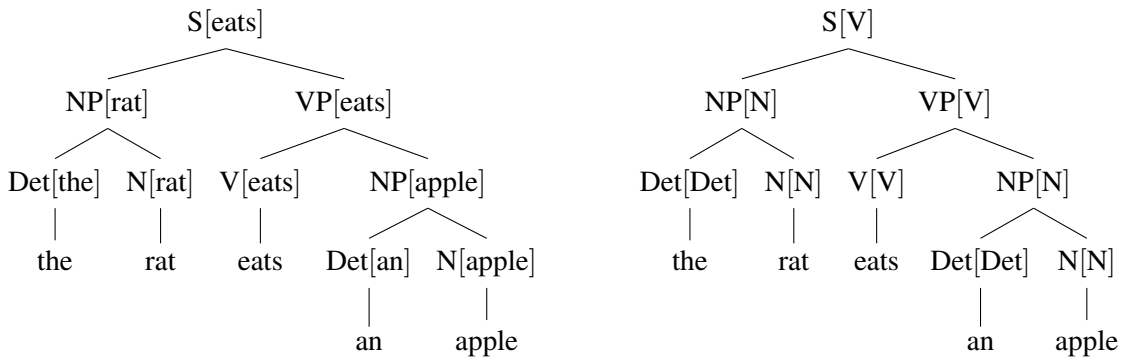

Figure 2.4: Parent annotation is used to weaken the independence assumption of a context-free grammar: The nonterminal symbols in the left tree are annotated with their lexical head noun, while the second tree uses its category.

can imagine, an accusative object is more likely to lead to certain NP rules while a subject might prefer a different set of rules. Hence, describing both with the same symbol leads to a generalization. Here, a latent information can be that the single symbol NP really consists of a variety of other symbols, each with different probabilities for the NP rules.

Matsuzaki et al. (2005) introduce a form of a PCFG with latent labels attached to each rule in the grammar. The latent symbols are described as $x \in \mathbb{I}$ where $\mathbb{I}$ is a set of latent symbols, for example $\mathbb{I} = \{x_1, x_2, x_3\}$ if one wants to consider three types of information. In the second step they use an expectation-maximization algorithm that they modified to work with their new grammar definition. EM is an unsupervised learning algorithm that maximizes the probability of the sentences in the corpus by redistributing the probability mass of the rules (Manning and Schütze, 1999). The EM algorithm is described in detail in Chapter 4. If you have the rule $\mathrm{A}[X] \rightarrow \mathrm{B\ C}$ where X stands for a latent information x_1, x_2, x_3, you actually have three rules that only differ in the left-hand side and that have almost the same weight. Performing EM training then redistributes the weight, so that for example the rule with x_1 has a higher weight than the other two. This is then interpreted as finding a latent information that prefers x_1 in a certain context.

They train their model with different amounts of latent symbols ranging from 1 up to 16 and discover that a higher number results in a higher f-score. Following their logic, this makes sense since the hidden information can be finer grained. Eventually they point out that an even higher performance would be possible with 32 or 64 symbols, but that current computers are not able to compute this task in acceptable time and that one needs to find a way to reduce the number of symbols while maintaining the same performance.

Prescher (2005) et al were working on a similar idea. They start by pointing out that manual grammar refinement like parent annotation works good on the Penn Treebank, but these transformations are not as sufficient on other treebanks, languages or even other domains. Therefore, they favor an automatic selection of states to split. These techniques would be independent from language and domain. They start by introducing a way to lexicalize a grammar by encoding information about the words in the nodes, once with the words themselves and again with POS tags (see Figure 2.4). This information flows bottom-up through the categories, forming more complex heads. While this is another technique of adding more concrete features and information, similar to Klein and Manning (2003), Prescher (2005) add a certain amount of hidden information to the heads and perform an EM algorithm. They discover as well that an increased amount of information results in much bigger grammars. As baseline they use the lexicalized grammar with 50k rules that naturally required no training time. A version with ten information types resulted in 400k rules and a training time of 7 days. Doubling the amount of latent information seems to double the number of rules in the grammar.

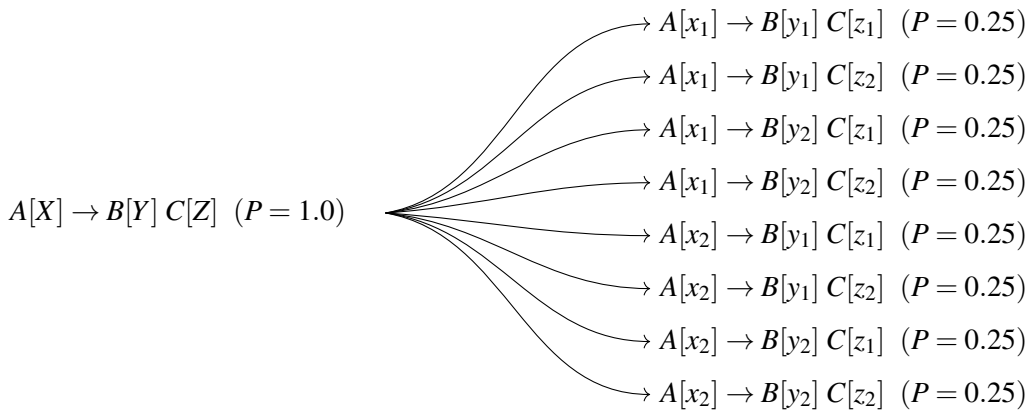

Figure 2.5: Demonstration of state splitting for a binary rule, where $X = \{x_1, x_2\}, Y = \{y_1, y_2\}, Z = \{z_1, z_2\}$. The weight of the rule is evenly split over the newly created ones. Additional steps add randomness and redistribute the rule weights using an EM algorithm.

2.3 Splitting and Merging

As pointed out before, automatically adding latent information to a grammar can help to improve it. Since this method is independent of the language or the domain it is applied to, this generalization is an important criterion when I apply these techniques to other grammar formalisms than PCFG. But the automatic splitting of symbols results in an abundance of new rules that slow down the parsing process. Also one can argue if all of them add the same level of usefulness. Klein and Manning (2003) demonstrate that splitting certain symbols can even lower the accuracy of the new grammar.

Petrov et al. (2006) therefore take on the idea of Matsuzaki et al. (2005) and Prescher (2005) and introduce a technique to merge certain states after splitting them, if they are not adding enough usefulness to the grammar. This method allows them to split symbols dynamically while retaining a relatively small grammar. With an f-score of 90.2%, they set a new record in parsing accuracy and integrated their algorithm into the Berkeley Parser[2]. Because the number of rules and symbols grow exponentially with the amount of splits, Petrov et al. (2006) decided to start with an unlexicalized grammar that was as minimal as possible. They read it from the section 2-21 of the Penn Treebank after removing unnecessary annotations from the trees. Eventually they transform the trees to follow the binarized X-bar form because the EM training performs faster on a binarized grammar. As they start their experiments, the initial grammar contains only 98 symbols, 236 unary and 3.240 binary rules. This minimal approach causes expected low values in precision (65.8) and recall (59.8). They point out that the removal of any annotation is fully intended and in contrast to Matsuzaki et al. (2005) they do not add parent annotation to reduce the search space for the EM algorithm. Their focus is concentrated on a fully automated system.

In the splitting process all symbols in the grammar are replaced by two new ones (see Figure 2.5) and subsequently EM training is performed to find the best probabilities for the new grammar. Note that splitting a symbol more than two times can be archived by multiple iterations of the algorithm. To make the EM algorithm perform correctly, they have to make sure that the probability of the newly created rules are different. If I split a binary rule in Figure 2.5, I create eight new rules, four for A^1 and four for A^2, similar to $A[x_1]$ or $A[x_2]$. Assuming that A^1 and A^2 produce only this single rule with the probability 1, all new rules would have $1/4$ as their weight. But how should the EM algorithm that highly relies on the rule probabilities start to decide which rule is more likely in a certain context if all rules have the same weight? To break the symmetry, Petrov et al. (2006) add 1% of randomness, meaning that they add a number between 0.01 and 0.00 to the rule weight and normalize afterwards.

Even if a symbol is only split into two new ones, the size of the grammar would reach the limits of what is computable after a few iterations. But multiple splits can also decrease the accuracy of the grammar. Petrov et al. (2006) explain this idea on the comma POS tag. In contrast to most other POS

[2] https://github.com/slavpetrov/berkeleyparser

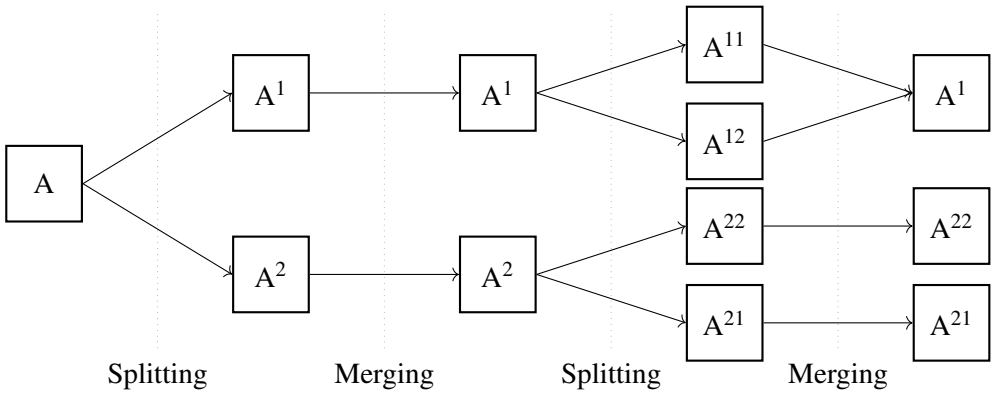

Figure 2.6: Illustration of two split-merge cycles, resulting in three new symbols instead of four.

tags, the comma tag only labels a single terminal symbol. Splitting this symbol would not only make no sense, but could also worsen the grammar because other rules using the POS tag would make false assumptions about the difference of the split symbol.

To dynamically decide whether to split a symbol or not, they take the following approach: First they split every symbol into two new ones and perform EM training to learn the probabilities for them. Then they decide for each symbol separately, if there would be great loss in likelihood, if this symbol would have not been split. If that is the case, they merge the symbols back together, reducing the size of the grammar, while still splitting important symbols. This procedure can then be repeated. After six cycles their initial grammar is only 17% of the size the grammar would normally have without merging. Take Figure 2.6 for example. Let us assume that A describes a nominal phrase. First, I split it into A^1 and A^2 and during the merging process I discover that it was the right decision to do the split. So in the next iteration, I split A^1 and A^2 again. After a new round of EM training, I note that I do not benefit from splitting A^1 into A^{11} and A^{12}, so I want to revert the split. A^2 on the other hand should still be split. A case like this is plausible for NPs: First I split subject NPs and object NPs. In the next iteration, I discover that subject NPs should not be split any further, while object NPs can still be differentiated to direct and indirect objects.That this example is not made up is demonstrated by the analysis that Petrov et al. (2006) provided. They find that the splits and the discovered latent annotations can be interpreted linguistically. This is especially interesting as it closes the circle to Klein and Manning (2003): They selected symbols to split by their linguistic knowledge, but have probably overseen some. This algorithm on the other hand does not have such knowledge but can reveal all hidden information that eventually can be interpreted by a human with background in linguistics.

