Spectral Learning for Weighted Logic Programming with Latent States

Xiaochen Zhu

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

Logic programming is a flexible formalism for denoting and inferring relationships between items in the form of a predicate with its arguments. Weighted logic programming further extends this formalism to associate a weight with each of these items that are inferred or assumed to be true as an axiom. Balkir et al. [4] showed how weighted logic programmes can be generalised to include latent states from an arbitrary semiring, improving the expressivity of the weight associated with each inferred item so that it is in the form of a vector. The problem of learning such WLPs remained open. In this paper, we identify a subset of such WLPs that can be learned using a spectral algorithm. We also show that certain existing programs, such as those for dynamic programming for parsing algorithms, can be transformed into programs in this set of WLPs, so that our spectral algorithm generalises that of Hsu et al. [16] and Cohen et al. [7].

Acknowledgements

Table of Contents

Chapter 1

Introduction

Weighted logic programming (WLP) has demonstrated its potential over the past decade as an important technique for probabilistic reasoning, which has many successful applications in machine learning, natural language processing [21], computational biology [9] and end-to-end differentiable proving[23]. WLP offers a general bottom-up logic inference model that every provable item takes a numerical value which is interpreted as a probability. One of the advantages is that many scenarios can be fit into such a model to perform probabilistic inference. Many algorithms are driven by seeking the "most probable" solution for the programme [5].

Weighted logic programming has a certain formalism by assuming the existence of three underlying 3 essential random variables I, A and P which has the following interpretation.

1. I, a set of conditional inputs, encoded as axioms.
2. A, a set of axioms known to be true, the axioms we use in proofs.
3. P, a deductive proof of the goal theorem using axioms.

In most of the cases, we are calculating the probability distribution of certain proofs, finding $p(P|I,A)$[6]. To put it out more clearly, given certain facts or premises and a goal theorem we want to prove, there are many possible distinct proofs to our goal. We are evaluating the probability of each proof. In addition, sometimes we are not only interested in the final goal, but we also care about the path we've been through. For example, when we are dealing with finite-state transducers, the emissions on each path are likely to be meaningful and distinct.

Now we provide a toy example to elaborate the idea of weighted logic programming. Consider we are predicting how does a cell grow via binary fission by using the isotope labelling method [25]. The offspring will carry radioactive material proportional to its generation. More generation means less radioactive. Let's say observed 5 radioactive cells in total over a few minutes after we labelled the only ancestor with 1μg isotope. Then we have the following rules and observations under this scenario.

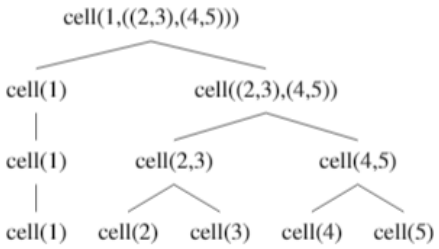
- $I = \{Radioactive(1), ..., Radioactive(5)\}$

- $A = \{r_1 = cell(X) \rightarrow Radioactive(X), r_2 = cell(X,Y) \rightarrow cell(X)\ cell(Y),$
  $r_3 = cell(X) \rightarrow cell(X)\}$

$Radioactive(X)$ indicates the amount of radioactive material in the cell, r_1 is an identity rule, r_2 is describing 2 children cells has a common ancestor where fission happens, r_3 is describing a cell stays static. In addition, we know or **learned** a fact that this type of cell likes to grow. So we have the corresponding parameters as below.

- $Radioactive(X) = 1$
- $cell(X) = Radioactive(X) * 1.0$
- $cell(X,Y) = cell(X) * cell(Y) * 0.9$
- $cell(X) = cell(X) * 0.1$

Intuitively we could come up with the following proof to backtrack the generation of these 5 cells with the proof tree as below. There are many possible proofs and we would like the one with the lowest cost. This type of problem is typically solved by dynamic programming [11]. Here we expand our proof for the sake of clarity.



$$P = cell(1,((2,3),(4,5)))$$
$$= cell(1) * cell((2,3),(4,5)) * 0.9$$
$$= 0.1^2 * 0.9^4$$

Figure 1.1: Proof Tree(Left) Calculation(Right)

There are many fields of the problem that could fit into this model. For example, probabilistic parsing to find syntax tree structure [20] or branching process that involves recursive rules in biology [13]. As mentioned before, there could be multiple proofs available for a single goal. Since each proof has a value as its probability or weight, we could find an optimal proof by taking the one with the largest probability. However, in the current setup, there are 2 key questions unresolved.

Firstly, there are no global structural preferences or dependency structures that could be grasped inside a proof. Replacing a conditional input or changing their order would not change the probability of a proof. This is usually resolved by creating subcategories of rules such as parent annotation or lexicalization. However, this would significantly increase data sparsity because of the blowup of rules, especially in real-life scenarios when conditional input set I and ruleset A are large.

By introducing latent state to each variable, we could model such dependency structure conveniently saving the effort of manually creating rules and reducing the data sparsity. We can capture the unobserved statistical correlation of given data that can be explained by latent states. A prominent adaptation is probabilistic context-free grammars with latent states (L-PCFGs), by giving each non-terminal symbol a number of latent states [22].

The weighted logic programmes depend on the transition probabilities as their parameters. Such probabilities are trained by learning from golden proofs as to its data. The second key question is that training the model is also a non-trivial task. A straightforward method is a bootstrapping algorithm Expectation Maximisation(EM). However, this simple method still has limitations that need to use plenty of data going through many iterations, to minimise a loss function that cannot guarantee to reach global minimal. By contrast, the Spectral Learning method can be applied to enhance learning efficiency in many general fields, unsupervised, semi- or supervised learning [18].

A successful and efficient algorithm *Spectral Learning of Latent-Variable PCFGs* is proposed [7], which provides a strong theoretical foundation on applying latent variables and spectral learning methods in weighted logic programming. However, currently, this algorithm is designed only for context-free grammar, the proof tree in this domain strictly follows Chomsky Normal Form(CNF), having a single non-terminal parent and 2 non-terminal children. In other scenarios, more complex relationships exist between axioms in proofs, which often contains multi-parent and multi-children, such as Bayes Network. The goal of this dissertation is to extend the algorithm in italic above to handle more general weighted logic programmes.

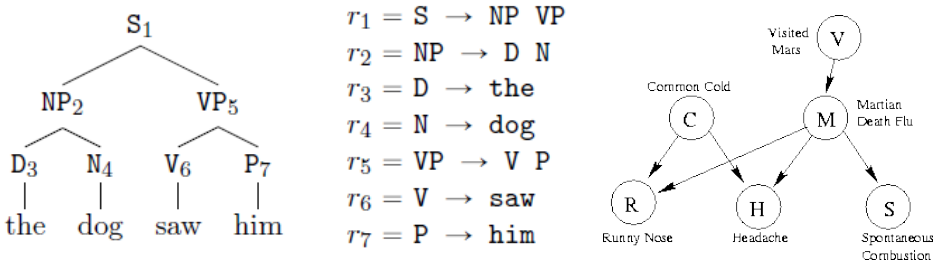


Figure 1.2: Context-Free Grammar (Left) Bayes Network(Right)

In this dissertation, our extension of the algorithm is based on 2 results:

- *Type-specify reified programme* We've provided a formalism that could handle multiple types of variables in a single proof. We cast a variable as an instantiation of an axiom. We also reified certain variables into parameters that do not directly participate in the calculation.
- *Variable interaction graph* We've investigated the interactions between variables as parents of an axiom. We created a connectivity graph to model such interaction and thus create a corresponding feature matrix for each left-hand side of the axiom.

This dissertation derives a more general algorithm that can handle multi-parent and multi-children axioms with different types of variables, which is a relaxed scenario from CNF. We implemented and tested our algorithm under the single-parent domain against the Projected Gradient Descent(PGD) and EM algorithm. Spectral learning shows a very acceptable Micro F1 score of 0.81 with a variance of 0.0003 within 20.57 minutes of running time. The algorithm is highly robust and efficient compared with other algorithms that have larger variance and longer running time ($>$300min).

The result verifies what Cohen showed that a latent-variable programme can be learnt using spectral learning which is highly efficient and has a PAC-style guarantees [7].

Chapter 2

Background

2.1 Weighted Logic Programming

Weighted logic programming (WLP) is a generalization of bottom-up logic programming where each proof is assigned with a numerical weight, which is the sum of weights of every axiom used in the proof.[6] Each provable item has a numerical value. The numerical values are commonly interpreted as probabilities.

1. I, a set of conditional inputs, encoded as axioms.
2. A, a set of axioms known to be true, the axioms we use in proofs.
3. P, a deductive proof of the goal theorem using axioms.

Consider we have the following example WLP, a weighted finite state machine, which satisfied the definition above by having the following components:

- $I = initial(q_0)$.
- $A = edge(q_0,q_1), edge(q_0,q_2), edge(q_1,q_1)..., edge(q_2,q_3)$.
- $goal = reach(q_0,q_3)$.
- $P = reach(q_0,q_3) \oplus = initial(q_0) \otimes edge(q_0,q_1) \otimes edge(q_1,q_3)$.

This proof describes a valid path from start state q_0 to accepting state q_3. In the proof we have 2 operators, $\oplus =$ **aggregation operator** and $\otimes$ **semiring product operator**. A valuation function $val[]$ always exists in WLP that returns the numerical value of a variable, sometimes it is omitted for simplicity. Depends on the shape of data it's working with, it might have different realizations [11]. However, in the current case, we can interpret the proof as below:

$$\begin{aligned} P &= val[initial(q_0)] \times val[edge(q_0,q_1)] \times val[edge(q1,q3)] \\ &= 1 * 0.7 * 0.5 = 0.35. \end{aligned} \tag{2.1}$$

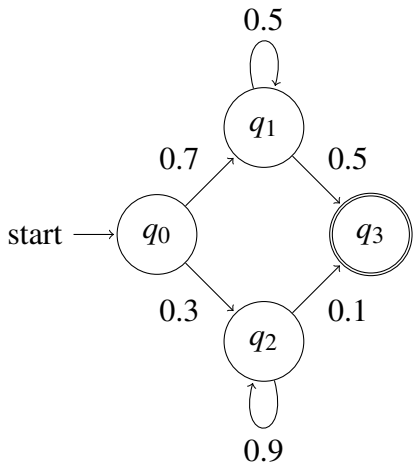


Figure 2.1: An example of the weighted logic programme

2.2 Latent Variable

Latent variables are variables that cannot be directly observed but mathematically inferred from other observed variables. The relationship between such latent variables and observable variables are what we're interested. A very naive understanding could be clustering task in unsupervised learning. How many clusters and which cluster an observable instance belongs to is the latent variables we are investigating. They are many examples for such applications, for example the alignment between source and target text in machine translation, or part-of-speech tagging in a hidden Markov model.

