

Hierarchical Relational Learning

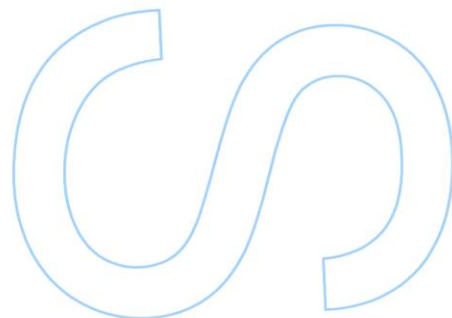
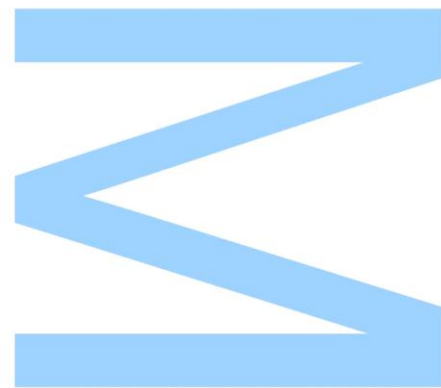
Luís Filipe Cruz Queijo

Mestrado Integrado de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2021

Orientador

Inês de Castro Dutra
Professora Auxiliar

Faculdade de Ciências da Universidade do Porto

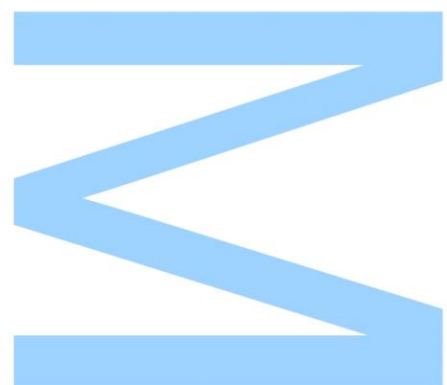




Todas as correções determinadas
pelo júri, e só essas, foram efetuadas .

O Presidente do Júri,

Porto, ____/____/____



Universidade do Porto

Hierarchical Relational Learning

Luís Filipe Cruz Queijo

Master's Thesis

Supervisor:

Inês de Castro Dutra

Departamento de Ciências de Computadores
Faculdade de Ciências da Universidade do Porto
September 2021

Acknowledgments

I first want to thank my supervisor, Inês Dutra for the amazing support that she gave me throughout the year, and the patience and willingness to help on my first grand academic accomplishment. I couldn't have made it without her suggestions and assistance, for which I am very thankful.

I would also like to thank my family as their unwavering support is what made me pursue such a high academic education, allowing me to publish this work. This also includes some close family friends that I grew up with and consider them as part of my family.

Finally, I would like to thank my friends that I made through these years, their friendship proved to be invaluable at surmounting the challenges that university presented us. I'd also like to thank a few close-knit friends from high school which helped me alleviate my mind from work when I needed it the most.

Abstract

Nowadays, classical machine learning approaches fail to capture relational and hierarchical knowledge, being unable to find relations between two elements in a multi-dimensional way. To reach this objective and capture intermediary relations between a lower-level and higher-level relation, we created a C-based program which processes a file with Prolog facts, and analyses them, building a relation network in which it iterates over and draws conclusions from. The result is a combination of learnt rules obtained by the usage of predicate invention on the different combinations of relations observed in the relation network. Such rules are simplified to the point that they are described by previously learnt lower-level rules. Such simplification of facts, and consequent rule synthetization is important, as it structures and organizes an initial input of data into something simpler to work with, and less redundant. While not acting as a substitute of a more advanced ILP system, it could be used as a pre-processing step, simplifying its execution.

Resumo

Hoje em dia, as técnicas de machine learning não conseguem capturar conhecimento relacional e hierárquico, sendo incapazes de obter relações entre dois elementos de forma multidimensional. De forma a alcançar este objetivo e capturar relações intermédias entre uma relação de alto-nível e outra de baixo-nível, criámos um programa baseado em C que processa um ficheiro com factos Prolog e os analisa, criando uma rede de relações na qual itera e tira conclusões. O resultado é uma combinação de regras aprendidas através do uso de predicate invention nas diferentes combinações de relações observadas na rede de relações. Estas regras são simplificadas até ao ponto em que são descritas por regras de baixo-nível previamente aprendidas. Esta simplificação de factos, e a sua consequente sintetização de regras é importante, dado que estrutura e organiza um input inicial de informação em algo mais simples, maleável e menos redundante. Embora não seja um substituto para sistemas ILP mais avançados, pode ser utilizado como um passo de pré-processamento, simplificando as suas execuções.