The question how to decide whether the split of the symbol contributes any improvement is not trivial. Petrov et al. (2006) do not want to know the improvement that a symbol alone contributes, they are interested in the contexts that arise from splitting multiple symbols that may depend on one another. Therefore, the question is rather how the likelihood of the grammar would suffer if one symbol would have not been split. If this loss is not huge, the splitting was not necessary. This score can be computed from the inside and outside probabilities of a parse (see Chapter 4), but performing the time consuming EM training for all possible grammars with removed symbols is reasonable.

Hence, they suggest a way to estimate the loss by using the inside and outside scores based on the completely split grammar. In short, they recover the inside and outside scores for the original states based on the relative probabilities of the split states. Given these scores, they can calculate the loss for each position in the parse tree by comparing the old likelihood to the one of the split symbols. This procedure is explained in detail in Chapter 4, where I will explain the exact implementation.

2.4 Results and Evaluation

The Penn Treebank (Marcus et al., 1993) is used to train and evaluate the described methods. Usually, sections 2-21 are used for the training, while the sections 1, 2 or 22 serve as a development corpus to

$$LP = \frac{|\,P \cap G\,|}{|\,P\,|} \qquad LR = \frac{|\,P \cap G\,|}{|\,G\,|} \qquad F_1 = \frac{2 \times LP \times LR}{LP + LR}$$

Labeled precision | Labeled recall | F_1

Figure 2.7: Calculation of precision, recall and the F_1-score, assuming that P is a set of constituents in the parse and G the constituents in the gold tree.

fine tune parameters. The evaluation itself is performed by parsing the sentences in section 23 and then comparing the calculated trees to the hand annotated trees in the section. Depending on the paper, some researchers ignore sentences over a certain length, usually sentences with 40 words or more. This trend was necessary for slow computers in the 1990s, but is fading what makes it difficult to exactly compare the results.

2.4.1 Precision, Recall and F-Score

Evaluating the correctness of the output of a parser is usually done by the PARSEVAL algorithm (Abney et al., 1991). Because comparing the parse tree and the hand annotated gold tree would normally result in a binary output where the parse tree is either exactly equal to the gold tree or it is not. To evaluate a parse tree more in a more sophisticated way, Abney et al. (1991) introduced the concept of labeled precision (LP) and labeled recall (LR). Both assume that a tree can be represented as a set of constituents that are tuples of the label of a node and the indexes of the words it covers. The precision is the amount of constituents of the parse tree that appear in the gold tree. On the other hand, the recall measures the amount of constituents in the gold tree that are used in the parse tree. The f-score then combines both measurements. The most commonly used implementation for these scores is EVALB by Sekine and Collins (1997).

2.4.2 Results

Even if not all researchers commit to the convention of using section 2-21 for training and section 23 for evaluation, the comparison in Table 2.1 shows the increase of f-score over the past 15 years. There is an increase from the early work of Johnson (1998) to the experimentations with unlexicalized grammar by Klein and Manning (2003). But this is mostly due to the fact that other improvements have been made in the meantime. As Klein and Manning (2003) point out, their score with unlexicalized grammars is slightly lower than the best parsing performance at their time. However, most interesting for this thesis is the difference between the work of Matsuzaki et al. (2005) or Prescher (2005) and Petrov et al. (2006), who introduced the merging process.

Publication	**New technique**	**Trained on**	**Evaluated on**	**LP**	**LR**	**F_1**
Johnson (1998)	Parent annotation	2-21	22	80.0	79.2	79.6
Klein and Manning (2003)	POS annotation	2-21	23	86.9	85.7	86.3
Matsuzaki et al. (2005)	Latent annotation	2-20	23	86.6	86.7	86.7
Prescher (2005)	Latent annotation	2-21	22	85.2	85.0	85.1
Petrov et al. (2006)	Merging	2-21	23	**90.3**	**90.0**	**90.1**

Table 2.1: Comparison of the best results reached by the presented papers and the used sections of the Penn Treebank.

Interpreted Regular Tree Grammars

3.1 Introduction

Interpreted regular tree grammars (Koller and Kuhlmann, 2011) can represent a wide variety of algebras. Tree-adjoining grammars (TAG), linear context-free rewriting systems (LCFRS), strings, trees and even graphs for semantic representation are a few examples that are supported by the official implementation *Alto*. IRTGs can easily be used as synchronous grammars, where an input for multiple interpretations is parsed or generated simultaneously. Another valuable feature of IRTGs is to use them as a transducer to translate an object of one interpretation to one of any other interpretation. Interpretations are defined as an algebra and a homomorphism to parse an object of the algebra to an underlying derivation tree and the other way around by interpreting a derivation tree to an object of the algebra. Due to the limitations of this thesis, I will only briefly introduce important definitions and concepts that are crucial for the understanding of IRTGs, but I highly suggest the publication by Koller and Kuhlmann for a deeper understanding of the topic.

IRTGs are discriminating between derivation and interpretation. Koller and Kuhlmann use the derivation process of context-free grammars as an example. Usually this is seen as a string rewriting process where the right hand sides of the applied rules are replaced by their left hand side: "Peter likes Mary $\Rightarrow$ N likes Mary $\Rightarrow$ N V Mary $\Rightarrow$ N V N $\Rightarrow$ N VP $\Rightarrow$ NP VP $\Rightarrow$ S". IRTGs on the other hand first create a derivation tree that shows how the rules are applied and in a second step interpret this tree.

These derivation trees are described by the underlying regular tree grammar $\mathcal{G}$, while the various interpretations are used to interpret the derivation tree over their respective algebra. If an IRTG is used as a transducer, the derivation tree is created for an input of any interpretation and then interpreted by another interpretation. An example to demonstrate this is semantic parsing. Here the IRTG has two interpretations: one with a string algebra and one with a s-graph algebra (Koller, 2015). Now one can create the derivation tree for a string and then read off the semantic structure for it by interpreting the tree over a s-graph algebra.

Using a RTG as the core comes with another benefit: Most algorithms, like for example expectation-maximization, only have to be implemented on the RTG, completely independent of the used interpretation. To gain access to the optimized and well tested algorithms for a new algebra, one is only required to implement the algebra itself, but not the various algorithms.

3.2 Formal Definitions

Koller and Kuhlmann (2011) define an IRTG $\mathbb{G} = (\mathcal{G}, I_1, ..., I_n)$ as a tuple consisting of its RTG $\mathcal{G}$ and n interpretations $I_1, ..., I_n$ that describe how a derivation tree can be created or evaluated.

3.2.1 Regular Tree Grammar

While normal context-free grammars generate strings, regular tree grammars describe a language of trees. The symbols at the nodes of the tree are assigned a *rank* that state how many subtrees are branching from the symbol. Formally, these symbols together with their rank, are defined in the set Σ, a so called "signature". RTGs also need a set of nonterminal symbols N, including a start symbol S and production rules in the set P. Formally, a RTG is defined as $\mathcal{G} = (N, \Sigma, P, S)$. One way of understanding rules for context-free string grammars is that a nonterminal symbol produces a string that contains placeholders for

other strings, which are subsequently generated by nonterminal symbols on the right hand side of the rule. For tree grammars on the other side, a nonterminal symbol creates a new tree, where each node contains a symbol out of Σ. Like string grammars, the nonterminal symbols on the right hand side of the rule are wildcards for the trees that they will generate. The following rule for example creates a tree with the symbol $r/3$[1] at its node and three subtrees, created by the nonterminal symbols X,Y,Z: $A \rightarrow r(X,Y,Z)$. Throughout this thesis I will generalize RTG rules in the form $A \rightarrow f(A_1,..,A_n)$ for a rule with the arity n and the symbol $f/n \in \Sigma$. If not mentioned otherwise, this includes terminal rules $A \rightarrow g$, where $g/0$ is a symbol of rank 0 (Comon et al., 2007). The underlying RTG of an interpreted regular tree grammar will be referred to as $\mathcal{G}$.

3.2.2 Algebra

Informally speaking, IRTGs use algebras to describe how other grammar formalisms and structures work and what they are defined on. If I, for example, want to use an IRTG to represent a certain context-free grammar, I have to specify that I work with strings and what *operations* I know to combine two strings into one. The latter is rather simple, as it is only the concatenation operation $a \bullet b = ab$. Secondly, I have to define the *domain* of the algebra as a set of all terminal symbols in the original CFG and the $\bullet$-symbol of the concatenation-function (Koller and Kuhlmann, 2011).

In detail, algebras also depend on a signature Δ and need a function for each symbol in it. Let $f/n \in \Delta$ and A be the domain: $f^{\mathcal{A}} : A^n \rightarrow A$. These functions are necessary to *evaluate* a term t to an object of the algebra $\mathcal{A}$: $[\![t]\!]_{\mathcal{A}}$. A *term* is a tree over the symbols in Δ: T_Δ. Imagine the string algebra again: The signature Δ consists of the symbol $\bullet/2$ and the terminal symbols of the CFG, all with rank 0. Therefore, the tree of strings $\bullet$(the, $\bullet$(squirrel, sleeps)) is a string over Δ. To evaluate it, I apply the binary concatenation function "$\bullet$" to join the leaves "'squirrel' and "sleeps" and subsequently "the" and "squirrel sleeps", resulting in "the squirrel sleeps", an object of the algebra. In summary, algebras are used to transform a tree over symbols of their signature to an object of the algebra. Logically, this process can be reversed by decomposing an object of an algebra to a term.

3.2.3 Tree Homomorphism

Keep in mind that the evaluation of an algebra is performed on its signature Δ which is not necessarily equal to the signature Σ of the RTG $\mathcal{G}$. To close the missing link, I will introduce tree homomorphisms that transform trees over Σ to trees over Δ. Formally, a tree homomorphism is a total function $h : T_\Sigma \rightarrow T_\Delta$. I specify a tree over Δ for every tree over Σ that I have defined on the right-hand side of a rule in $\mathcal{G}$. More specifically, tree homomorphisms can be described as a set of pairs $\langle f, h(f) \rangle$ with $f \in \Sigma$ and $h(f) \in T\Delta \cup x_1,...,x_n$ (see Koller and Kuhlmann (2011) and Comon et al. (2007)). The picture of the function consists therefore of the symbols in the target algebra Δ and n variables $x_1,...,x_n$ for a symbol f/n. These variables will be replaced by the subordinate trees over Δ as soon as they are calculated. As an example, let $t_1 = f(X,Y)$ be a tree over Σ and $t_2 = NP(x_1,x_2)$ a tree over Δ with $h(t_1) = t_2$. x_1 is then substituted by the picture of the subtree of X and x_2 of that of Y.

3.2.4 Interpretation

Now that I have explained the required definitions, I can introduce the exact concept of an interpretation. Let an interpretation I be a tuple of an algebra $\mathcal{A}$ and a tree homomorphism h that uses the signature of $\mathcal{A}$ as its target signature. With algebra and homomorphism combined, one can now define for example a string algebra for German, a string algebra for English or use any other algebra together in the same IRTG.