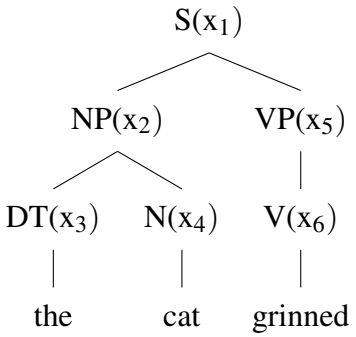


Figure 2.2: Example of L-PCFGs tree, where x_i represents the latent state of a non-terminal symbol

It can bring many intrinsic or extrinsic benefits by introducing latent variables in many traditional models, for example, increasing model accuracy, reducing the dimension of data and reducing effort in manual feature selection. However, it's highly probable that we cannot gasp the true hidden structure and correlation behind observed values. In this case, we manually choose a reasonable number for latent states and let the machine fits the data into the states automatically. A successful implementation is latent-variable

probabilistic context-free grammar, which increases parse tree prediction accuracy by adding latent states for each grammar symbol [22].

2.3 Spectral Learning Algorithm

Spectral learning is a very efficient algorithm compared with traditional estimation algorithms (e.g. Expectation-Maximization). When we are dealing with proof trees and natural language processing, we often suffer from data sparsity. The algorithm projects high dimensional sparse data into lower dimensions by using singular value decomposition.

$$\Omega = \mathcal{U}\Sigma\mathcal{V}^{\top}. \tag{2.2}$$

SVD can be perform on any matrix $\Omega \in R^{m*n}$ to result in $\mathcal{U} \in R^{m*m}$ as left singular vectors, $\mathcal{V} \in R^{n*n}$ as right singluar vectors, and a diagonal matrix $\Sigma \in R^{m*n}$ containing all singular values.

$$\Omega \overset{\text{truncated}}{\approx} \mathcal{U}_k\Sigma_k\mathcal{V}_k^{\top}. \tag{2.3}$$

In a real implementation, we often use truncated SVD that chooses k much smaller than $rank(\Omega)$ to compute the largest k singular values to approximate the original matrix in order to reduce the dimension of data. We consider the best choice of k is the one that generates the matrix with the closest Frobenius norm to the original matrix [10].

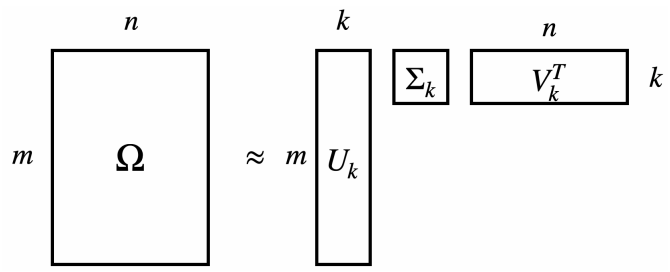


Figure 2.3: Visualisation of Truncated SVD

A typical routine of learning is to extract features of training data into such matrix and choose a small k, then use the generated left singular and right singular vectors to project original data into lower dimensions. Combining the concept of latent variables, we can think of this as the entire training data falls into the categories of groups of latent variables, then we only consider the groups' representation.

2.4 Related Work

Bayesian Network and its derivations are important models that are used in probabilistic modeling. Since each node could represent a random variable and each edge

represents the conditional probability for the corresponding variable. However, such representations are limited to non-recursive rules and linear relationships [1]. Dynamic Bayesian Network(DBN) could express the relationship between random variables over adjacent time steps. Such temporal structure could be seen as an introduction of recursive rules [17].

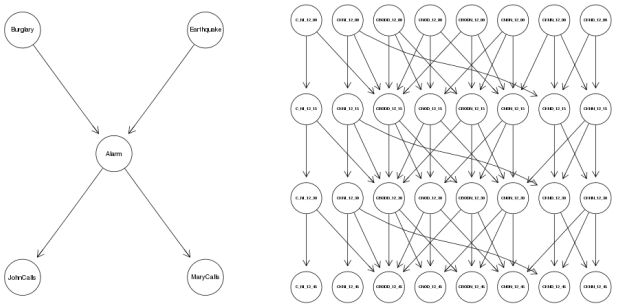


Figure 2.4: Simple Bayes Network (Left) Dynamic Bayes Network (Right)

Furthermore, Probabilistic Boolean Network(PBN) [27] is conceptually related to DBN but more flexible in adding more recursive rules between variables and modelling their dependencies. PBN shares the feature of rule-based Boolean Networks, robust to handle uncertainty. The dynamics of such a network could be studied as Markov Chains.

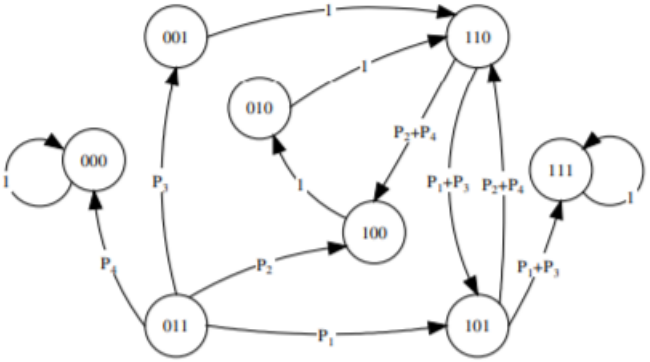


Figure 2.5: Probabilistic Boolean Network

Compared to the models described above, our proposed model focuses more on latent state representations behind each variable. In addition, we adapted recursive rules and modelled the dependency between multi-parent and multi-children of a given variable. Instead of paying attention to the entire network, we emphasize a path that could lead us from given inputs to a goal variable. For example, consider the path from state ”000” to ”110” in the network above. There are many possible paths with different transition probabilities as parameters. Given many proofs followed by this distribution, we are learning the parameters and thus be able to reconstruct such networks.

Conducting probabilistic inference, finding proof to the goal, remains an open problem. It’s a very complex issue considering real-life scenarios when variables are under

different domains (different types) and different forms of distribution (continuous, discrete, etc.). Hybrid logic programming has provided an inference algorithm in solving such issues [24] from a logical perspective. Minervini et al. introduced a Conditional Theorem Prover (CTP) for end-to-end differential proving that's more suitable in graphical perspective. The CTP could prove and learn about logic rules over small graphs with high time-complexity using the gradient-descent based method. In this dissertation, we only focus on learning discrete distributions of transition parameters with larger and longer proofs in a more efficient way which is spectral learning.

Chapter 3

Weighted Logic Programming with Latent Variables

3.1 Definition

Weighted Logic Programming (WLP) is bottom-up programming, where proof is assigned a value as its weight by processing the weight of axioms used in the proof. Each provable item has a value. On top of that, Latent Variable Weighted Logic Programming (L-WLP) extends WLP by also assigning multiple latent states to variables.

Similar to WLP, L-WLP also has 3 main components as below these 3 major components below:

1. I, a set of conditional inputs, encoded as axioms.
2. A, a set of axioms known to be true, for us to use in proofs.
3. P, a deductive bottom-up proof of the goal theorem using axioms.

In order to perform calculations, we define 2 operators below:

1. $\oplus =$, an aggregation operator requires a sum over all ways of grounding the variables that appear only in the rule body.
2. $\otimes$, an binary, associative operator that used as semiring product[1]

We also define 3 types of variables:

1. Observed input as $i, i \in [0,1]$, describing the probability of observing this input. This is usually derived by counting the relative frequency.
2. Simple variable types as $s, s \in R^{h_s} * 1$, usually a row vector with each of its entries as the probability distribution of a latent state from h_s. We use superscript to identify different variables, and subscripts to identify its latent states s_h^p.

[1] An algebraic semiring consists of five elements $< \mathcal{K}, \oplus, \otimes, 0, 1 >$ where $\mathcal{K}$ is a domain closed under $\oplus$ and $\otimes$. $\oplus$ is a binary, associative, commutative operator, $\otimes$ is a binary, associative operator that distributes over $\oplus$, $0 \in \mathcal{K}$ is the $\oplus$ identity, and $1 \in \mathcal{K}$ the $\otimes$ identity [6] [11].

3. Rule variable as T, given a rule $r^1 = s^p \rightarrow s^{c_1}, ..., s^{c_n}$, we write $T^{r^1} = T^{s^p \rightarrow s^{c_1}, ..., s^{c_n}} \in R^{h_{s^1} \times ... \times h_{s^n} \times h_{s^p}}$ which is a tensor indicating the transition probability using this rule over a inference relationship.

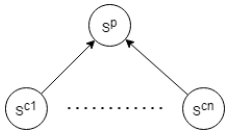


Figure 3.1: A tree representation for inference relationship

Inference (transition) relationships among variables are defined in set A, rule variables are used to ground such relationships. The figure above indicates the variable s^p is a parent variable concluded by many children $s^c_1, ..., s^c_n$. Similarly, s^p could be seen as a child if it could infer other variables, and its children can also be parents concluding from other variables as well.

In addition, we recognize axioms as predicates, this means an axiom will always take a fixed number of inputs, and the arguments are in strict order, just as when we are treating functions in programming. For example, $Rule1(A,B) \neq Rule1(B,A)$, if $Rule1(args1, args2)$ is not symmetric.

> You see a woman ('Beth') is taking a kid('Morty') to school, and then she went to a hospital to look after an old man('Rick') who looks similar to the women. What's the relationship between the kid and the old man?

Then you have the following component for calculation,

- I(observed facts)= $\{woman(Beth), kid(Morty), old(Rick),$ $take2school(Beth, Morty), lookafter(Beth, Rick)\}$
- A(inference rules) $\{r_1 = grandparent(X,Y) \leftarrow parent(X,Z)\ \&\ parent(Z,Y),$ $r_2 = parent(X,Y) \leftarrow take2school(X,Y),$ $r_3 = parent(X,Y) \leftarrow lookafter(Y,X)\ \&\ old(X)\}$
- One valid proof P can be expanded as

$$grandparent(Rick, Morty)\oplus = parent(Beth, Morty) \otimes parent(Rick, Beth) \otimes T^{r_1}$$

$$parent(Beth, Morty)\oplus = take2school(Beth, Morty)$$

$$parent(Rick, Beth)\oplus = lookafter(Beth, Rick) \times old(Rick)$$

Note that we use $\times$ in the last proof step because the 2 variables are observed and does not have a latent state, and they do not need to use any rule.