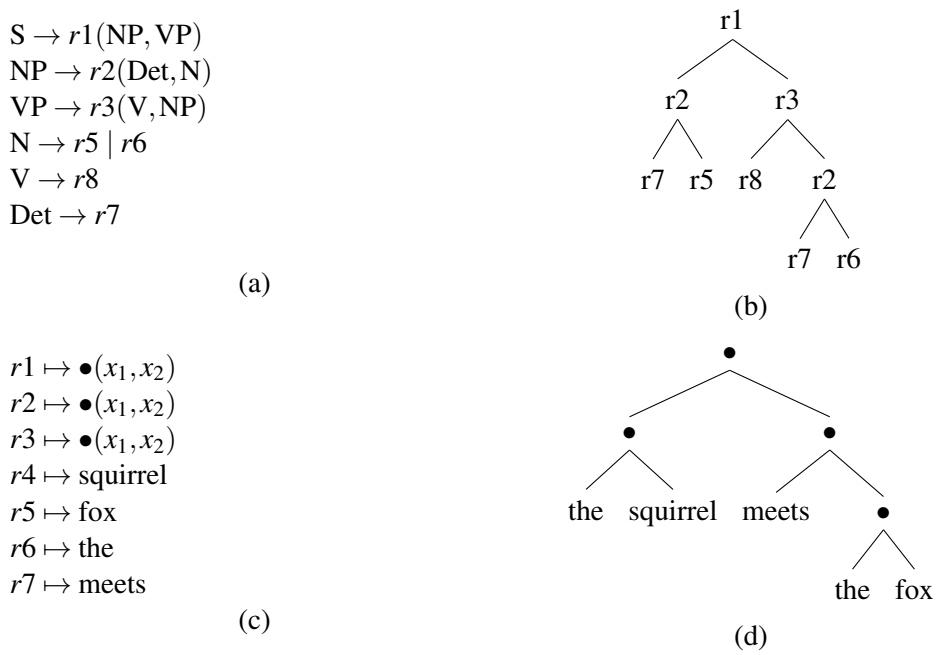

Figure 3.1: Example of the derivation and interpretation process in IRTGs. (a) is a RTG, (b) a derivation tree of (a), (c) shows the mappings of a tree homomorphism h for strings and (d) the tree (b) applied to h.

3.2.5 Example: Interpreting a Derivation Tree

To put the definitions into a context, I will describe how a derivation tree is interpreted by a string interpretation. Figure 3.1 (a) describes the production rules of the RTG $\mathcal{G}$ with the start symbol S, $N = \{\text{S, NP, VP, N, V, Det}\}$ and $\Sigma = \{r1/2, r2/2, r3/2, r4/0, r5/0, r6/0, r7/0\}$. This is sufficient for the underlying grammar. Now I define a string interpretation for it: The algebra $\mathcal{A}$ uses the concatenation operation described above and has the signature $\Delta = \{\bullet/2, \text{the}/0, \text{squirrel}/0, \text{meets}/0, \text{meets}/0, \text{fox}/0\}$. The homomorphism h maps trees over Σ to trees over Δ, as described in Figure 3.1 (c). The used IRTG is now defined as $\mathbb{G} = \langle \mathcal{G}, \langle \mathcal{A}, h \rangle \rangle$.

Now I let $\mathcal{G}$ generate a derivation tree that is shown in Figure 3.1 (b). To interpret this tree, I apply the homomorphism h to the tree in Figure 3.1 (b) and receive the tree over Δ in Figure 3.1 (d). The last step consists of using the operation function, linked to the symbol $\bullet/2$ of the algebra $\mathcal{A}$. As explained above, this will evaluate Figure 3.1 (d) to the string "the squirrel meets the fox".

3.3 Parsing

3.3.1 Decomposition

While the process from a derivation tree to an algebra object has become clear by now, I will explain in the following how an algebra object is parsed to a derivation tree. For the sake of simplicity, I assume an IRTG with only one interpretation: $\mathbb{G} = \langle \mathcal{G}, \langle \mathcal{A}, h \rangle \rangle$. To parse an algebra object a, I first have to compute all possible terms over $\mathcal{A}$. In other words, I search all terms over $\mathcal{A}$ (trees over the algebra signature Δ) that evaluate to a. To continue the example of the string algebra, this means finding all the ways to concatenate the words in a. If a is again "the squirrel meets the fox", the trees would be $\bullet(the, \bullet(squirrel, \bullet(meets, \bullet(the, fox))))$, $\bullet(\bullet(the, squirrel), \bullet(meets, \bullet(the, fox)))$ and many more. This may seem like an unnecessary overhead at first, but keep in mind that this algorithm has to work on any algebra and most of them are more complex than the one for strings.

Since $terms_{\mathcal{A}}(a)$ can be an infinitely large set, I describe the *decomposition* language with a regular tree grammar $D(a)$. The states of this grammar mark exact position in the terms, so that each is used only

[1] The notation "$/n$" describes the rank of the symbol.

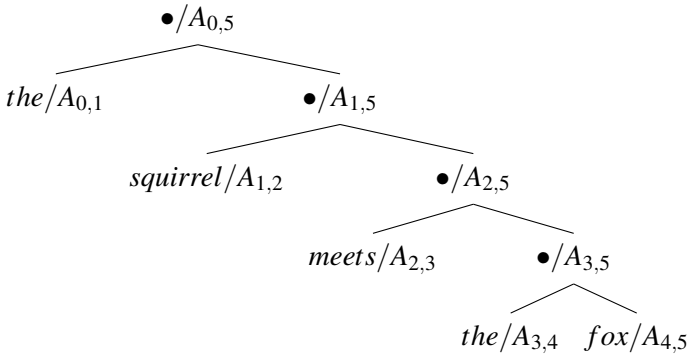

Figure 3.2: Exemplary decomposition tree of the string "the squirrel meets the fox" with the used nonterminals next to the symbols of Δ.

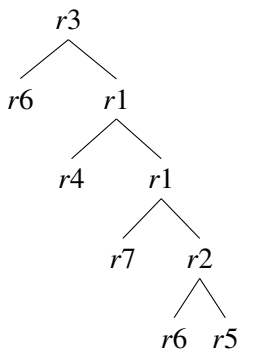

Figure 3.3: Inverse homomorphism applied to the tree in Figure 3.2.

once. In the case of the string algebra, the decomposition RTG consists of a rule for each word $A_{i-1,i} \to w_i$ for each $1 \leq 1 \leq n$ and rules $A_{i,k} \to \bullet(A_{i,j}, A_{j,k})$ for all $0 \leq i < j < k \leq n$ (Koller and Kuhlmann, 2011). In my example I have rules like $A_{0,1} \to the$, but also $A_{1,4} \to \bullet(A_{1,2}, A_{2,4})$. To visualize this, Figure 3.2 shows a tree that is generated by $D(a)$, where I added the nonterminals to the symbols of the tree for demonstration purposes.

3.3.2 Inverse Homomorphism

Next, I will discuss how trees over the algebra signature Δ can be converted to trees over the signature of $\mathcal{G}$, Σ. Since the homomorphism h transforms trees over Σ to trees over Δ, the *inverse homomorphism* h^{-1} reverses this function. The details of this process are not important to this thesis, but hopefully an example will make this clear: In h, $r1(NP,VP)$ would map to $\bullet(x_1,x_2)$. The latter however is far too ambiguous for a direct mapping, since there are three rules that are transformed to the concatenation. Therefore h^{-1} maps $\bullet(x_1,x_2)$ to $\{r1(NP,VP), r2(Det,N), r3(V,NP)\}$. Naturally, this can lead to a massive overhead.

To prepare the final step, I will apply the inverse homomorphism to the language of the decomposition RTG: $h^{-1}(L(D(a)))$. This leads to various trees over Σ, but with the original structure. Figure 3.3 shows one possible tree of the inverse homomorphism of the tree in Figure 3.2.

3.3.3 Intersection

To complete the parsing process, the correct trees have to be selected from the language of trees created by the inverse homomorphism. Conveniently, tree languages are closed under intersection, so they can be intersected with the language of the grammar $\mathcal{G}$. The parses for an input a of the interpretation I can therefore be described as:

$$parses_I(a) = h^{-1}(L(D(a))) \cap L(\mathcal{G})$$

If multiple interpretations are defined and a consists of multiple inputs, the parsing must be performed separately for each interpretation and the results intersected afterwards.

Most importantly for the following chapters is the structure of the parsing results, the "parse chart". It is described as a regular tree grammar, but here the states are tuples. These tuples consist of a state of $\mathcal{G}$ and one of the decomposition RTG $D(a)$. In the example, the following would be a valid state that covers the span $0,2$ with a NP: $\langle NP, A_{0,2} \rangle$

The Expectation-Maximization Algorithm

After splitting all states in a grammar, the weights of the rules have to be redistributed in a way to reveal latent information. Petrov and Klein, Prescher and Matsuzaki et al. use the expectation-maximization algorithm for PCFGs (Manning and Schütze, 1999) to learn new rule probabilities from a training corpus in an unsupervised way. The goal is to change the weights so that the sentences in the corpus are more likely to occur. In other words: they are maximizing the likelihood of the corpus. The training algorithm works by estimating how often a rule is used based on the inside and outside scores of a sentence.

4.1 Inside and Outside Probabilities for PCFGs

4.1.1 Inside Probabilities

Before I move on to the application of EM to IRTGs, it is useful to explain how the algorithm works on probabilistic context-free grammars, as described by (Manning and Schütze, 1999). For the sake of efficiency I assume that the grammar is in Chomsky normal form, consisting only of rules $A \to B\ C$ or $A \to a$, where A, B, C are nonterminals and a is a terminal symbol (Martin and Jurafsky, 2000). The inside score $\beta_J(p,q)$ describes the probability that the span (p,q) can be derived from the nonterminal symbol J. A span (p,q) describes the substring from the word at position p to the one at q. I define the base case as $\beta_J(k,k)$, where J covers a span of length 1. Hence, the probability that J generates the word w_k is the probability of the rule $J \to w_k$.

$$\beta_J(k,k) = P(J \to w_k) \tag{4.1}$$

In the inductive case $\beta_J(p,q)$ one has to sum up the results of two iterations: Finding all rules of the form $J \to R\ S$, and also splitting the span (p,q) into two parts at every possible position: $p < e < q$. Now it is possible to compute the inside probability of the first symbol R as $\beta_R(p,e)$ for the first part of the span and the one of the second symbol $S = \beta_S(e+1,q)$ for the remaining part. Eventually, both scores are multiplied by the probability of the rule $P(J \to R\ S)$.

$$\beta_J(p,q) = \sum_{R,S} \sum_{e=p}^{q-1} P(J \to R\ S)\beta_R(p,e)\beta_S(e+1,q) \tag{4.2}$$

4.1.2 Outside Probabilities

Assuming that the nonterminal J describes the span (p,q), the outside score $\alpha_J(p,q)$ is the probability of everything that lies left and right (outside) of the span. In the base case $\alpha_J(p,q)$, the span (p,q) covers the whole sentence $w_1, \ldots, w_m$ with $p = 1$ and $q = m$, leaving no words outside the defined span. If J is the start symbol S, the outside probability is defined as 1, otherwise 0. Since all scores are eventually multiplied by the result of the base case, it ensures that everything that does not lead to a starting state is assigned the probability 0.

$$\alpha_J(1,m) = \begin{cases} 1 & \text{if } J = S \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

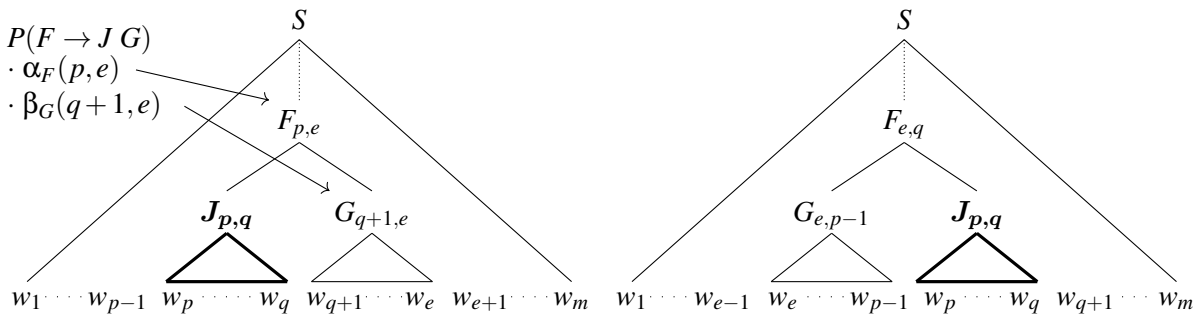

Figure 4.1: Illustration of the calculation of the outside score of J that covers the span (p,q).

In the inductive case for $\alpha_J(p,q)$, there are two possibilities: either J is the first symbol on the right-hand side of a rule or it describes the second one (see Figure 4.1). In the first case, the rule has the form $F \to J\ G$. To calculate the probability of what lies outside of the span of J one has to find the span covered by G. Since it is known that J describes the range (p,q) and it is the left child, the left boundary of G is $q+1$. The right boundary e is defined as $q+1 < e < m$ where m is the length of the sentence. In other words, G can cover any span from the next word to the rest of the sentence to the right. Finding e consequently defines the span of the entire rule produced by F as well: (p,e). The span is needed for the recursion of the outside score $\alpha_F(p,e)$ that computes the probability of what lies outside of F. Since it can also be the case that J is the second symbol, the rule must have the form $F \to G\ J$ and G can span from the beginning of the sentence to the symbol to the left of J. The results of both possibilities are summed up.

$$\begin{aligned}\alpha_J(p,q) = &\Big[\sum_{F,G}\sum_{e=q+1}^{m} \alpha_F(p,e)\ P(F \to J\ G)\ \beta_G(q+1,e)\Big] \\ &+ \Big[\sum_{F,G}\sum_{e=1}^{p-1} \alpha_F(e,q)\ P(F \to G\ J)\ \beta_G(e,p-1)\Big]\end{aligned} \tag{4.4}$$

4.2 An Abstract View on Spans and Trees

The computation of the inside and outside score is very well documented to work with strings. But in the case of IRTGs, the parse chart for an input a is not a single tree, but a language of all possible derivation trees $parses(a)$, usually represented as a RTG. Generating all derivation trees to use them for the calculation becomes unsuitable, since $parses(a)$ can describe a large or infinite language of trees.

Since RTGs can be interpreted in various ways, the concept of spans that describe a part of the input from position p to position q becomes too narrow. If one is using multiple interpretations with a string algebra, a symbol can derive spans of different lengths, depending on the interpretation. Other algebras like sets and graphs also cannot be described with spans. To solve both issues, I will take a step back and offer a more general solution to the idea of spans and parse trees.

4.2.1 Tree Automata

Thinking about a tree in case of the inside-outside algorithm makes sense, since it is easy to see the relations and dependencies between its nodes. Instead of iterating over possible rules for different spans and in fact building a single tree, I will now use a tree automaton (TA) that accepts all derivation trees that are generated by $parses(a)$. I define a tree automaton similar to a finite state automaton for string languages: $TA = (Q, F, Q_F, \delta)$. Q contains all states, $Q_F \subseteq Q$ are all final states and F contains the signature of symbols with ranks, similar to Σ in RTGs. δ describes the transition rules. Tree automata can either be top-down, starting at the start symbol of the grammar and using the states that generate symbols

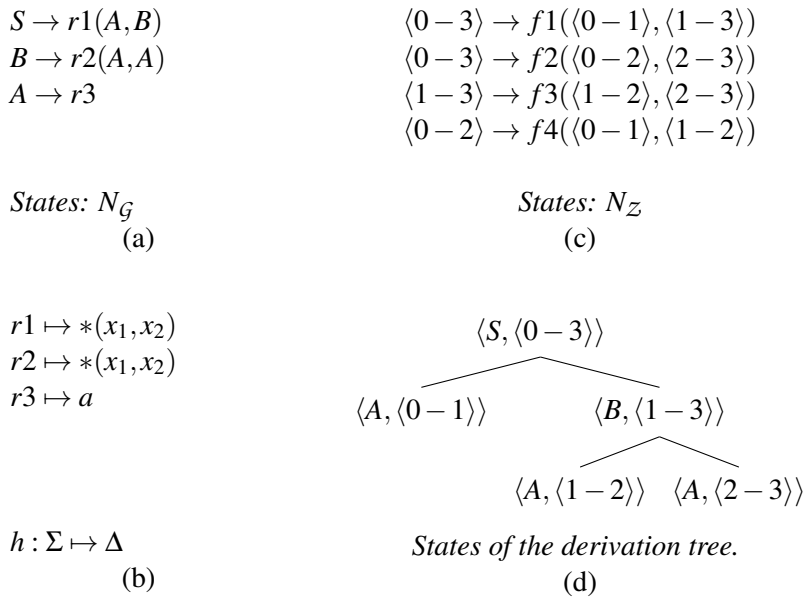

Figure 4.2: The structure of a parse chart. (a) is the original RTG with the mapping of the homomorphism of the string algebra in (b). (c) shows the RTG of the decomposition language of a string input of length 3 and (d) shows a derivation tree of the parse chart with pairs of the RTG in (a) and (c).

of rank 0 as final states. A bottom-up tree automaton on the other hand starts at the leaves and uses the state at the root as a final state. They also differ in the form of their transition rules. Take the RTG rule $A \to f(B,C)$ for example. If transformed to be used by a bottom-up tree automaton, it would have the form $f(B,C) \to A$, while the top-down automaton would use it as $A \to f(B,C)$.

Contrary to finite-state automata for strings, tree automata have no defined starting state. But by the logic of a bottom-up tree automaton, rules for symbols of rank 0 can be regarded as initial rules. A more detailed introduction to tree automata is presented by Comon et al. (2007). Just as regular grammars and final state automata are equivalent, one can transform a RTG to a TA. This makes it very helpful for me to formulate the inside and outside algorithm that I want to compute for states in $parses(a)$. I can now regard the parse chart as a tree automaton $TA(parses(a))$ as well and I will use the notations of RTGs and TA interchangeably.

4.2.2 Spans

In the following, I will generalize the concept of spans in the string algebra: the obvious way to interpret the spans assigned to symbols in a tree is that they provide information about the area of the sentence that can be derived from the symbol. One can also see them as a way to mark an exact position within the tree: the combination of a symbol and a span can never occur twice. Symbols may derive multiple parts of the input and a span can be generated from various symbols, but if a symbol derives a specific span, it marks a unique position in the tree. When I generalize from this perspective, I can neglect the positions (p,q) of a span and simply refer to it as some ζ. This abstraction stays true no matter what algebra is used to decompose an input, since its exact meaning to algebra can be neglected.

Consider the simple grammar $\mathcal{G}$ with nonterminals in $N_{\mathcal{G}}$ in Figure 4.2 (a) and a homomorphism for a string algebra in Figure 4.2 (b). I use it to parse the string *a a a*. In the process, the string algebra creates the RTG $D('a\ a\ a')$ with nonterminals in $N_{\mathcal{Z}}$ that describes all possible ways the three-word input could be decomposed (see Figure 4.2(c)). As explained in Chapter 3, intersecting $\mathcal{G}$ and $\mathcal{Z}$ results in the parse chart that uses tuples of nonterminals in $N_{\mathcal{G}}$ and $N_{\mathcal{Z}}$ as states. Figure 4.2 (d) shows the states of one of the two possible derivation trees. In the following, I will use ζ to describe states in the decomposition TA.

4.3 Inside and Outside Probabilities for Tree Automata

To formulate the definitions of inside and outside for tree automata, I can neglect the splitting of a binary rule that was required for PCFGs, as all information about the "spans" is already described in the states. PCFGs are required to be in Chomsky Normal Form, because the splitting of a span into arbitrary pieces would severely increase the complexity of the function. Due to the structure of the tree automaton, each symbol is already assigned to a part of the input and I can therefore generalize the definitions to use rules of any arity. The following definitions are based on the more general algorithms to evaluate tree automata in *Alto* that also store the results for each state to prevent redundant calculations.

4.3.1 Inside Probabilities

To compute the inside probability for a state in a parse chart, I first have to find all rules that lead to the state. In terms of RTGs, these are production rules with the state on their left-hand side. Contrary to the PCFG definition of the base case (see Equation 4.1) and the recursive case (see Equation 4.2), the base case is already part of my definition: If the rule has no children that would continue the recursion, the iteration uses the weight of the rule exclusively. Let $P(R)$ be the weight of the rule R:

$$\beta(A) = \sum_{\substack{R= \\ f(A_1,\ldots,A_n)\to A}} P(R) \cdot \prod_{i=1}^{n} \beta(A_i)$$

4.3.2 Outside Probabilities

However, for the outside scores of a tree automaton, it is simpler to distinguish between the recursive case (see Equation 4.4) and the base case (see Equation 4.3). To simplify the equations, I define the function $child(A)$ that describes a set of rules where A is a child. In terms of RTGs, A would occur on the right-hand side of the rule: $child(A) = \{f(A_1,\ldots,A_i) \to B \mid \exists i : A_i = A\}$. The base case for the symbol A is used if $child(A)$ is empty, because the recursion cannot continue. Let $child(A) = \emptyset$:

$$\alpha(A) = \begin{cases} 1 & \text{if } A \in Q_F \\ 0 & \text{otherwise} \end{cases}$$

The recursive case for PCFGs examines the possibility that the given symbol was the left or the right child and calculates the inside score only for the other state. To allow rules of any arity, I compare each child to A and for each match multiply the inside values of all other states. This follows the idea of finding the probability of everything that lies outside the given state. Let all rules R have the form $R = f(A_1,\ldots,A_n) \to B$:

$$\alpha(A) = \sum_{R \in child(A)} \sum_{\substack{i \in \\ \{i \mid \forall i : A_i = A\}}} \alpha(B) P(R) \prod_{j=1, j\neq i}^{n} \beta A_j$$

In detail, I search for all rules with A as a child and find the position of it in each. For the rule $R = f(X,A,Y) \to B$, A would be in position 2. Therefore, I multiply the inside value of the other children $\beta(X)$ and $\beta(Y)$ with the rule weight $P(R)$ and the outside score of the parent B.