In both examples above, we can see that a variable might be something we directly observed or inferred through other variables. We also notice that a variable is an instantiation of an axiom. For example, we have $a^1 = S(start, end)$ in syntax tree parsing, $a^2 = parent(X,Y)$ in example above, as axioms, and $v^1 = S(0,3), v^2 = parent(Beth, Morty)$ are variables instantiated from corresponding axioms. Reversibly, we can say a variable v^2 has a specific type $a^2 = parent(X,Y)$. Keywords like $Beth$, $(0,3)$ are parameters that do not participate in calculations directly.

As for a rule axiom, it does not need any instantiation, since it's a general rule regardless of any specific variables inferring with, but only care about the types of variables and if the parameters can be unified with the rule correctly.

Remark 1 *In this section, we only discuss inference rules with a single variable as a parent on the left-hand side, we will discuss more complex scenarios (multi-parent) in later sections.*

Thus, we correspondingly define 3 parameters for us to evaluate the probability:

1. $\pi(s, h_s)$ The probability of a variable being the root of our bottom-up proof tree, which means that variable is the goal of our proof.
2. $t(s^{c_1}, ..., s^{c_n}, h_{c1}, ...h_{cn} | s^p, h_{s^p})$ The probability of concluding a parent s^p at latent state h_{sp} with multiple children and their corresponding latent states, which is the transition probability of inference rules in A
3. $o(i_1, ...i_j | s, h_s)$ The probability of observing a sequence of conditional inputs $\{i_1, ..., i_j\} \subseteq I$ as the corresponding latent state of observable simple variable s.

Also we define the proof P as a directed acyclic graph (DAG) tree $P = \{\mathcal{V}, \mathcal{E}\}$ variables as vertices, and a single root node s_{root}. The edges[2] between nodes indicates inference relationships derived from set A.

3.2 Core Computation

If we go through the family of L-WLP, for example, HMM, L-PCFGs or Bayes Network with latent states, we are required to compute the probability of proof given a sequence of conditional inputs $(i_1, ..., i_n)$ and a fixed set of rules A, then estimate the probability of this sequence. Intuitively, we know that this is a process that requires dynamic programming techniques such as the CYK algorithm since it's highly likely for us to derive multiple valid proofs using different rule combinations. It's true that L-PCFGs and other L-WLP might have different objectives, (e.g. finding proof with maximum probability afterwards), however, they still share this common calculation as below.

1. Given I, A,a single P, and parameters (π, t, o) calculate the probability of P.
2. Given a proof $P = \{\mathcal{V}, \mathcal{E}\}$, for all $v \in \mathcal{V}$, calculate the marginal probability of $p(v), s = type(v)$.

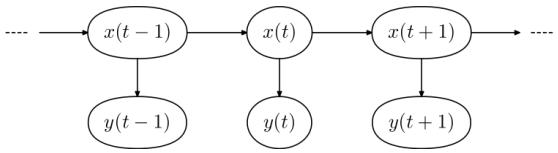


Figure 3.2: Example of hidden Markov model

[2]Instead of edge between 2 vertices, our edge is defined as children are directed to a single parent, such as edge($s^p \rightarrow s^{c_1}, ..., s^{c_n}$) is describing n edges from children to parent.

Algorithm 1 Probability of a single proof P

Require: $I = i_1, ..., i_n, P = \{\mathcal{V}, \mathcal{E}\}$
  $\pi(s, h_s)$ for root of proof P
  $t(s^{c_1}, ..., s^{c_n}, h_{c1}, ...h_{cn}|s^p, h_{s^p})$ for all inference rule in A that are used in P
  $o(i_1, ..., i_j|s, h_s)$ for all conditional input rule in P
Ensure: Probability of P
  for $v \in \mathcal{V}, type(v) = s$ and exist $o(i_1, ..., i_j|s, h_s)$ **do**
    Initialize the variable v where $v_h = o(i_1, ..., i_j|s, h_s)$
  end for
  for $v \in \mathcal{V}, type(v) = s$ and exist $edge(s_p \rightarrow \_) \in \mathcal{E}$ **do**
    Initialize variable $v^p_{h_{sp}} = \sum_{h_{c1},...,h_{cn}} t(s^{c_1}, ..., s^{c_n}, h_{c1}, ...h_{cn}|s^p, h_{sp}) \prod_i v^{c_i}_{h_{ci}}$
  end for
  Return $\sum_h v^{root}_h \pi(s, h)$ where $type(v) = s$

Given a simple hidden Markov model as shown above. The model has only 1 type. We have observable variables $y(0), ..., y(n)$, and each of them has a single latent variable $x(n)$. Thus, given a sequence of observable input of $I = y(0), ...y(n)$, we compute parameter $o(y(n)|x(n))$, and for variables concluded, we have $Transition(x(n), x(n-1)) = t(x(n-1)|x(n)) * Transition(x(n-1), x(n-2))$. Summing over latent states can be ignored in this case, since we only have 1 latent variable per variable, thus the composite variable is just a number. Then the probability of this sequence (proof) under this model can be written as

$$\pi(x(0)) \prod_{i=2}^{n} t(x(i-1)|x(i)) \prod_{i=1}^{n} o(y(i)|x(i)) \tag{3.1}$$

This equation is the equivalent to the form in HMM algorithm [16], which verifies our algorithm for calculating the probability of proof in general L-WLP.

The marginal probability is calculated using an algorithm called *Inside-Outside Algorithm*, which is an analogous form of *Forward-Backward Algorithm* [20]. In general, given a sequence of conditional inputs and parameters, this algorithm has 2 parts. As shown in the figure below, the *Inside* part sums over the partial proofs from the current variable p to the bottom (variables directly observed). The *Outside* part sums over proofs to get to this variable p. Thus, multiplying them together would give the likelihood of this type variable p appearing in any proof derivation tree.

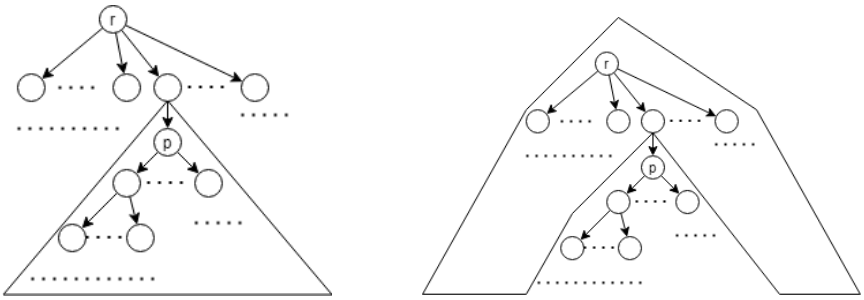


Figure 3.3: Inside tree (Left) and Outside Tree (Right) for variable p

Algorithm 2 Marginal Probability of a single variable v, $type(v) = s$

Require: I, A, P and all parameters π, t, o
Ensure: $p(v)$
Inside Base Case: ▷ when a variable is directly observed
Initialize $\alpha_h^s = o(i_1, ..., i_j | s, h_s)$
Inside Recursion: ▷ when a variable is concluded
Initialize $\alpha_{h_{sp}}^{s^p} = \sum_{parent} \sum_{h\, s^{c_1}} ... \sum_{h\, s^{cn}} t(s^{c_1}, ..., s^{c_n}, h_{c1}, ...h_{cn} | s^p, h_{sp}) \prod_i \alpha_{h\, s_i}^{s_i}$
The first summation means sums over all partial proofs with s_p as it's parent.
Outside Base Case: ▷ when a variable is the s_{root}
Initialize $\beta_h^s = \pi(s, h_s)$
Outside General Case: ▷ when a variable is used in inference
Initialize $\beta_{h_{sp}}^{s^p} = \sum_{child} \sum h_{s^p} ... \sum h_{s^{sibling_n}}$
$t(s^{sibling_1}, ..., s^p, ...s^{sibling_n}, h_{sibling_1}, ..., h_{s^p}, ..., h_{sibling_n} | s^{gp}, h_{s^{gp}}) \beta_{h_{s^{gp}}}^{s^{s^{gp}}} \prod_{i=1} \alpha_{h_{sibling_i}}^{s^{sibling_i}}$
This describes the process summing over all proofs until s^p will be reached as it's child (all possible ways to get to s^p from the root of the tree), and multiply with inside tree for sibling variables for s^p. s^{gp} indicates grandparent.
Return p(v) = $\alpha^s \beta^s$

Then it's obvious that in the training phase of L-WLP is to estimate these 3 parameters π, t and o for all **types** of variables. They are estimated by instantiations of the types and axioms we've mentioned in the definition. But before diving into this process, there are certain constraints that need to be satisfied.

Constraint 1 *Limit each inference relationship has only 1 parent variable. For example, in each proof or partial proof, the left-hand side equation contains a single variable.*

Constraint 2 *In each inference relationship, there is a unique transition axiom, a rule variable, T to represent the transition probability. For example, in each proof or partial proof equation, the right-hand side contains a term T, which has the dimension as the outer product of all variables included.*

Since proof P is a bottom-up tree, we split the vertices into 3 different parts, **root**, **nodes** and **leaves**. The root node has parameter $\pi(s, h) = \overline{\pi}_h^s$. Nodes are concluded which has parameter $t(s^{c_1}, ..., s^{c_n}, h_{c1}, ...h_{cn} | s^p, h_p) = T_{h_{c1}, ...h_{cn}, h_p}^{s^p \to s^{c_1}, ..., s^{cn}}$. The leaves are directly observed which has parameter $o(i_1, .., i_j | s, h) = \overline{o}_h^{s \to i_1, .., i_j}$. They are probabilities satisfying conditions below.

- $\sum_{i=1}^{h_s} \overline{\pi}_i^s = 1$
- $\sum_{p=1}^{h_{sp}} \sum_{c_1=1}^{h_{s^{c_1}}} \cdots \sum_{c_n=1}^{h_{s^{cn}}} T_{c_1, ...c_n, p}^{s^p \to s^{c_1}, ..., s^{cn}} = 1$
- $\sum_{k=1}^{h_s} \overline{o}_k^{s \to i_1, .., i_j} = 1$

Definition 1 *Assume we have a L-WLP proof tree with parameters initialized above, and for each variable s included in the proof, there is an invertible square matrix $G^s \in R^{h_s \times h_s}$. We define the following parameters:*

- *For every inference relationship of* $s^p \to s^{c_1}, ...s^{c_n}$,
  $C^{s^p \to s^{c_1},...s^{c_n}}(y^1,...,y^n) = T^{s^p \to s^{c_1},...s^{c_n}}(y^1 G^{s^{c_1}},...y^n G^{s^{c_n}})(G^{s^p})^{-1}$
  where $C^{s^p \to s^{c_1},...s^{c_n}}(y^1,...,y^n)_{hs^p} = \sum_{h\, s^{c_1}} ... \sum_{h\, s^{c_n}} C_{h_{s^p},h_{s^{c_1}},...,h_{s^{c_n}}} y^1_{h_{s^{c_1}}} ... y^n_{h_{s^{c_n}}}$ *a row vector. It's original form is a tensor* $C^{s^p \to s^{c_1},...s^{c_n}} \in R^{h_{s^p} \times h_{s^{c_1}} \times ... \times h_{s^c_n}}$.
- *For every directly observed variable* $s \to i_1, ..., i_j$,
  $c^{\infty}_{s \to i_1,...,i_j} = \overline{o}^{s \to i_1,...,i_j}(G^s)^{-1}$.
- *For variable s,*
  $c^1_s = G^s \overline{\pi}^s$.

These are extended definitions from L-PCFGs [7], it has been proved that when G is an identity matrix with the corresponding size, it can be ignored in the above equations. Based on these forms we can then derive the observable representation and then proceed to spectral learning.

We now assume 2 feature functions $\phi(In^v) \in R^d$ and $\psi(Out^v) \in R^{d'}$ that maps the inside tree and outside tree for a variable v into corresponding vectors. This action captures the dependency relations among itself, its parent concluding with siblings, and the children it's inferring from. Because the proof might have different lengths depending on the conditional input, and a goal could be proofed in many different ways. The inside and outside trees vary in size and height. Thus, introduce another constraint.

Constraint 3 *For a specific type s, among all the proofs, the feature vectors of its instantiated variable v must have the same size. This can be achieved by either only considering a limited range of trees or giving place-holder paddings.*

Condition 1 *For each variable v in the given proof tree, we can find matrices* $I^s \in R^{d_s \times h_s}$ *and* $J^s \in R^{d'_s \times h_s}$ *where* $s = type(v)$ *that must be full rank. In addition, there is a diagonal matrix* $\Gamma^s = diag(\gamma^s) \in R^{h_s \times h_s}$ *where each entry is the prior probability for each latent state* $\gamma^s_h = p(H = h|s)$. *H is the current state.*

Condition 2 *For each v in the given proof tree, we can find projection matrices* $U^s \in R^{d_s \times h_s}$ *and* $(V^s)^\top \in R^{d'_s \times h_s}$ *where* $s = type(v)$. *Such that we can derive corresponding invertible matrices* $G^s = (U^s)^\top I^s$ *and* $K^s = V^s J^s$.

These conditions are parallel to those proposed in the spectral learning algorithm for L-PCFGs [7]. It's necessary to satisfy them to compute the correct representations below.

- $I^s_h = \mathcal{E}[\phi(In^s)|H = h]$
- $J^s_h = \mathcal{E}[\psi(Out^s)|H = h]$

In consequence, we are facing very sparse data. The sparsity grows exponentially with valid rules involving the current variable type s. Here the spectral algorithm comes into play, the dimension of such vector is reduced by applying linear transformation using projection matrices. Thus for an inference rule, we can calculate the following

- $Y^{s^p} = U^{s^p} \phi(In^{s^p})$
- $Z^{s^p} = (V^{s^p})^\top \psi(Out^{s^p})$

- $Y^{s^c} = U^{s^c}\phi(In^{s^c})$ for all children of s^p

Theorem 3.2.1 *Assume condition 1 holds, we can derive projection matrix for each node(variable) in the proof tree by using T-SVD(Truncated Singular Value Decomposition) on feature matrix* Ω^s

$$\Omega^s = \mathcal{E}[\phi(In_s)\psi(Out_s)^\top] = U^s\Sigma^s(V^s)^\top$$

3.3 Observable Representations

With the facts and constraints we have defined above, we can then give the observable representation of the program.

- $\Sigma^s = \mathcal{E}[Y^s(Z^s)^\top]$
- $D^{s^p \to s^{c_1},...s^{cn}} = \mathcal{E}[Z^{s^p}Y^{s^{c_1}}...Y^{s^{cn}}]$ for nodes s_p concluded
- $d^{s\to i} = \mathcal{E}[(Z^s)^\top]$ for leaves s directly observed

In addition to these basic representations, by applying the transformation in definition 1 and with access to feature functions, we can also derive the following observable representations.

- $C^{s_p\to s_{c1},...s_{cn}}(y^1,...,y^n) = D^{s_p\to s_{c1},...s_{cn}}(y^1G^{s_{c_1}},...y^nG^{s_{c_n}})(\Sigma_p^s)^{-1}$
- $c^\infty_{s\to i} = d^{s\to i}(\Sigma^s)^{-1}$
- $c^1_s = \mathcal{E}[Y^s|root(s)]$ we only count Y^s if s is the root of a proof.

3.4 Deriving Empirical Estimates

Now consider we have given an L-WLP program with a set of conditional input set I, a set of inference rules A, and also M fully-expanded proof trees as training examples $\{P^1,...,P^M\}$.

Then for every variable type s involved in the tree, we can extract its feature matrix Ω^s. One thing to notice is that some variables may be used multiple times in a proof, thus we can obtain multiple feature matrices in this single tree. Then we perform the same process for every tree to get the expected value.

Algorithm 3 T-SVD Decomposition for variable type s

Require: $\{P^1,...,P^M\}, P^i = \{\mathcal{V}^i,\mathcal{E}^i\}, V^i = v^1,...,v^N, I, A, s$

Initialize the normalized feature matrix

$$\hat{\Omega}^s = \frac{\sum_{i=1}^M\sum_{j=1}^N[|type(v^j)=s|]\phi(In^{v^j})\psi(Out^{v^j})^\top}{\sum_{i=1}^M\sum_{j=1}^N[|type(v^j)=s|]}$$

Return $\hat{\Omega}^s = \hat{U}^s\hat{S}^s(\hat{V}^s)^\top$, $\hat{U}^s \in R^{d\times h_s}$ is the left singular vectors carrying the h_s dimensional embedding for inside features, and $\hat{V}^s \in R^{h_s\times d'}$ is right singular vectors for outside features. We also define $\hat{\Sigma}^s = (\hat{U}^s)^\top\hat{\Omega}^s\hat{V}^s$(?).

Algorithm 4 Spectral Algorithm for single parent L-WLP

Require: $\{P^1,...,P^M\}, P^i = \{\mathcal{V}^i, \mathcal{E}^i\}$, $\mathcal{V}^i = \{v^1,...,v^J\}, I$
$A = \{r^1,...,r^N\}, N << M, r^i = s^p \to s^{c_1},...,s^{c_n}$
(Step 0: T-SVD Decomposition)

- For each variable type s, initialize $\hat{U}^s$, $\hat{V}^s$ and $\hat{\Sigma}^s$

(Step 1: Projecting to Lower Dimensions)

- For $i \in [M]$, for all $v \in V^i$, compute

$$y^{(i,v^p)} = (\hat{U}^{s^p})^\top \phi(In^{(i,v^p)})$$

, where $type(v) = s^p$. $In^{(i,v^p)}$ means the inside tree of variable v^p in proof tree P^i. ▷ Projecting a parent variable v's inside feature into lower dimensions.

- For $i \in [M]$, if a variable v^p is concluded by children $v^{c_1},...v^{c_n}$ and unifies with a rule $r \in A$ such that $r = s^p \to s^{c_1},...,s^{c_n}$. Then for each child, compute

$$y^{(i,v^{c_k})} = (\hat{U}^{s^{c_k}})^\top \phi(In^{(i,v^{c_k})})$$

,where $j \in [n]$, $type(v^{c_k}) = s^{c_k}$. ▷ Projecting children's inside feature.

- For $i \in [M]$, compute

$$z^{(i,v^p)} = (\hat{V}^{s^p})\psi(Out^{(i,v^p)})$$

,where $type(v^p) = s^p$. ▷ Projecting outside feature

(Step 2: Calculate Correlations)

- For all variable s, normalizing coefficient

$$\delta^s = \frac{1}{\sum_{i=1}^{M}\sum_{j=1}^{J}[|type(v^j) = s|]}$$

▷ Normalizing by frequency of type s over all proofs

- For each rule in A,

$$\hat{D}^{s^p \to s^{c_1},...s^{c_n}} = \delta^{s^p} \sum_{i=1}^{M} (z^{(i,v^p)}) \prod_{j=1}^{J} (y^{(i,v^{c_j})})^\top$$

, where $v^p \to v^{c_1},...,v^{c_j}$ unifies with $s^p \to s^{c_1},...,s^{c_n}$.

- For each s directly observed,

$$d^{s \to i} = \delta^s \sum_{i=1}^{M} (z^{(i,v)})^\top$$

, where $type(v) = s$.

(Step 3: Compute Final Parameters)

- $\hat{C}^{s^p \to s^{c_1},...s^{c_n}}(y^1,...,y^n) = D^{s^p \to s^{c_1},...s^{c_n}}(y^1,...y^n)(\Sigma^{s_p})^{-1}$
- $\hat{c}^{\infty}_{s \to i_1,...,i_j} = d^{s \to i_1,...,i_j}(\Sigma^s)^{-1}$
- $\hat{c}^1_s = \frac{\sum_{i=1}^{M}[|s_i = s \;\&\; root(s_i)|]y^i}{\sum_{i=1}^{M}[|root(s_i)|]}$

3.5 Multi-parent resolution

The previous chapter extends the algorithm from Cohen et .al [7] by reifying predicates and assigning different types to each variable. However, as mentioned earlier, the previous algorithm has only considered CNF where a rule contains at most 1 parent and 2 children. Multiple children are an easy extension, but the multi-parent case would be more ambiguous, considering the different joint probability of latent states. Here, we provide a *Variable Interaction Graph* to capture the dependencies between parent variables on the left-hand side of rules.

Definition 2 *The variable interaction graph $G = \{\mathcal{V}, \mathcal{E}\}$ such that a variable $p, q \in \mathcal{V}, (p,q) \in \mathcal{E}$ if and only if both p and q appear together in the same proof in the left hand side.*

Inside such graph, it includes all the possible variables that act as parents on the left-hand side. Then we find all connected components of G by calling them $C_1 ... C_K$. There are no edges between distinct among any connected components. We will expand the feature matrix $\Omega = \mathcal{E}[\phi'\psi'^{\top}|C_j]$ based on different component. For each parent in a component, we construct 2 vectors ϕ' and ψ' accordingly.