4.4 Expectation-Maximization

4.4.1 PCFGs

If I write about training a grammar on a corpus, I actually mean the redistribution of the rule weights, so that the sentences in the corpus receive a higher likelihood. A higher likelihood means that my grammar

becomes more suitable to generate these sentences. In general, I define the weight of a rule as follows (Manning and Schütze, 1999):

$$\hat{P}(A \to \tau) = \frac{C(A \to \tau)}{\sum_T C(J \to T)} \tag{4.5}$$

$C(R)$ describes how often a rule is used, so the new weight is the number of occurrences of the rule in relation to the general frequency of its left-hand side symbol. This procedure works flawlessly for corpora that contain annotated trees for each sentence, since the absolute frequencies can easily be counted in the trees. However, most corpora are unannotated and therefore consist only of sentences without further information about the syntactic structure. To solve this hidden data problem, I *estimate* how often a rule is being used in a sentence based on the inside and outside scores. But first I will define the likelihood of a sentence as $\beta_S(1,m)$, since it is the probability that the start symbol S derives the whole sentence from position 1 to m. Next I need to find the probability that a symbol A derives the span (p,q). Logically, this is archived by multiplying the probability of everything that lies *inside* a span with the probability of everything *outside* of it. Hence I define $E(A,p,q) = \alpha_A(p,q) \cdot \beta_A(p,q)$. Extending this function to estimate the likelihood that A is used at any place in the derivation requires finding all possible values for the position p and q. To set it in relation to the likelihood of the sentence, I divide by $\beta_S(1,m)$,:

$$E(A) = \sum_{p=1}^{m} \sum_{q=p}^{m} \frac{\alpha_A(p,q)\beta_A(p,q)}{\beta_S(1,m)} \tag{4.6}$$

The next step is to estimate how often a specific rule is used in a derivation. I use a similar form as in Equation 4.6, but insert the inside values for the children, since they describe the span covered by the rule. The outside score of the left-hand side symbol on the other hand provides the probability of everything outside the span of the rule. Eventually, another iteration is needed to find a position to split the span of the binary rule for the first and the second children.

$$E(A \to J\ R) = \frac{\sum_{p=1}^{m-1} \sum_{q=p+1}^{m} \sum_{d=p}^{q-1} \alpha_A(p,q) P(A \to J\ R) \beta_J(p,d) \beta_R(d+1,q)}{\beta_S(1,m)} \tag{4.7}$$

Now that I have estimated the counts of a rule and its symbol, I can formulate the *maximization* step in which I assign new rule weights. Following my definition in Equation 4.5, I define:

$$P(A \to J\ R) = \frac{E(A)}{E(A \to J\ R)}$$

After substituting the equations:

$$P(A \to J\ R) = \frac{\sum_{p=1}^{m-1} \sum_{q=p+1}^{m} \sum_{d=p}^{q-1} \alpha_A(p,q) P(A \to J\ R) \beta_J(p,d) \beta_R(d+1,q)}{\sum_{p=1}^{m} \sum_{q=p}^{m} \alpha_A(p,q) \beta_A(p,q)}$$

To estimate the rule weights not only on a single sentence but on the whole corpus, I assume that all sentences are independent from each other. I can therefore sum up both the rule and the symbol estimations and then divide them as shown above. Note that I neglected the case for unary rules $A \to a$, but its derivation follows the same logic as the general case by inserting the inside calculation into the estimation of a symbol in Equation 4.6.

4.4.2 Tree Automata

Performing EM training on an IRTG means optimizing the weights of the RTG to increase the likelihood of the parse charts. In case of the PCFG, I did not explicitly calculate a parse tree for the sentence, even though the algorithm would require only minor changes to archive this. In contrast, I now rely on a previously calculated parse chart represented as a tree automaton. The following definitions are based in the implementation of *Alto*. Like described in Chapter 4.2.2, parse charts can be represented as TA with states of the form $\langle A, \zeta \rangle$, where A is a state of the IRTGs grammar $\mathcal{G}$ and ζ a state of the decomposition

grammar. From now on, I will refer to states in $\mathcal{G}$ with capital letters. The rules in the parse chart have the form $f(\langle A_1, \zeta_1 \rangle, ..., \langle A_n, \zeta_n \rangle) \rightarrow \langle A, \zeta \rangle$, where $f(A_1, ..., A_n) \rightarrow A$ is the corresponding rule in $TA(L(\mathcal{G}))$ while $f(\zeta_1, ..., \zeta_n) \rightarrow \zeta$ origins in the decomposition automaton $TA(L(D(a)))$.

Estimating the probability of a rule follows the same scheme as PCFGs (see Equation 4.5). I count how often the rule is used and divide by the number of occurrences of all rules with the same parent state. Like with PCFGs, I first want fo estimate how often a symbol A of $\mathcal{G}$ was used within a tree automaton (see Equation 4.6). Instead of finding spans, I iterate over all states in the parse chart and select those that contain the given symbol. Consequently, the probability of the whole sentence is replaced by the sum of the inside scores for all final states $\Upsilon \in Q_f$.

$$E(A) = \sum_{\langle A, \zeta \rangle \in Q} \frac{\alpha(A)\beta(A)}{\sum_{\Upsilon \in Q_f} \beta(\Upsilon)}$$

To estimate the counts of a rule, I apply the same idea as in Equation 4.7. Let the rule I want to estimate be $R_{\mathcal{G}} = f(A_1, ...A_i) \rightarrow A$. I now have to find all rules in the TA, where the states of $\mathcal{G}$ match the states of $R_{\mathcal{G}}$, so that I can multiply the inside probabilities of the children.

$$E(R_{\mathcal{G}}) = \frac{\sum_{R = f(\langle A_1, \zeta_1 \rangle, ..., \langle A_n, \zeta_n \rangle) \rightarrow \langle A, \zeta \rangle} \alpha(\langle A, \zeta \rangle) P(R) \prod_{i=1}^{n} \beta(\langle A_i, \zeta_i \rangle)}{\sum_{\Upsilon \in Q_f} \beta(\Upsilon)}$$

Following the procedure of the PCFG case, the maximization that assigns new weights for the rules is computed by dividing the estimation of the rule by the estimation of the parent state:

$$P(f(A_1, ..., A_n) \rightarrow A) = \frac{E(A)}{E(f(A_1, ..., A_n) \rightarrow A)} \tag{4.8}$$

Finally, the estimation can be generalized in the same way as PCFGs to take all parse charts in the corpus into account. To redistribute the rule weights for the complete grammar $\mathcal{G}$, I assign the result of the function in Equation 4.8 to all rules in $\mathcal{G}$. Expectation-maximization is a hill climbing algorithm that can run in multiple iterations and increase the likelihood of the corpus in each run. But one has to be aware that it can run into a local maximum where the likelihood does not increase anymore, even if the estimated rule weights are not optimal. When I refer to performing EM training in the following chapters, I assume that the training runs multiple times until the likelihood of the corpus increases only marginally. Chapter 6 discusses various values for this threshold.

State Splitting for IRTGs

At this point I have investigated the principles of state splitting for probabilistic context-free grammars and have also defined interpreted regular tree grammars and how expectation-maximization works on them. Now that the presupposed topics have been introduced, I will demonstrate how state splitting is performed on interpreted regular tree grammars.

5.1 Splitting a Grammar

5.1.1 Splitting

One of the benefits of IRTGs is that most algorithms are implemented directly on the underlying regular tree grammar, independently of the used interpretations. Hence, the splitting is also performed on the RTG, so that all algebras can benefit from it. To initialize the splitting process of the RTG $\mathcal{G} = (P, \Sigma, N, S)$, I start by creating a new RTG $\mathcal{G}_{Sp} = (P_{Sp}, \Sigma_{Sp}, N_{Sp}, S)$. Since the start symbol is not split, it stays the same as in $\mathcal{G}$. For each nonterminal symbol $A \in N$, I introduce two new ones, A^1 and A^2:

$$N_{Sp} = \{A^1, A^2 \mid A \in N\}$$

Consequently, I then replace the old nonterminals by their split counterparts in all the rules that they occur in and construct P_{Sp} at the same time. As demonstrated in Figure 5.1, a rule with arity 3 creates 16 new rules, each with an eighth of the original weight. Both A^1 and A^2 are also parent states, so I construct the same right-hand side twice for both of them. Note that the symbols are also changed in a way that they stay unique and are used only once. The exact naming of the symbol is irrelevant, as long as the homomorphism can map them to a term of a certain algebra.

5.1.2 Adding Randomness

Petrov et al. (2006) point out the importance of slightly altering the weight of the split rules, so that they are not equal. The grammar eventually learns new probabilities according to a corpus in the EM training. Depending on the training data, it may be necessary to break the symmetry of the rules, so that the EM training can find differentiations between them.

Like Petrov et al. (2006), I add 1% randomness to each rule weight to break the symmetry. This way, the EM algorithm will find an initial way to differentiate between the rules. I define the randomness as a pseudo-random number between 0.00 and 0.01, generated by the function ρ. I further assume that each result of ρ is a different one. These random number is then added to the weight p of all new rules in P_{Sp}:

$$P'_{Sp} = \{A \to f(A_1, ..., A_n)[p + \rho] \mid A \to f(A_1, ..., A_n)[p] \in P_{Sp}\}$$

5.1.3 Renormalization

For a probabilistic grammar to be valid, the weights of all rules with the same left-hand side have to sum up to one. After adding the randomness, it is most likely that this criterion is not met anymore. I therefore have to renormalize the rules, so that their weights sum up to one again, while the proportions stay the same. Let φ be the sum of the rule weights for a given symbol:

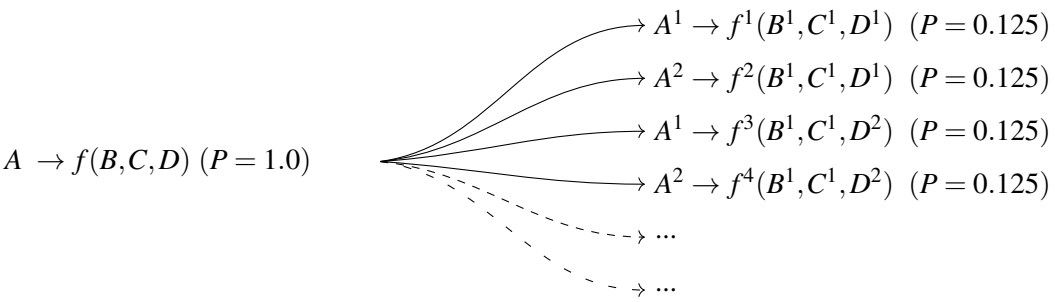

Figure 5.1: Demonstration of state splitting for an exemplary RTG rule. If all symbols are split, a rule with an arity of 3 is split into 16 new rules.

$$\varphi(A) = \sum_{A \rightarrow f(A_1,...,A_n) \in P'_{Sp}} P(A \rightarrow f(A_1,...,A_n))$$

To adjust the weights I then divide each rule weight by φ and redefine the set of rules P_{Sp}:

$$P_{Sp} = \{A \rightarrow f(A_1,...,A_n)[\frac{p}{\varphi(A)}] \mid A \rightarrow f(A_1,...,A_n)[p] \in P'_{Sp}\}$$

5.1.4 Tree Homomorphisms

Since the newly created rules use a different signature, the homomorphisms for all interpretations have to be rebuilt. Let h be the old homomorphism and h_{Sp} the homomorphism for Σ_{Sp}. As demonstrated in Figure 5.1, the symbol f is replaced by the symbols in $\{f^i \mid 1 \leq i \leq 16\}$, but they nevertheless should be interpreted in the same way as the old symbol:

$$h_{Sp} = \{f^i \mapsto h(f) \mid f^i \in \Sigma_{Sp}, f \in \Sigma\}$$

5.2 Identifying Important States

After splitting all symbols of the original RTG, I perform an EM training with the new grammar $\mathcal{G}_{Sp}$ on the training corpus. I expect the rules of symbols that have a hidden latent information to develop distinct probabilities.