Definition 3 *For a component with j items, $C_j = s^1 ... s^j$, we define $\phi' = (\phi_{s^1}||...||\phi_{s^n})$ and $\psi' = (\psi_{s^1}||...||\psi_{s^n})$, $||$ means concatenation. We are concatenating feature vectors of all variables in a component, such that ϕ_s and ψ_s are allowed to check the relevant proof if and only if s exist in the current proof, otherwise ϕ_s and ψ_s are zero vector.*

To understand this more clearly, we provide a toy example below. Consider we have the following rule system and a large enough data set containing sufficient proofs as below. The right-hand side of inference rules is omitted for convenience.

- $A = \{r_1 = A, B \rightarrow ...; r_2 = B, C \rightarrow ...; r_3 = A \rightarrow ...; r_4 = B \rightarrow ...; r_5 = D \rightarrow ...\}$

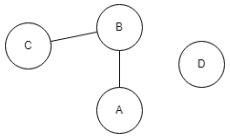


Figure 3.4: Variable Interaction Graph Example

Based on the rules, we have constructed the variable interaction graph as defined. We could easily observe that there are 2 components, $C_1 = A, B, C$ and $C_2 = D$. Now suppose we have in total M training examples, and currently focus on rule $r_1 = A, B \rightarrow$ for its transition probability tensor. Recall that $\phi(In) \in R^d, \psi(Out) \in R^{d'}$. We obtain the feature vector $\phi' = (\phi_A||\phi_B||\phi_C) \in R^{3d}$, same for ψ'. For both variable A and B their feature vectors follow this formalism but with different emphasis. By following the algorithm below. As we can easily see ϕ_C would be zero vector for both variables since in the current rule C does not participate in the calculation.

This algorithm then can be used in algorithm 3 for T-SVD decomposition. Even though the dimension of the feature matrix has changed, we are still able to obtain the correct projection matrix for a variable. For example, if the original projection matrix in a

Algorithm 5 Calculating feature matrix Ω for parents in multi-parent rule

Require: $P^1 ... P^M$, $r = s^{p_1} ... s^{p_n} \rightarrow s^{c_1} ... s^{c_m}, \{s^{p_1} ... s^{p_n}\} \subseteq C$
  for i in $1, ..., n$ **do**
    $\phi'_{s^{p_i}} = [], \psi'_{s^{p_i}} = []$
    for c in C **do**
      $\phi_c = \overline{0} \in R^d, \psi c = \overline{0} \in R^{d'}, N = 0$
      for c, s^i both occur as parent through $P^1 ... P^M$ **do**
        $\phi_c += \phi(In^c), \psi c += \psi(Out^c), N += 1$
      end for
      $\phi'_{s^{p_i}} = \phi'_{s^{p_i}} || \frac{1}{N} \phi_c, \psi'_{s^{p_i}} = \psi'_{s^{p_i}} || \frac{1}{N} \psi_c$ ▷ concatenation after normalization
    end for
    $\Omega^{s^{p_i}} = \phi'_{s^{p_i}} {\psi'_{s^{p_i}}}^\top$
  end for
  return $\Omega^{s^{p_1}} ... \Omega^{s^{p_n}}$

single parent case is $U \in R^d$, the new matrix would be $U \in R^{(d*\varepsilon) \times m}$, where ε is the number of variables in the current variable's connected component. Consequently, we could use algorithm 4 to calculate the transition tensor parameters with a minimal extension below. The correctness of the calculation relies on the proof in the appendix.

- $\hat{C}^{s^{p_1} ... s^{p_n} \rightarrow s^{c_1}, ... s^{c_m}}(y^1, ..., y^m) = D^{s^p \rightarrow s^{c_1}, ... s^{c_n}}(y^1, ... y^n)(\Sigma^{s^{p_1}})^{-1} ... (\Sigma^{s^{p_n}})^{-1}$
- $\hat{c}^{\infty}_{s^{p_1} ... s^{p_n} \rightarrow i_1, ..., i_j} = d^{s^{p_1} ... s^{p_n} \rightarrow i_1, ..., i_j}(\Sigma^{s^{p_1}})^{-1} ... (\Sigma^{s^{p_n}})^{-1}$
- $\hat{c}^{1}_{s^{p_1} ... s^{p_n}} = \frac{\sum_{i=1}^{M} [|s_i = s^{p_1} ... s^{p_n} \ \& \ root(s_i)|] y^i}{\sum_{i=1}^{M} [|root(s_i)|]}$

Chapter 4

Experiment Setup

The experiment aims to evaluate our Spectral algorithm by comparing it with the traditional Expected Maximization(EM). However, considering EM is becoming out-of-fashion, we also include more recent algorithm Projected Gradient Descent(PGD) for comparison. Because L-WLP is a proposed model and there is no off-the-shelf data and algorithm, it's necessary to create a reliable data set and also modify the existing algorithm to fit our domain. The following implementation has been done for the evaluation.

- WLP generator
- Spectral Learning algorithm
- Approximate Search Evaluation algorithm
- Expected Maximisation algorithm
- Projected Gradient Descent algorithm

4.1 WLP generator

The algorithm is designed to solve a wide range of weighted logic programmes including different types of dynamic bayesian networks and other types of probabilistic inference problems under multiple domains. It's natural that each of the domains might have unique specifications and relations, for example, the number of parents and children, expected recursion depth and reusable axioms. Creating proper models for real-world problems is time-consuming and deviates from the theoretical perspective of this project. Thus, we choose to design a WLP program generator that we could set the number of axioms, recursive depth and distribution parameters of axioms to obtain proof trees with observable facts as our data.

We assume all reification and proper grouping has already been done when constructing the model. Then we denote each type of the variables as a combination of lower case letters from 26 alphabets, and the observable facts are combinations of upper case letters. Each of the nodes also contains a prefix and suffix for better visualization. The

suffix denotes the level of the tree in top-down order except for observable facts and their direct parents. The prefix denotes the index of the same type of variables on the same level.

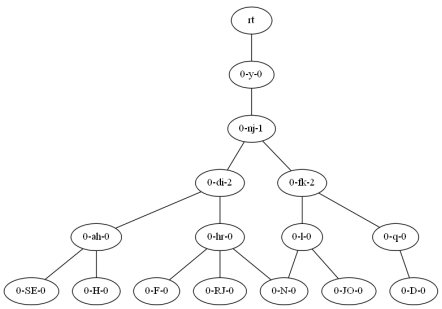


Figure 4.1: Example Proof Tree

The number of parents and children of a rule, max recursion depth and max tree depth are hyper-parameters for the generator that can be chosen freely. The parameters between rules are sampled by using Dirichlet Distribution to ensure it follows the constraints in chapter 3 and follows the definition of the probability distribution.

$$p(x) \propto \prod_{i=1}^{k} x_i^{a_i - 1}$$

Under this distribution, we could sample a vector $\bar{x}$ where $x_i > 0$ and $\sum_i x_i = 1$. For example, if we have a rule $A \to BC$ where each of them has arbitrary latent state h_A, h_B, h_C, we could construct a rule transition tensor $T \in \mathcal{R}^{h_A \times h_B \times h_C}$ which is calculated as the outer product of 3 Dirichlet distribution vectors.

To sample a full tree, we also need to set up prior probabilities of variables. For example, the probability of which set of variables is the root of a proof tree. For rules with the same set of parents, the prior probability of which rule is likely to be generated must sum up to 1. After constructing the tree skeleton, a different layer of pre-terminals and observable facts are sampled and attached separately under the same constraints.

The sampling iterative choose variables with the corresponding latent states by following the parameters described above. Then the latent state will be ignored, only the type of the variable will be kept as a training proof tree to preserve the desired ambiguity. In order to properly manage the tree and simplify further calculations, a tree type data structure is created. Variables sampled will be recorded as a **node** type object with **node.parent**, **node.children** and other attributes for convenience.

The final data will be presented as a single **node** object, the full tree can be retrieved by recursively calling **node.children** attribute. The object also has 2 functions that return the inside tree and outside tree of the current node within a designated range for feature extraction in the spectral learning algorithm.

To visualize a tree, a **draw_tree()** function is included by using **pydot** library. This function is only for demonstration purpose, in training and other computation, the only tree we use is the **node** object.

4.2 Spectral Learning Algorithm

The implementation of the Spectral Learning Algorithm follows entirely as the design in Chapter 3. However, some details need to be considered under specification. For example, the feature vector may vary from different problem domains. For example, in the realm of syntax tree parsing, feature vectors that contain contextual information are preferred [15]. One could also design a carefully engineered vector with interpretable features.

A naive design could be just using the one-hot encoding of adjacent variables in the proof tree. For example, to get an inside feature vector of a node in the proof tree, we directly concatenate all variables of its inside the tree and convert each of the variables into a one-hot vector. Assume we have k different types of variables in total, and the inside tree of the node has n variables, the final inside feature vector would be in shape $n * k \times 1$, same for outside vectors.

- $\phi(In^s) = [one\_hot(x) \text{ for } x \text{ in } In^s]$
- $\psi(Out^s) = [one\_hot(x) \text{ for } x \text{ in } Out^s]$

Bear in mind that tree size is growing exponentially with regard to the depth of a proof tree, it's crucial to understand the space complexity could easily be overwhelming if the depth and variable size of a WLP program are large. The sparse feature vector would have a size of $O(b^d * |A|)$ where b is the average branching factor, d is the depth of the tree and $|A|$ is the axiom set cardinality of a WLP. We also need to take the square of this bound because we need to calculate the feature matrix Ω for T-SVD decomposition.

During the experiment, feature collection and T-SVD constantly exceeds my RAM capacity of 32GB. This is fixed by introducing in-place calculation and setting up a max length for feature vectors. If a feature vector is longer than this maximum, we cut off the tail. If the length is smaller, we can simply pad it with zeros. The outside feature max length should be larger than the inside because usually, the outside tree of a variable is larger than the inside.

Another point to discuss is data sparsity. Since the rules, parameters and proof trees are automatically generated, not all rules are used in the limited artificial data set. It's normal to have rules that show up only a few times in sampled trees, or even have not been used at all. This causes 2 problems, zero probability and negative parameters. Zero probability is because we never met any instance in the training, so we cannot compute the feature matrix thus the parameter for the rule. This could be simply solved by using add-one smoothing or we just discard this rule from A. Because it's not the focus of the project how to model unseen rules.

As for negative parameters, this is caused by insufficient training examples or other sample errors of the algorithm [8]. The negative values occur in the outer product of inside and outside feature vectors after projection, which is the D and C tensor in the formula below.

$$\hat{C}^{s^p \to s^{c_1},\ldots s^{c_n}}(y^1,\ldots,y^n) = D^{s^p \to s^{c_1},\ldots s^{c_n}}(y^1,\ldots y^n)(\Sigma^{s_p})^{-1}$$

Consequently, the final marginal probability of a variable that is computed by our parameter might be negative. This would jeopardise the final parsing quality during evaluation. To fix this issue, Cohen et. al has proposed a solution of taking the absolute value as $argmax_t(\sum_{s \in t} |\mu(s)|)$ instead of $argmax_t(\sum_{s \in t} \mu(s))$ in the original Goodman's algorithm [14].

In this specific implementation, another post-processing method is used by directly taking the soft-max of the entire distribution parameter tensor. This also cancels out negative probabilities and clamp all values by following probability distribution into range (0,1] and sums up to 1.

It's still ambiguous if negative parameters carry a special meaning during our sampling. Consider if a very large negative value, it would be projected to a small value if there exist any other positive values in the parameter. However, by taking the absolute value, it would be a large positive value. We cannot conclude if the soft-max method is appropriate without doing further investigation. More theoretical and empirical work needs to be done for a better understanding of this issue.

4.3 Approximate Search Algorithm

One of the fundamental differences between Spectral Learning and other traditional learning algorithm is Spectral Learning is inference-free. During the learning process, it is not required to iteratively compute marginal probability by using the Inside-Outside algorithm. We are learning from the tree structure directly.

On the other hand, Gradient Descent and Expectation Maximisation have to find variables with the highest marginal probability by calling the Inside-Outside algorithm. During this process, we need to find all possible ways to prove an axiom or a variable p with the given subset of observable facts, or all possible ways to use current axioms in further proofs recursively. Such recursion can be extremely time-consuming considering the exponential size of the tree, which needs to be carefully handled. Under the domain of syntax tree parsing, we could use the famous CYK-algorithm. However, in other domains, this algorithm has its limitations.

$_0$a $_1$very $_2$heavy $_3$orange $_4$book $_5$

	1 *a*	2 *very*	3 *heavy*	4 *orange*	5 *book*
0 a	Det			NP	NP
1 very		Adv	AP	Nom	Nom
2 heavy			A,AP	Nom	Nom
3 orange				Nom,A,AP	Nom
4 book					Nom

Figure 4.2: Example of CYK algorithm

In the example above, indeed the CYK algorithm finds every possible provable constituent based on the given spanning in each entry of the matrix. However, such for-

mulation is grounded by the fact that words are given in temporal order and we only use the adjacent words as a possible spanning.

To put it more clearly, we say we can only use adjacent words as facts to prove a constituent is valid. Using {'heavy', 'orange'} to prove a *Nom* is valid in this case since these 2 words are adjacent. Using {'heavy', 'book'} to prove a *Nom*, in this case, is illegal, on the other hand. However, in other domains, a more flexible combination of axioms are allowed without considering adjacency. For example, if we have 3 separate facts {A,B,C}, we could not only use {A,B} but also {A,C}. This will result in a problem finding all legal subsets of usable axioms or facts. Then a number of possible combinations we need to iterate through increases from polynomial-size n^2 to exponential size 2^n.

Thus, it's not possible to iterate through all of them, especially when we introduced recursive rules such as the identity rule $s \rightarrow s$. Initially, a depth-first search algorithm is implemented as an attempt to solve this problem. However, the average running time is extremely high, and a maximum recursive depth and recursive timeout are set to skip through instances that are hard to compute.

1. **for** each axiom a, set $agenda[a] :=$ value of axiom a
2. **while** there is an item a with $agenda[a] \neq \mathbf{0}$
3. (* remove an item from the agenda and move its value to the chart *)
4. choose such an a
5. $\Delta := agenda[a]$; $agenda[a] := \mathbf{0}$
6. $old := chart[a]$; $chart[a] := chart[a] \oplus \Delta$
7. **if** $chart[a] \neq old$ (* only propagate actual changes *)
8. (* compute new resulting updates and place them on the agenda *)
9. **for** each inference rule "$c \oplus\!= a_1 \otimes a_2 \otimes \cdots \otimes a_k$"
10. **for** i from 1 to k
11. **for** each way of instantiating the rule's variables such that $a_i = a$
12. $$agenda[c] \oplus\!= \bigotimes_{j=1}^{k} \begin{cases} old & \text{if } j < i \text{ and } a_j = a \\ \Delta & \text{if } j = i \\ chart[a_j] & \text{otherwise} \end{cases}$$
(* can skip this line if any multiplicand is $\mathbf{0}$ *)

Figure 4.3: Weighted agenda-based deduction in a semiring [12]

By using an agenda-deduction algorithm, we could assign values for axioms we used inside a tree proof with the score. And based on this, we could perform an approximate search to find sub-optimal solutions. In this particular example, we use beam search to find limited proofs and then re-rank and pick the best of them.

4.4 Expectation Maximisation

Expectation Maximisation (EM) is a traditional algorithm to estimate parameters for the latent state model. The shortcoming of this method is it is time-consuming and suffers from local minima based on random initialization. We extended the EM algorithm from L-PCFGs parameter estimation to L-WLP based on Matsuzaki's [22] algorithm and Avneesh's [26] python implementation.

The cycle of EM has 3 important steps: Expectation Step, Parameter Update and Maximisation Step.

In the Expectation Step, we would like to estimate the total probability of each inside and outside tree of a variable. Inside an Expectation Step, we are updating the parameter based on previous parameters and correlation estimated in the current time step. In the maximisation step, we are normalizing the parameters based on the count of our re-estimation.

Algorithm 6 Expectation Maximisation for L-WLP

Require: rule $r^{(i)}$, inside tree $t^{(i_1)}...t^{(i_n)}$, outside tree $o^{(i_1)}...o^{(i_m)}$ and$b^{(i)} = 1$ if the rule is at the root of a tree, otherwise 0 for i in training examples $\{1...M\}$. Also the MAX_IT as the maximum iteration.

(Step-0: Parameter Initialisation)

for r in A **do**

$r = s^{p_1}...s^{p_m} \rightarrow s^{c_1}...s^{c_m}$

$\hat{C}^r \in R^{h_s^{c_1} \times ... \times h_s^{c_n}, h_s^{p_1} \times ... \times h_s^{p_m}}$

end for

$c^1_{s^{p_1}...s^{p_m}} = R^{h_s^{p_1} \times ... \times h_s^{p_m}}$

for iteration t in range of MAX_IT **do**

(Step-1: Expectation Step)

Estimate Y and Z. Compute the partial counts and total tree probability g using Inside-Outside algorithm for all t and o using $\hat{C}^r_{t-1}$

$\hat{E}^r = \frac{\prod_{i=1}^m Z(o^i) \prod_{i=1}^n Y(o^i)}{g}$

(Step-2: Update Parameters)

for r in A **do**

$\hat{C}^r_t = \hat{C}^r_{t-1} \odot \hat{E}^r_t$ ▷ $\odot$ is element-wise product

$c^1_{s^{p_1}...s^{p_m}} = c^1_{s^{p_1}...s^{p_m}} + b^{(i)} \frac{c^1_{s^{p_1}...s^{p_m}} \odot Y(r^i)}{g}$ ▷ update when current rule is the root

end for

(Step-3 Maximisation Step)

Normalize parameter with respect to the count of corresponding latent states

for r in A **do**

$\forall h_s^{c_1}, ..., h_s^{c_n}, h_s^{p_1}, ..., h_s^{p_m}$:

$\hat{C}^r(h_s^{c_1}, ..., h_s^{c_n}, h_s^{p_1}, ..., h_s^{p_m}) = \frac{\hat{C}^r(h_s^{c_1}, ..., h_s^{c_n}, h_s^{p_1}, ..., h_s^{p_m})}{\sum_{r'=r} \sum_{h_s^{c_1}, ..., h_s^{c_n}} \hat{C}^r(h_s^{c_1}, ..., h_s^{c_n}, h_s^{p_1}, ..., h_s^{p_m})}$

end for

end for

4.5 Projected Gradient Descent

In recent years, gradient descent based methods have become a mainstream algorithm for latent variable estimation. The idea behind gradient descent is still trying to tune the parameters so that we can maximize the probability of the given training proof trees. However, we need to be careful when using gradient descent based methods, especially when our parameters have a very specific domain. Under the scope of this project, the parameter for a rule must lie in $[0,1]$ and sums up to 1.

To ensure updated parameters satisfies this constraint, we considered 2 approaches

- Projection onto probability simplex
- Reparameterization

The routine and implementation of the gradient descent algorithm are very straightforward, thanks to the **autograd()** packages and PyTorch. As long as we store parameters into a tensor and turn on **require_grad** attribute for each tensor object, PyTorch automatically remembers the operations involving these tensors and create a graph. Once needed, we simply call the **backward()** on the final loss we computed, then gradients would be calculated automatically for corresponding tensors.

Gradient Descent Routine

1. Pick $\mathbf{x}_0 \in R^n$ as parameter
2. Loop until condition is matched
   (a) Calculate descent value as $-\Delta f(\mathbf{x}_t)$
   (b) Pick a step size α_t
   (c) Update parameter as $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \Delta f(\mathbf{x}_t)$

The steps above are the standard routine of gradient descent. In our experiment $\mathbf{x}$ would be any parameter tensors involved in a tree proof, and $f(\mathbf{x})$ would be the loss function as negative log-likelihood computed by using Inside-Outside algorithm. α is the learning rate. Details of PyTorch implementation follow the skeleton code provided in a similar context of compound probabilistic context-free grammar repository [19].

On top of the above procedure, we need to add an additional step to control the updated values that can still follow the constraint. Under the set of tensor in the real domain $\mathbf{x}_0 \in R^*$, where $*$ denotes the dimension of the parameter used in the corresponding rule, we define a subset Q contains all the tensors that satisfy the constraint.

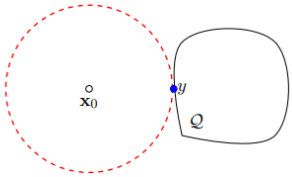


Figure 4.4: L2-norm ball [2]

Projected Gradient Descent

1. Pick $\mathbf{x}_0 \in Q$ as parameter
2. Loop until condition is matched
   (a) Calculate descent value as $-\Delta f(\mathbf{x}_t)$
   (b) Pick a step size α_t
   (c) Update parameter as $\mathbf{y}_{t+1} = \mathbf{x}_t - \alpha_t \Delta$
   (d) Projection $\mathbf{x}_{t+1} = \arg\min_{\mathbf{x} \in Q} \frac{1}{2} ||\mathbf{x} - \mathbf{y}_{t+1}||_2^2$

Projected gradient descent is a simple technique that if the value after gradient descent is leaving the legal value set of Q, project it back as its closest value in the legal domain. The closest value is on the boundary of the L2-norm ball touching set Q, and point always exist [2].