But as discussed earlier, not all states contain as much hidden information as others and for the sake of performance should not be split after all. Hence, I have to compute a score to differentiate between them. The most obvious way to approach this problem is to ask how much the grammar benefits from a certain state being split.

This would be equal to performing EM training for the original grammar and for one with only one split state and then comparing the likelihood of the corpus. If the grammar with the split state performs noticeably better, it shows that this state should split. However, this selective comparison completely neglects the dependencies between states - split or otherwise. One could think for example of a certain type of adjective phrase that would only benefit from splitting if it occurs next to a direct object. But the differentiation if a NP is a subject, or a direct or indirect object, can only be made if the NP state is split simultaneously.

A better way to approach this problem is to ask a reversed question: How much does the likelihood of the corpus decrease if I split every state except a certain one? If it does not decrease noticeably, I know that the state is not suitable for splitting. But if the grammar worsens, I can be sure that there are some latent annotations being revealed by splitting the state. Since all the other states are split as well, dependencies between the states remain intact.

While this approach is working in theory, it is unrealistic to perform a time consuming EM training for every single state in the grammar. Depending on the size of the grammar and the training corpus, EM training can last between several hours or even up to a day. Therefore I have to use all information provided by a single training for a grammar with all states split to estimate the loss of likelihood for each state without the retraining process.

5.2.1 Recovering Inside and Outside Scores

Before answering the question of the loss of likelihood for a state based on the whole corpus, I will zoom in and answer the most fundamental question: Given a certain tree automaton representing a parse chart, what is the loss of likelihood in one specific position? I will further assume that there is a state $q_1 = \langle \zeta, A^1 \rangle$ in the parse chart that marks a specific position within the tree automaton, where a split state A^1 has been used. Now I need to find the state $q_2 = \langle \zeta, A^2 \rangle$ that uses the same state ζ from the decomposition automaton, but the sister state of A^1 as the second state. The original state on this position would be $\langle \zeta, A \rangle$.

Because I have already shown how to compute the inside and outside score, I can calculate the inside scores $\beta(A^1), \beta(A^2)$ and the outside scores $\alpha(A^1), \alpha(A^2)$. Before going any further, I want to answer the question what value the inside and outside scores had in the original RTG $\mathcal{G}$. Petrov et al. (2006) estimate that A combines the outside scores of its split states, so it is simply the addition of the scores of the split states:

$$\alpha(\langle A, \zeta \rangle) = \alpha(\langle A^1, \zeta \rangle) + \alpha(\langle A^2, \zeta \rangle) \tag{5.1}$$

Recovering the inside probability of A requires the weighting of both sister states. If one has appeared more often, it is more suitable to describe the connected states and consequently should be weighted higher than the other state. The calculation of the *relative frequency* is not described by Petrov et al. (2006), but the implementation of the Berkeley Parser can help to clarify this term. As explained in Chapter 4, the likelihood of a certain state can be estimated by multiplying its inside and outside score. Therefore, I define $e(B) = \beta(B)\alpha(B)$. The relative frequency of a state A^1 at one position in the parse chart can then be computed by putting its likelihood estimation in relation to the likelihood of A^1 and its sister state A^2:

$$p(\langle A^1, \zeta \rangle) = \frac{e(\langle A^1, \zeta \rangle)}{e(\langle A^1, \zeta \rangle) + e\langle A^2, \zeta \rangle)}$$

Next, I will expand this Definition for all states in the parse chart and all parse charts in the corpus. Let T be a set of all parse charts in the corpus and Q_{Zt} be all states in the decomposition automaton for the parse chart t. By summing up the results for all ζ in Q_{Zt} and all parse charts $t \in T$, I can estimate the relative frequency for a nonterminal in $\mathcal{G}_{Sp}$:

$$p_T(A^1) = \frac{\sum_{t \in T} \sum_{\zeta \in Q_{Zt}} e(\langle A^1, \zeta \rangle)}{(\sum_{t \in T} \sum_{\zeta \in Q_{Zt}} e(\langle A^1, \zeta \rangle)) + (\sum_{t \in T} \sum_{\zeta \in Q_{Zt}} e(\langle A^2, \zeta \rangle))}$$

Eventually, I can formulate my definition for the estimated inside score for a state in $\mathcal{G}$ that takes the frequencies of the split states into account:

$$\beta(\langle A, \zeta \rangle) = p_T(A^1)\beta(A^1, \zeta) + p_T(A^2)\beta(A^2, \zeta) \tag{5.2}$$

Recovering the likelihood of a state at a specific position can then be described by substituting Equation 5.2 and Equation 5.1 into the definition of e:

$$e_r(\langle A, \zeta \rangle) = (p_T(A^1)\beta(\langle A^1, \zeta \rangle) + p_T(A^2)\beta(\langle A^2, \zeta \rangle)) \cdot (\alpha(\langle A^1, \zeta \rangle) + \alpha(\langle A^2, \zeta \rangle)) \tag{5.3}$$

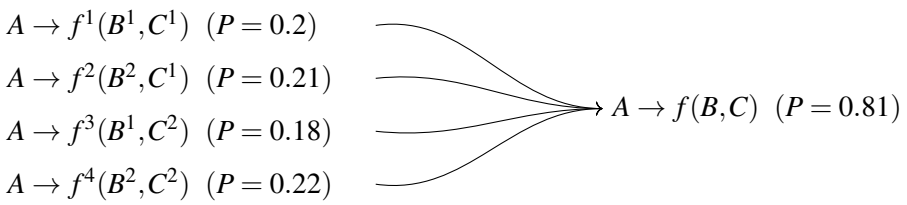

Figure 5.2: Merging rules for the same nonterminal on the left-hand side.

5.2.2 Estimating the Loss in Likelihood

To estimate the loss in likelihood that would be caused by the merging of a state A, I rely on two scores: One that describes its likelihood in the split grammar and one that recovers the likelihood of A in $\mathcal{G}$, where it has not been split. The latter has been described in the previous section. To find out the score for a split A, I face the problem that A does not exist anymore, as it has been replaced by A^1 and A^2. To estimate its likelihood including the latent information, I can simply sum up the values of the split states:

$$e_l(\langle A, \zeta \rangle) = e(A^1) + e(A^2) \tag{5.4}$$

Now I have introduced one score for the split grammar in Equation 5.4 and one for the original grammar in Equation 5.3. To calculate the loss in likelihood, I divide the likelihood in the grammar with latent annotation by the score in the original one. After extending the calculation for the whole corpus and all states in the tree automaton, I define the loss in likelihood for a symbol in N as follows:

$$\Delta(A) = \prod_{t \in T} \prod_{\zeta \in Q_{Zt}} \frac{e_l(\langle A, \zeta \rangle)}{e_r(\langle A, \zeta \rangle)}$$

The loss will be calculated for all nonterminals in N of the original grammar $\mathcal{G}$ and sorted. Values less than zero suggest that the likelihood will worsen and the state should be split. Petrov et al. (2006) suggest splitting the best 50% of the states. Note that depending on the grammar and corpus, some of the best 50% of the states could still worsen the grammar or add no additional benefit, if they have received a score of zero or greater. Therefore I suggest ignoring these states.

5.3 Merging

To merge selected states in a grammar, I simply reverse the process of splitting. Instead of replacing one state by two new ones, I search all rules for occurrences of split states and replace them by the original one. Consequently, this leads to duplicated rules that only differ in their symbol and their probability, as shown in Figure 5.2. To decide if rules with different symbols were originally the same rule, I have to take the interpretations into account. Consider the following rules for example: $A \to f^1$ and $A \to f^2$. Assuming that A was previously a split state, one might be tempted to merge both rules into a single one. But it can likely be that homomorphisms will perform differently for f^1 and f^2. Therefore I define equality of merged rules in a more explicit way :

Definition 5.3.1. Rule equality The rules of a RTG $A \to f(A_1, ..., A_n)$ and $B \to g(B_1, ..., B_n)$ are equal if and only if $A = B \wedge A_1 = B_1 \wedge ... \wedge A_n = A_n$ and the interpretations of f is equal to the interpretation of g. For all interpretations of an IRTG $I_1, ..., I_n$ with their respective homomorphisms $h_1, ..., h_n$ the interpretation of two symbols f and g are equal if and only if for all homomorphisms $h_1, ..., h_n$: $h_1(f) = h_1(g) \wedge ... \wedge h_n(f) = h_n(g)$.

To merge the probability of different rules that have found to be equal under Definition 5.3.1, one can make out two distinct cases. If the symbol on the left-hand side is not a state that should be merged, I can simply sum up the probabilities as shown in Figure 5.2. All four former different rules are now replaced by a single one with the combined weight of all of them.

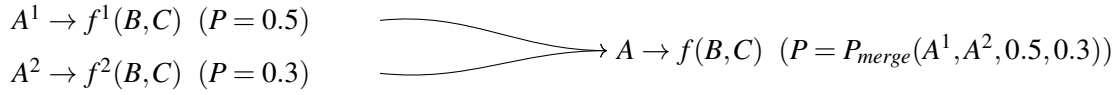

Figure 5.3: Merging rules for the same right-hand side. The rules for each split symbol is multiplied by its estimated frequency.

On the other hand, if the symbol in the left-hand side is a split symbol, the rule weights must be normalized. Let me assume that A^1 and A^2 are now merged to state A. All rules of A^1 have weights that sum up to one and also all rules of its sister state sum up to one as well. To maintain a valid probability model, both weights have to be combined. Like estimating the probability of a state, I weight the probability of both rules by the relative frequency of their parent states. Therefore, I multiply the weight of the rule of A^1 by $p_T(A^1)$ and add A^2 multiplied by $p_T(A^2)$. Eventually I normalize it by dividing by $p_T(A^1) + p_T(A^2)$ as shown in Figure 5.3. In detail, merging a rule for a state A^1 and probability p_1 and a rule for A^2 and weight P_2, is defined as:

$$P_{merge}(A^1,A^2,p_1,p_2) = \frac{p_1 p_T(A^1) + p_2 p_T(A^2)}{p_T(A^1) + p_T(A^2)}$$

In the process, I have created a refined grammar by splitting all symbols and subsequently merging those that did not contribute to the likelihood of the corpus. This split-merge cycle can be repeated several times to reveal more latent annotations.

Results and Discussion

In this chapter, I demonstrate the performance of my implementation of the state-splitting algorithm that is integrated into *Alto*. First, I explain the creation of a minimal Penn Treebank grammar and the differences to the implementation of Petrov et al. (2006). Subsequently, I discuss the results and use a small hand-written grammar to prove the general effectiveness of my implementation.

6.1 Experimental Setup

6.1.1 Data

The IRTGs that I will use for my experiments are created with *Alto* based on the sections 2-21 of the Penn Treebank. They contain an interpretation for the strings and one for trees of strings. For the training process, I create a corpus based on the same section 2-21. To save time in the EM training, I exclusively parse the trees and not the strings. Trees have the advantage of already providing a syntactic structure that makes the creation of the derivation language easier. The trees were modified in exactly the same way as I will describe for the grammar, resulting in 39.794 trees for the training process.

6.1.2 Tree Transformation

Like Petrov et al. (2006), I perform various transformations to the trees in the corpus. My goal is to create a simple grammar that does not use more symbols and rules than necessary. During the split-merge cycle, the amount of rules will increase dramatically and therefore I want to start with the lowest possible amount of rules. Another reason is to open the search space for the algorithm by letting it find the actually important annotations itself. The first step is to replace all words at the leaves by their part-of-speech tag to create an unlexicalized grammar. Since the grammar now does not require a single rule for each lexical item, their number decreases drastically.

Additionally, I introduce a super start symbol "TOP". State splitting does not allow the start symbol to be split because it can lead to inaccuracies when computing the inside scores. By using "TOP" that produces the former start symbol I, allow the algorithm to discover latent information in "S".

Petrov et al. (2006) state that they remove all unnecessary annotations to the trees, but they do not describe this procedure in detail. Therefore, I present various transformations and how they affect the number of symbols and rules, as well as the f-score. The first strategy is to remove all branches containing

Removing:	NONE	Index, NONE	Annotations, NONE	Baseline
States	1.168	295	73	1.226
States (binarized)	63.037	56.056	38.221	65.849
Rules	31.616	26.364	15.014	31.734
Rules (binarized)	93.485	82.125	53.162	97.357
Time	250min	137min	61min	202min
F_1	68.39	69.32	70.54	69.8

Table 6.1: Analysis of different tree transformations. Parsing was performed on the binarized grammar, distributed on 20 CPUs. The f-score calculated for the first 100 sentences in Penn Treebank section 22 with length ≤ 20.

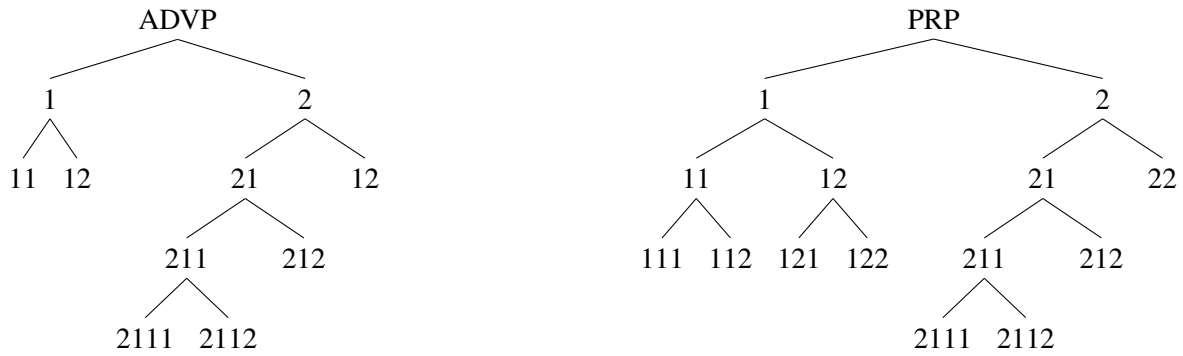

Figure 6.1: The hierarchy of the splitting of the symbols “ADVP” and “PRP”.

the “-NONE-” element. It was originally used to mark the position from where a phrase was moved in the sentence and is therefore empty. No lexical word will ever be derived from this category. The next approach is to remove the index annotation from nonterminals like “NP-SBJ-1”. Depending on the symbol, there are up to 20 different annotations and removing them can decrease the size of the grammar noticeable. Eventually, another strategy is to remove all annotations that are appended to the symbol with a hyphen. Therefore, the remaining names are very general. I test three combinations of these techniques and compare the number of rules and states, and the time to parse 100 sentences from section 22 that contain no more than 20 words. I also compute the f-score using EVALB. For the baseline, I use the scores of a grammar that is unlexicalized and uses “TOP” as a super start symbol.

The analysis of the results in Table 6.1 shows that removing all annotations and “NONE” elements reduces the number of states to almost 10% of the baseline grammar. The huge difference between the states in the normal and the binarized grammar surprises, but one has to keep in mind that the trees themselves have not been binarized and that even rules with 51 children exist in the grammar. It is promising to see that the parse time has been reduced to 30%, while the f-score stayed approximately the same. The measurement of time should only be used to see a tendency, since it can highly be influenced by other processes on the server. It is also interesting to see that removing only “-NONE-” leads to an increase in time and the worst f-score.

As a result of the comparison, I will accept the trade-off between time and f-score performance and use a minimal grammar without any annotations or “-NONE-” rules. Even though I speculate that the difference in f-score is higher on a complete test set, I follow the approach of Petrov et al. (2006) and assume that state splitting will cover this up.

6.1.3 Binarization

Even though my algorithms for expectation-maximization in Chapter 4 and for state splitting in Chapter 5 can be performed for rules with any arity, a binarized grammar will improve the overall performance. In contrast to Petrov et al. (2006), I do not binarize the trees in the corpus, but the IRTG itself. State splitting should perform well for any algebra, even if the input is not binarized. Binarization for IRTGs works by introducing new rules to the RTG to describe a high arity rule with multiple binary ones. During this process, the interpretations are not changed. This way, the tree interpretation will still contain trees of any arity. The newly introduced states will be ignored by the splitting algorithm. Again, I use *Alto* to perform the binarization.

6.2 Differences to Petrov et al. (2006)

The following experiments are inspired by Petrov et al. (2006), but not an exact replication of their set-up. While my implementation is concentrated on basic state splitting, they also introduce methods for smoothing and inference (Petrov and Klein, 2007) that are not part of this thesis. Petrov et al. (2006) report that smoothing the rule weights helps to prevent overfitting and results in an additional rise in f-score.

WHADJP	35	WHPP	8	SQ	4	MD	3	NN	2	PDT	2
SINV	34	QP	7	RRC	4	VBN	3	WHNP	2	JJ	2
X	34	NAC	7	UCP	4	EX	3	JJR	2	NP	2
WHADVP	19	CONJP	7	.	4	S	3	VBP	2	WRB	2
PRT	14	ADVP	6	PRP$	4	SBARQ	2	IN	2	WRB	2
LST	14	VBZ	6	PP	3	RRB	2	RB	2	FRAG	2
INTJ	14	VBZ	6	DT	3	PRN	2	ADJP	2	WP	2
PRP	8	WDT	5	NNP	3	JJS	2	VP	2	SBAR	2

Table 6.2: Number of latent annotations after six split-merge cycles.

With their coarse-to-fine approach, they were able to drastically lower the parsing time by pruning the search space for the parser. Additionally, Petrov et al. create a minimal binarized X-bar grammar for their experiments. My IRTG on the other hand may have a binarized RTG, but the string and tree interpretation still describe the original trees, where some symbols branch into more than 50 children.

Another difference lies in the used test data. As usual, Petrov and Klein evaluate their system on sentences of the Penn Treebank section 23 that contain forty words or less. Unfortunately, the performance of the implemented parser in *Alto* makes it impossible to process such long sentences for a grammar that has been split several times. Therefore, I restrict my evaluation to 1.034 sentences with twenty words or less. Since Petrov and Klein implemented their algorithm into the Berkeley Parser, they most likely rely on other features that further improve their performance.

6.3 Analysis: Penn Treebank Grammar

I start my experiments by performing six split-merge cycles on an IRTG. In each cycle, I keep the best 50% of the newly split symbols. Note that states that are ignored by the algorithm like the ones created during the binarisation or the start symbol are excluded from this calculation. As a threshold for the EM training, I choose 10^{-8}, causing the training cycles to stop, as soon as the change in likelihood drops below this threshold.

First, I analyze how some symbols were split in detail. See “ADVP” and “PRP” in Figure 6.1 for example. The distinct hierarchy of the splits demonstrates the structure of revealed latent information inside the symbols. There seems to be one part of the “ADVP” symbol that contains deep hidden information. Except for “11” and “12” of each new state pair, only one symbol will be split again resulting in the often split states “2111” and “2112”. One part of the “PRP” symbol is split in the very same way, while the other part seems to contain more equally distributed information. Splitting the symbol “PRP1” results in new splits that are both split again. Here the hidden information seems to be equally partitioned to “111”, “112”, “121” and “122”. Since the used IRTG is unlexicalized, the actual words described by the split symbol cannot be recovered.

The absolute numbers how often all symbols were split are presented in Table 6.2. Surprisingly, rather uncommon categories like “X” or “WHADJP” have been split most frequently, while part-of-speech tags like “NNS” are barely split at all. Comparing these results to those of Petrov and Klein reveals that most of their frequently split symbols have not been split at all by my implementation. However, symbols that they have split only once or twice are most common in my results. One may interpret these unexpected results as wrong rating in my implementation, but it is more complicated. I will revisit this behavior in Section 6.5.

Figure 6.2 demonstrates how dramatically the amount of rules increases in each split-merge iteration. While the original grammar contained 53.162 rules, each iteration adds about 30.000 - 50.000 new rules. After six cycles, the grammar contains 270.568 rules, five times as much as in the beginning. The number of newly introduced states increases from 32 to 261 between the first and last cycles. Please note that the numbers are compared to the amount of states in the original grammar. Therefore, splitting a single symbol will lead to only one new symbol, since the original one is replaced by two.

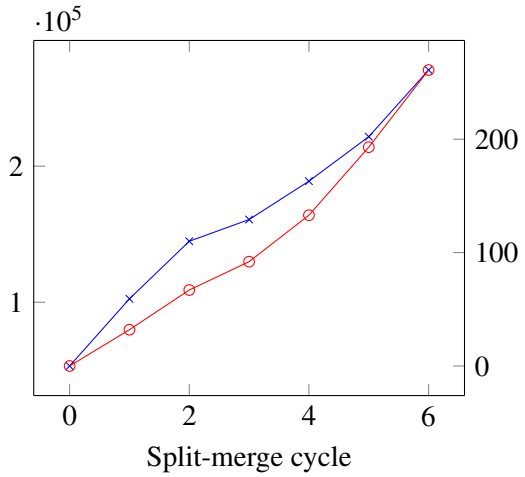

Figure 6.2: The increasing number of rules (blue) on the left axis and added states (red) on the right.

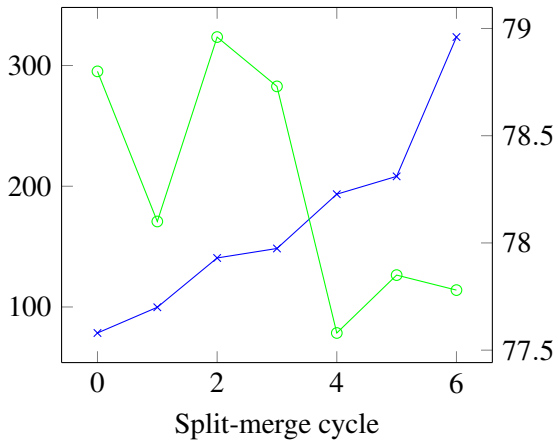

Figure 6.3: Parse time for test sentences in minutes (blue) on the left and the f-score (green) right.