However, the routine above did not describe how could we sample $\mathbf{x} \in Q$. As we can see, we operate under a continuous space. A naive approach would be to sample vectors uniformly and discretely from the target space then calculate the argmin. Obviously, such process is time-consuming thus unacceptable. A better method is to project the illegal value to a probability simplex [28].

Algorithm 7 Euclidean Projection onto Probability Simplex

Require: $\mathbf{x} \in \mathcal{R}^D$
sort $\mathbf{x}$ into $\mathbf{u}$: $u_1 \geq ... \geq u_D$
Find $\rho = max\{1 \leq j \leq D : u_j + \frac{1}{j}(1 - \sum_{i=1}^{j} u_i) > 0\}$
Define $\lambda = \frac{1}{\rho}(1 - \sum_{i=1}^{\rho} u_i)$
return $\mathbf{y}$ s.t. $y_i = max\{x_i + \lambda, 0\}, i = 1, ..., D$

The above algorithm describes the process of projecting a vector with length D onto probability simplex. As for the parameter tensor, we could do this operation by flattening, projection then reshaping it back to its original view.

Another approach is using a trick taught in the MLPR course, which is reparameterization. Instead of using constrained values $\mathbf{x} \in Q$ as our parameter, we could convert this problem into an unconstrained version.

Reparameterization Routine

1. Pick $\mathbf{x}_0 \in R^n$ as parameter
2. Loop until condition is matched
   (a) Calculate descent value as $-\Delta f(softmax(\mathbf{x}_t))$
   (b) Pick a step size α_t
   (c) Update parameter as $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \Delta f(\mathbf{softmax(x)}_t)$

We simply use softmax to acquire the transition parameters as an intermediate value in the process of calculation. The parameter initialized this time is theoretically unconstrained. To calculate the final gradient and get an update we could just simply apply the chain rule.

But practically speaking, we still want our parameters and gradients to be in a proper range. This is because exponential values could easily cause overflow in floating-point calculations. For example, the largest value that a float64 number can store in python is about e^{300}. However, there are lots of values that could be selected as initial parameters. Thus, we need to use a numerically stable softmax and clip the gradients to make all values sensible.

Chapter 5

Evaluation and Result Analysis

To verify our algorithm and understanding, we've set up each component with the following parameters:

- **WLP Generator**: A single parent for each rule, maximum of 5 latent states for each type of variable, 20 axioms that involve variables only, largest branching factor per level is 5 and at most 40 different observable facts and recursion allowed. In the worst case, this will create $5^5 = 3125$ variables in a tree.
- **Spectral Algorithm**: Maximum inside feature number of 1000, maximum outside feature number of 2000. We tested 2 versions, one using softmax to process negative parameters, the other is taking the absolute value of marginal probabilities.
- **Expectation Maximization Algorithm**: Initialize all variables with 5 latent states, and run 1000 epochs.
- **Projected Gradient Descent**: Initialize all variables with 5 latent states, and run 1000 epochs with a learning rate of 0.01. We tested 2 versions, one using projection onto probability simplex, the other is using reparameterization using log and softmax.
- **Data Split** We've sampled a total of 1000 proof trees out from 5 different programs generated and split them into 8:1:1 as training, validation and testing sets

For result evaluation, we focus on 3 metrics, micro-F1, macro-F1 and time elapsed in training for each algorithm. Similar to the evaluation of parse trees, the F1 score is calculated as the proportion of correct variables in golden and predicted proof trees as precision and recall as shown in the above equations.

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

$$Precision_{micro} = \frac{TP_1 + ... + TP_n}{TP_1 + ... + TP_n + FP_1 + ... + FP_n}$$

$$Recall_{micro} = \frac{TP_1 + ... + TP_n}{TP_1 + ... + TP_n + FN_1 + ... + FN_n}$$

$$Precision_{macro} = \frac{Precision_1 + ... + Precision_n}{n}$$
$$Recall_{macro} = \frac{Recall_1 + ... + Recall_n}{n}$$

Algorithm	data-0	data-1	data-2	data-3	data-4	Avg	Var
EM	0.60	0.52	0.68	0.74	0.53	0.61	0.0074
Spectral-softmax	0.65	0.71	0.57	0.56	0.65	0.63	0.0031
Spectral-abs	0.78	0.80	0.83	0.82	0.80	0.81	**0.0003**
Pgd-reparam	0.78	0.80	0.83	0.82	0.79	0.80	0.0004
Pgd-simplex	0.81	0.83	0.87	0.85	0.79	**0.83**	0.0008

Table 5.1: Micro Avg F1 score for 5 algorithms in Single-Parent Programs

Algorithm	data-0	data-1	data-2	data-3	data-4	Avg	Var
EM	0.58	0.52	0.63	0.65	0.54	0.58	0.0026
Spectral-softmax	0.63	0.68	0.59	0.56	0.59	0.61	0.0017
Spectral-abs	0.68	0.73	0.79	0.78	0.74	0.75	0.0014
Pgd-reparam	0.68	0.73	0.79	0.78	0.74	0.74	0.0015
Pgd-simplex	0.76	0.80	0.82	0.82	0.74	**0.79**	**0.0011**

Table 5.2: Macro Average F1 score for 5 algorithms in Single-Parent Programs

As we can see from the tables, the F1 score of Micro Avg is always higher than Macro average. This can be explained by the variable size of trees we've sampled. Some trees can be very small with only a few variables, but others could have hundreds. If a tree has a large branching factor, then the rule used has fewer constraints (by other variables), then we have more rules to select. Thus, it's more likely to go wrong and have a lower precision and recall per tree. However this phenomenon is alleviated by taking the TP and FP per tree, larger trees with more items correct could balance this out.

Across algorithms, we can see PGD-simplex has the best F1 score in both Micro avg and Marco avg metrics. But our Spectral-abs algorithm achieves a slightly lower but similar performance. The result could be better if careful processing is done in feature selection or feature scaling [8]. In this project, we did not introduce auxiliary processes such as feature scaling or Markovization which have been empirically proved that are helpful for the final result. In addition, we use a very naive feature extraction function. As discussed in section 4.2, there are plenty of other feature functions that either contains contextual information or carefully engineered features depending on the specific problem domain that could improve final results.

As expected, Spectral-abs achieves the lowest variance in the Micro avg F1 score. This demonstrates one of the advantages of the Spectral Learning algorithm which does not suffer from local-optimal. By contrast, Pdg and EM algorithms have a higher variance in Micro avg F1. This is especially severe in the EM algorithm which makes

it extremely unstable resulting in the largest variance in both metrics. However, in Macro avg, Spectral learning has a higher variance, this could result from that the performance of spectral learning varies with the size of the tree or the number of latent states. The final parameter might vary after projecting into different dimensions.

Between Spectral algorithms, we can see Spectral-abs performs constantly higher than using softmax to regulate parameters. As discussed in section 4.2, this is because a large negative value will result in a large positive parameter if we take the absolute value, but will result in a very small positive value if we take its softmax. This again verifies Cohen's empirical results [8]. This also helps to explain why Spectral-softmax has a relatively larger variance. If a particular dataset contains more negative values after T-SVD, then it would result in a poorer F1 score.

Between Pgd algorithms, Pgd-simplex performed constantly better than Pgd-reparam. This deserves further investigation since the relationship between them is not very clear. On of the hypothesis is because Pgd-reparam, it contains many exponential operations as discussed in 4.5. We did not use any pre-given optimizer in the training script. It might be the case that there are flaws in implementation that did not regulate either gradients or parameter updates.

Algorithm	Avg Epoch(s)	Expected Total(min)
EM	50.43	840.50
Spectral	-	**20.57**
Pgd	18.77	312.83

Table 5.3: Runtime Comparison

This runtime is estimated on a laptop with Intel i7-8750H with a base frequency of 2.20GHz and a boost frequency of 4.10GHz, together with 32GB DDR4-2666 MHz RAM. As expected, the Spectral algorithm has the fastest running time. This is because we do not need to do any inference in the training phase. As mentioned in section 4.3, inference and Inside-Outside algorithm calculation could be time-consuming. Before using the approximate search algorithm, by using a simple depth-first search EM and Pgd would have an unacceptable time-complexity due to the size of our inference problem.

Space complexity and RAM consumption are not measured in the experiment. However, intuitively Spectral learning consumes much more RAM than other types of algorithms. As mentioned in section 4.2, sparse feature extraction and calculating projection matrix takes up lots of RAM and exceeded the maximum capacity my laptop have in the early phase of the experiment.

Chapter 6

Conclusions and Future Work

In this work, we introduced a new type of probabilistic logic programming, Weighted Logic Programming with Latent States. We extended from L-PCFGs and general WLP so that our new program could accept multiple types of variables and reify the same types of variables initialized from different observable facts. In addition, the number of parents or children is no longer constrained in axioms.

On top of that, we designed core computation algorithms for the program. We also introduce the Spectral Learning algorithm that could learn transition parameters quickly and without suffering from local-optima. We also extended the algorithm to deal with multi-parent rules by using connected components in variable interaction graphs.

To evaluate our algorithm, we decide to compare it with traditional algorithms like Expectation-Maximization, and a more modern algorithm Projected-Gradient Descent. We modified those algorithm so that they could fit in our problem domain. We also designed a random program generator that provide us with sufficient data.

In the implementation, we focused on the single-parent rule program only due to lack of time. The experiment result matches our expectations and hypothesis. The Spectral-Learning algorithm has the lowest variance and very acceptable F1 score, only a few points away from the best. This number would go up if a more sophisticated implementation is applied. We also try out different implementations and small derivations of Spectral learning and Pgd algorithm to explore more possibilities.

This work could be extended in many ways, including but not limited to:

- Implement and modify all algorithms to be compatible with multi-parent rule programs and evaluate on more datasets to analyze the performance.
- Model real-world problems, such as dialogue commonsense inference [3], branching factor analyses in biology [9] and evaluate the performance.
- Investigate more on negative values in sampling for Spectral Learning
- Design a more general feature extraction function such as using graph-based parsing [29].

We will continue to work on this topic, especially implementing the multi-parent compatible algorithms we designed over this summer towards paper publication.

Overall, this work could be seen as a small step forward in probabilistic logic programs in the open-domain world. It also again verified the efficiency and robustness of the Spectral Learning algorithm. We also hope the tryouts in the derivation of algorithms could help readers avoid twists and turns in their own implementation.

Bibliography

[1] Animashree Anandkumar, Daniel Hsu, Adel Javanmard, and Sham Kakade. Learning linear bayesian networks with latent variables. In *International Conference on Machine Learning*, pages 249–257. PMLR, 2013.

[2] Andersen Ang. Projected gradient algorithm - angms.science, Oct 2020.