Next, I analyze how the increased size of the grammar affects the time to parse the test data. The parsing was performed on twenty CPUs simultaneously with 256GB memory. One has to take into account that multithreading comes with an additional processing overhead, therefore the presented values should only be seen as a rough estimate. Nevertheless, a clear tendency becomes visible as the parsing time increases. While the original grammar required 76 minutes of CPU time, it increases to 324 minutes after six cycles. These dramatic results emphasize the importance of pruning techniques that were used by Petrov and Klein (2007) to speed up the parsing process.

When regarding the f-scores for the various grammars in Figure 6.3, it becomes obvious that the implementation has shortfalls when working on treebank sized data. While the f-score of the grammars of Petrov et al. (2006) increased notably with the number of new grammar symbols, the values in my experiments stay roughly the same, between 79 and 77.

6.4 Analysis: Small Grammar

Since it became obvious that the state-splitting algorithm did not perform as expected on the data from the Penn Treeebank, I demonstrate its performance on a handwritten grammar.

The small grammar contains only a rule for the start symbol $S \to r1(NP, VP)$, one for a nominal phrase $NP \to r2(Det, N)$ and one for the verbal phrase $VP \to r3(V, NP)$. I also add two possible rules for the determiner, four rules for different verbs and four rules for nouns. Like in the previous IRTG, this grammar can interpret the symbols as strings and trees of strings. Since there is only one possible syntactic structure for S, NP and VP, I create a training corpus that contains the same structure but with different linguistically correct combinations of verbs, nouns and determiner, resulting in 64 different trees.

As the changes do not occur not in syntactic structure, but in the preterminal rules I expect multiple splits for the determiner and nouns, as well as a few splits for the verb or verbal phrase. This falls in line with the expectations that Petrov et al. (2006) formulate for their experiment.

After only two split-merge cycles, the likelihood of the corpus increases only marginally. As I expected, the determiner was split two times and the noun four times. I interpret this as finding a distinct rule for each word. This distinction benefits the multiple new rules for "NP" that have been created in the process, so that every used combination of determiner and noun is described in a separate noun phrase rule. Also the symbol for the verbal phrase has been split two times in the second iteration. This could be caused by different verbs in "V" that influenced their parent state "VP". Due to the randomness in the rule weights and the hill-climbing nature of the EM algorithm, the exact values differ slightly in each new run. Nevertheless, the observed trend stays the same.

Iteration	Added symbols	Rules	F-score	Time for state splitting	Time for parsing
0 (baseline)	0	53.162	78.78	-	78
1	32	102.515	78.10	30	99
2	67	144.805	78.96	37	141
3	92	160.746	78.73	44	147
4	133	188.940	77.58	45	192
5	193	221.631	77.85	48	207
6	261	270.568	77.78	55	324

Table 6.3: Creating and evaluating an IRTG with state splitting. Time in minutes, parts of the calculations were performed in multithreading on 20 cores. Parsing was performed on the sentences of the section 23 of the Penn Treebank with 20 words or less.

6.5 Discussion

The analysis of the split hierarchy for two symbols has demonstrated that my implementation is capable of discovering hidden information for states. However, the general investigation reveals problems in selecting the right states to split. Consequently, the f-score does not rise as expected, but remains roughly the same value. However, in my experiments with the small hand-written lexicalized grammar, the state-splitting algorithm showed the expected behavior.

If my implementation works on a grammar with a simple corpus, why does it not scale to a treebank grammar? I suspect a loss of precision of some of the used `double` variables. These are used to store rule probabilities, inside and outside scores and the merge weights as well. In Java, a variable of the data type `double` will be rounded to 0, if its value drops below 10^{-323}. While this seems to be a very low value at first, it turns out that the result of many calculations that are crucial for state-splitting algorithm are below this threshold. In the original grammar, rule probabilities of 10^{-6} are not uncommon. In the computation of the inside and outside scores however, these rule weights are multiplied by other rule weights many times, resulting in values between 10^{-120} and 10^{-200} for states that appear in many rules. Subsequently, the estimation of the merge weights and the recovered inside and outside scores can heavily decrease the number. Consequently, the scores for common symbols drop to zero and mislead the calculation of the estimated loss in likelihood caused by the symbol, as well as the computation of the rule weights. In fact, after the first split-merge cycle 25% of the rules had a probability of 0.0. After six iterations, their amount increased to 35%. These rules can further cause severe damage if their weight is used in a multiplication and causes the whole calculation to result in zero.

Analyzing the merge weights and the occurrences of a symbol in the grammar confirm this suspicion. The symbols that have been split by Petrov et al. (2006) most often (“NNP”, “JJ”, “NNS”, “NN”) are used by my baseline grammar several thousand times. However, their merge weights in the first split-merge cycle are 0. On the other hand, the symbols that have been highly split by my implementation (see Table 6.2) occur in average only 80 times. But even if the loss in precision is the main cause for the malfunctioning implementation, there are still open questions. For example, how is it that the Berkeley Parser does not experience this kind of problem? My implementation is handling cases like zero probabilities in exactly the same way. Both the Berkeley Parser and my implementation are using logarithmic values to prevent the precision loss in the same situations. A reason could be binarization of the grammar. Petrov et al. (2006) binarize their PCFG in X-bar style with 4.076 rules. The binariziation of the IRTG however results in more than 50.000 rules. Since the probability of each of them is used in the estimation process, they could have a huge impact on the result. Another question would be why the symbols “NP” and “VP” are split exclusively in the first iteration. With 12.000 and 6.000 occurrences, they are the most common symbols in the grammar. My implementation is summing up the logarithmic merge weights for all positions in the parse trees and ignores all invalid results. For the extremely common “NP” and “VP” symbols, I suspect that there have to be at least a few positions with a valid value.

6.6 Future Work

While I have shown that my implementation is performing well for small grammars, the program has to be improved to work on treebank sized grammars and corpora. To prevent a continuous loss in precision I suggest a detailed examination of the EM algorithm and the calculation of the merge weights. It must be avoided that scores of states that are holding crucial information are dropping to zero. Additionally, it should be examined how one should handle those states and rules that have lost precision to prevent them from harming future calculations.

Shindo et al. (2012) have introduced a symbol refinement algorithm for tree substitution grammars and rely on the Pitman-Yor Process for their Bayesian probability model. It is worth a thought to abandon the inside and outside scores and use a more sophisticated model instead.

Contrary to Petrov et al. (2006), who remove any kind of annotation from the corpus trees and completely rely on their algorithm, Bansal and Klein (2010) report an improvement by additionally applying parent annotation to the trees. The idea of including every available information seems tempting, but it must be considered that the introduced algorithm for IRTGs has to perform for any algebra and not exclusively on strings or trees of strings.

Nevertheless, all notable state-splitting algorithm in the last years included some kind of smoothing and inference. Therefore I am confident that future versions of my implementation will highly benefit from these methods.

Conclusion

In this bachelor thesis I have investigated the concept of automatic state splitting and how it can be applied to interpreted regular tree grammars.

The idea of refining a probabilistic context-free grammar by splitting its states is known since the 1990s. Early approaches tried to weaken its independence assumption by encoding the position of symbol in a tree into its label. One of the most common strategies is parent annotation, where the syntactic category of the superordinate node is attached to the symbol. While these ideas have proven that the original annotations in treebanks are often too general, applying the transformations to all symbols results in a very large grammar. Therefore, researches selected only a few promising states by hand.

Automatic state splitting however is able to make this decision without any linguistic knowledge. In the presented algorithm by Petrov et al. (2006), a score for each symbol is computed that states whether the grammar would benefit from splitting the symbol or not. It is also beneficial that it can be applied to treebanks of any domain or language. Interestingly, these automatically created grammars can be linguistically interpreted, proving that the algorithm indeed reveals latent information.

My goal was to apply this algorithm to the more expressive interpreted regular tree grammars. In the process, I explained the structure of IRTGs and outlined the parsing process. This information is crucial to applying the expectation-maximization algorithm to IRTGs. EM training is an unsupervised learning algorithm that is needed to find better weights for the rules of the split grammar.

Eventually, I presented in detail how the state-splitting algorithm can be adjusted to IRTGs. Subsequently, I perform the implemented algorithm on an IRTG that has been created based on the Penn Treebank. Unfortunately, the increasing size of the grammar causes problems with the precision of a data type in Java. As a consequence, necessary scores for frequent symbols cannot be calculated and the created grammar performs poorly compared to results of Petrov et al. (2006).

However, the algorithm acts well on smaller grammars, proving that my implementation is indeed correct. Finally, I briefly discuss two state-of-the-art publications that show more sophisticated methods of how state splitting can be applied on grammar formalisms other than PCFGs.

Overall, I have demonstrated that state splitting can be performed on interpreted regular tree grammars and can therefore be applied to IRTGs that represent expressive grammar formalisms like tree adjoining grammars or linear context-free rewriting systems.

References

Steven Abney, S Flickenger, Claudia Gdaniec, C Grishman, Philip Harrison, Donald Hindle, Robert Ingria, Frederick Jelinek, Judith Klavans, Mark Liberman, et al. Procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proceedings of the workshop on Speech and Natural Language*, pages 306–311. Association for Computational Linguistics, 1991.

Mohit Bansal and Dan Klein. Simple, accurate parsing with an all-fragments grammar. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1098–1107, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1858681.1858793`.

Eugene Charniak. Tree-bank grammars. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1031–1036, 1996.

John Cocke. *Programming languages and their compilers: Preliminary notes*. Courant Institute of Mathematical Sciences, New York University, 1970.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, February 1970. ISSN 0001-0782. doi: 10.1145/362007.362035. URL `http://doi.acm.org/10.1145/362007.362035`.

Mark Johnson. Pcfg models of linguistic tree representations. *Computational Linguistics*, 24(4):613–632, 1998.

Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

Dan Klein and Christopher D Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

Alexander Koller. Semantic construction with graph grammars. In *Proceedings of the 14th International Conference on Computational Semantics (IWCS)*, London, 2015.

Alexander Koller and Marco Kuhlmann. A generalized view on parsing and translation. In *Proceedings of the 12th International Conference on Parsing Technologies*, IWPT '11, pages 2–13, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. ISBN 978-1-932432-04-6. URL `http://dl.acm.org/citation.cfm?id=2206329.2206331`.

Christopher D Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

James H Martin and Daniel Jurafsky. Speech and language processing. *International Edition*, 2000.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. Probabilistic cfg with latent annotations. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 75–82. Association for Computational Linguistics, 2005.

Slav Petrov and Dan Klein. Learning and inference for hierarchically split pcfgs. In *Proceedings of the National Conference on Artificial Intelligence*, volume 22, page 1663. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics, 2006.

Detlef Prescher. Inducing head-driven pcfgs with latent heads: Refining a tree-bank grammar for parsing. In *Machine Learning: ECML 2005*, pages 292–304. Springer, 2005.

Satoshi Sekine and Michael Collins. Evalb bracket scoring program. *URL: http://www. cs. nyu. edu/cs/projects/proteus/evalb*, 1997.

Hiroyuki Shindo, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata. Bayesian symbol-refined tree substitution grammars for syntactic parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Long Papers-Volume 1*, pages 440–448. Association for Computational Linguistics, 2012.

Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967. ISSN 0019-9958. doi: http://dx.doi.org/10.1016/S0019-9958(67)80007-X. URL `http://www.sciencedirect.com/science/article/pii/S001999586780007X`.