[3] Forough Arabshahi, Jennifer Lee, Mikayla Gawarecki, Kathryn Mazaitis, Amos Azaria, and Tom. Mitchell. Conversational neuro-symbolic commonsense reasoning. In *AAAI*, 2021.

[4] Esma Balkir, Daniel Gildea, and Shay B. Cohen. Tensors over semirings for latent-variable weighted logic programs. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 73–90, Online, July 2020. Association for Computational Linguistics.

[5] Shay B Cohen, Robert J Simmons, and Noah A Smith. Dynamic programming algorithms as products of weighted logic programs. In *International Conference on Logic Programming*, pages 114–129. Springer, 2008.

[6] Shay B Cohen, Robert J Simmons, and Noah A Smith. Products of weighted logic programs. *Theory and Practice of Logic Programming*, 11(2-3):263–296, 2011.

[7] Shay B Cohen, Karl Stratos, Michael Collins, Dean P Foster, and Lyle Ungar. Spectral learning of latent-variable pcfgs: Algorithms and sample complexity. *The Journal of Machine Learning Research*, 15(1):2399–2449, 2014.

[8] Shay B Cohen, Karl Stratos, Michael Collins, Dean P Foster, and Lyle H Ungar. Experiments with spectral learning of latent-variable pcfgs. 2013.

[9] Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[10] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[11] Jason Eisner and John Blatz. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proc. of Formal Grammar*, pages 45–85, 2007.

[12] Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Practical weighted dynamic programming and the dyna language. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, page 281–290, USA, 2005. Association for Computational Linguistics.

[13] Alexandra Gavryushkina, David Welch, and Alexei J Drummond. Recursive algorithms for phylogenetic tree counting. *Algorithms for Molecular Biology*, 8(1):1–13, 2013.

[14] Joshua Goodman. Parsing algorithms and metrics. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 177–183, Santa Cruz, California, USA, June 1996. Association for Computational Linguistics.

[15] David Hall, Greg Durrett, and Dan Klein. Less grammar, more features. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 228–237, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[16] Daniel Hsu, Sham M Kakade, and Tong Zhang. A spectral algorithm for learning hidden markov models. *Journal of Computer and System Sciences*, 78(5):1460–1480, 2012.

[17] Finn V Jensen, Uffe Kjærulff, Kristian G Olesen, and Jan Pedersen. Et forprojekt til et ekspertsystem for drift af spildevandsrensning (an expert system for control of waste water treatment—a pilot project). *Technical Report*, 1989.

[18] Kamvar Kamvar, Sepandar Sepandar, Klein Klein, Dan Dan, Manning Manning, and Christopher Christopher. Spectral learning. In *International Joint Conference of Artificial Intelligence*. Stanford InfoLab, 2003.

[19] Yoon Kim, Chris Dyer, and Alexander M. Rush. Compound probabilistic context-free grammars for grammar induction. *CoRR*, abs/1906.10225, 2019.

[20] Karim Lari and Steve J Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56, 1990.

[21] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999.

[22] Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. Probabilistic cfg with latent annotations. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 75–82, 2005.

[23] Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. Learning reasoning strategies in end-to-end differentiable proving. In *International Conference on Machine Learning*, pages 6938–6949. PMLR, 2020.

[24] Davide Nitti. Hybrid probabilistic logic programming. 2016.

[25] Bernardo A. Petriz and Octavio Luiz Franco. Chapter one - application of cutting-edge proteomics technologies for elucidating host–bacteria interactions. In Rossen Donev, editor, *Proteomics in Biomedicine and Pharmacology*, volume 95 of *Advances in Protein Chemistry and Structural Biology*, pages 1–24. Academic Press, 2014.

[26] Avneesh Saluja, Chris Dyer, and Shay B. Cohen. Latent-variable synchronous CFGs for hierarchical translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1953–1964, Doha, Qatar, October 2014. Association for Computational Linguistics.

[27] Ilya Shmulevich, Edward R Dougherty, Seungchan Kim, and Wei Zhang. Probabilistic boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, 2002.

[28] Weiran Wang and Miguel Á. Carreira-Perpiñán. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application. *CoRR*, abs/1309.1541, 2013.

[29] Yajie Ye and Weiwei Sun. Exact yet efficient graph parsing, bi-directional locality and the constructivist hypothesis. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4100–4110, Online, July 2020. Association for Computational Linguistics.

Chapter 7

Appendix

7.1 Proof for 3.5

Consider a general multi-parent and multi-children rule $P_1...P_n \rightarrow C_1...C_m$ with inference relationship expanded as below. It has n parents and m children and a $(n+m)$-way transition parameter tensor T. We also assumes given a connected component C such that $P_1,...,P_n \subseteq C$ and $|C| = \varepsilon$.

$$P_1 \otimes ... \otimes P_n \oplus = C_1 \otimes ... \otimes C_m \otimes T^{P_1...P_n \rightarrow C_1...C_m}$$

Since originally we've provided function $\phi(In) \in R^d$ and $\psi(Out) \in R^{d\prime}$ we construct the feature matrix for each $P_1...P_n$ with concatenated feature vector with ϕ' and ψ'

$$\Omega_C^{P_i} = \mathcal{E}[\phi' {\psi'}^\top | C_j] \in R^{\varepsilon d \times \varepsilon d'}$$

This action has no effect on latent state numbers nor relevant projection. Since T-SVD decomposition has guaranteed to obtain the corresponding projection matrix given the number of singular values. We decompose the matrix as $\Omega_C^{P_i} = USV^\top$ where $U \in R^{\varepsilon d \times h}, S \in R^{h \times h}, V^\top \in R^{h \times \varepsilon d}$ supposing P_i has h latent states.

Recall that the definition of $I_h^{P_i} = \mathcal{E}[\phi(In^{P_i})|h]$ in condition 1, $I^{P_i} \in R^{d \times h}$ is a full rank matrix. After adapting the concatenation setup, even the size of I changed, it's still a full rank matrix regarding to latent states h. There could be zero entries in $I_h^{P_i} = \mathcal{E}[\phi'|h] \in R^{\varepsilon d \times h}$ considering some variable in the interaction graph does not participate in the current proof. However, $\phi(In^{P_i})$ must be in ϕ' according to the definition, and $d >> m$ makes the original matrix full rank, thus the current matrix is also full rank. This satisfies the condition 1.

Then we could derive a corresponding invertible matrix $G^{P_i} = (U^{P_i})^\top I^{P_i}$, the dimension stays the same as the single parent case since $h \times \varepsilon d * \varepsilon d \times h = h \times h$. Hence, we could also prove this for the pair J^{P_i} and K^{P_i}. So $\Sigma^{P_i} = G^{P_i} diag(\gamma^{P_i})(K^{P_i})^\top$. This satisfies condition 2 and lead to Theorem 3.2.1 naturally.

This extension also compatible with calculations since dimension matches. Given that

$$D^{P_1...P_n \rightarrow C_1...C_m} = \mathcal{E}[(R = P_1...P_n \rightarrow C_1...C_m) Z_1...Z_n Y_1^T ...Y_m^T | P_1...P_n]$$

By using the chain rule and marginalizing hidden variables we have

$$D^{P_1\dots P_n\to C_1\dots C_m}_{i_1\dots i_n\dots i_{(n+m)}} = \Sigma_{h_1\dots h_n\dots h_m} p(P_1\dots P_n \to C_1\dots C_m, h_1\dots h_n\dots h_{n+m}|P_1\dots P_n)$$
$$\times \mathcal{E}[Z_{i_1}\dots Z_{i_n} Y_{i_{(n+1)}}{}^{\top}\dots Y_{i_{(n+m)}}{}^{\top}|R = P_1\dots P_n \to C_1\dots C_m, h_1\dots h_n\dots h_{n+m}].$$

By definition we have

$$p(P_1\dots P_n \to C_1\dots C_m, h_1\dots h_{(n+m)}|P_1\dots P_n) = \prod_{k=1}^{n} \gamma^{P_k}_{h_k} \times t(P_1\dots P_n \to C_1\dots C_m, h_1\dots h_{n+m}|P_1\dots P_n)$$

According to the independent assumption we made, using definitions of K and G matrices, we have

$$\mathcal{E}[Z_{i_1}\dots Z_{i_n} Y_{i_{(n+1)}}{}^{\top}\dots Y_{i_{(n+m)}}{}^{\top}|R = P_1\dots P_n \to C_1\dots C_m, h_1\dots h_n\dots h_{n+m}]$$
$$= \prod_{k=1}^{n} \mathcal{E}[Z_{i_k}|A = P_k, H = h_k] \times \prod_{k=1}^{m} \mathcal{E}[Y_{n+k}|A = C_k, H = h_{(n+k)}]$$
$$= \prod_{k=1}^{n} K^{P_k}_{h_k} \times \prod_{k=1}^{m} G^{c_k}_{h_{n+k}}$$

By putting all these together, we could say for a multi-parent and multi-children distribution tensor $D^{P_1\dots P_m\to C_1\dots C_n}$ can be written as below

$$D^{P_1\dots P_n\to C_1\dots C_m}_{i_1\dots i_n\dots i_{(n+m)}} = \sum_{h_1\dots h(n+m)} \prod_{k=1}^{n} K^{P_k}_{i,h_k} \times \prod_{k=1}^{m} G^{c_k}_{(i+n),h_{n+k}}$$
$$\times \prod_{k=1}^{n} \gamma^{P_k}_{h_n} \times t(P_1\dots P_n \to C_1\dots C_m, h_1\dots h_{n+m}|P_1\dots P_n)$$

Hence, by the definition of tensor together with conditions 1 and 2, we have

$$[D^{P_1\dots P_n\to C_1\dots C_m}(y_1,\dots,y_m)]_{i_1,\dots,i_n} = \sum_{i_{(n+1)},\dots,i_{(n+m)}} y_{1\,i_{(n+1)}} \times \dots \times y_{m\,i_{(n+m)}}$$

$$= \sum_{h_1\dots h(n+m)} \prod_{k=1}^{n} \gamma^{P_k}_{h_k} \times K^{P_k}_{i_k,h_k} \times [T^{P_1\dots P_n\to C_1\dots C_m}(y_1 G^{C_1},\dots,y_m G^{C_m})]_{h_1,\dots,h_n}$$

$$= T^{P_1\dots P_n\to C_1\dots C_m}(y_1 G^{C_1},\dots,y_m G^{C_m}) \prod_{k=1}^{n} diag(\gamma^{P_k}) K^{P_k}{}^{\top}$$

This extends the proof in Cohen et al. [7], with such identity we've proven that the model is compatible with multi-parent rules, and by using the variable interaction graph and concatenated features the spectral learning algorithm is compatible with multi-parent rules with minimal extensions.