Contents

| | |
|---|-------------|
| ACKNOWLEDGMENTS | I |
| ABSTRACT..... | II |
| RESUMO | III |
| CONTENTS | IV |
| TABLE LIST..... | VI |
| FIGURE LIST | VII |
| ABBREVIATION LIST | VIII |
| 1 INTRODUCTION..... | 1 |
| 1.1 MOTIVATION | 1 |
| 1.2 OBJECTIVE | 1 |
| 1.3 SUMMARY | 2 |
| 2 BACKGROUND | 3 |
| 2.1 FIRST ORDER LOGIC..... | 3 |
| 2.2 LOGIC PROGRAMMING..... | 4 |
| 2.3 RELATIONAL MACHINE LEARNING | 5 |
| 2.3.1 <i>Inductive Logic Programming</i> | 6 |
| 2.3.1.1 Progol | 7 |
| 2.3.1.2 Aleph | 8 |
| 2.3.2 <i>Hierarchical Relational Learning</i> | 9 |
| 3 STATE OF THE ART | 10 |
| 3.1 AUTOMATIC AND HIERARCHICAL LEARNING | 11 |
| 3.2 FLEXIBILITY AND LIMITATIONS OF ILP..... | 16 |
| 3.3 COMPLEXITY AND OPTIMIZATION OF ILP APPROACHES | 19 |
| 3.4 PREDICATE INVENTION OVERVIEW AND ANALYSIS | 21 |
| 4 HIERARCHY DISCOVERY | 24 |
| 4.1 KNOWLEDGE STRUCTURE | 25 |
| 4.2 RELATIONSHIP GROUPING EXTRACTION | 29 |
| 4.2.1 <i>Structure</i> | 29 |
| 4.2.2 <i>Execution</i> | 32 |

| | | |
|----------|---|-----------|
| 4.3 | COMMON FACTOR DETECTION AND HIERARCHICAL LEARNING | 40 |
| 4.3.1 | <i>Structure</i> | 40 |
| 4.3.2 | <i>Concepts and execution</i> | 42 |
| 5 | RESULTS..... | 50 |
| 5.1 | METHOD 1 – ITERATIVE FUNCTION | 50 |
| 5.1.1 | <i>Input and output</i> | 50 |
| 5.1.2 | <i>Analysis</i> | 55 |
| 5.2 | METHOD 2 – ADJACENCY POINTERS | 57 |
| 6 | CONCLUSION | 59 |
| 7 | BIBLIOGRAPHY | 64 |

Table List

| | |
|--|----|
| Table 1 - Simplified comparison of ILP systems | 17 |
| Table 2 - Method 1 Runtimes in seconds | 54 |
| Table 3 - Method 2 Runtimes in seconds | 57 |

Figure List

| | |
|---|----|
| Figure 1 - Test Family..... | 25 |
| Figure 2 - Graph List | 26 |
| Figure 3 - Connected List | 26 |
| Figure 4 - Graph Order..... | 27 |
| Figure 5 - Group List | 30 |
| Figure 6 - Rel List..... | 30 |
| Figure 7 - Concrete Group List..... | 31 |
| Figure 8 - Module 2 initial execution..... | 33 |
| Figure 9 - Module 2 initial execution..... | 34 |
| Figure 10 - Discovery Function Trace | 37 |
| Figure 11 - Second Module Output | 38 |
| Figure 12 - Hierarchical Learning Alternatives..... | 41 |
| Figure 13 - Join List..... | 42 |
| Figure 14 - Common Factor Analysis Concept | 43 |
| Figure 15 - Bridge Analogy | 44 |
| Figure 16 - Learning without perspective..... | 44 |
| Figure 17 - Learning with perspective | 45 |
| Figure 18 - Join List Creation..... | 46 |
| Figure 19 - Common Factor Analysis..... | 47 |
| Figure 20 - Pandora Function | 49 |
| Figure 21 - Control Input..... | 50 |
| Figure 22 - Control Output..... | 51 |
| Figure 23 - Control+ Input..... | 51 |
| Figure 24 - Control+ Output..... | 52 |
| Figure 25 - Control+ Clustered Input | 52 |
| Figure 26 - Control+ Clustered Output | 53 |
| Figure 27 - Control+ Extended Input | 53 |
| Figure 28 - Control+ Extended Output..... | 54 |
| Figure 29 - Visual representation of method 1 runtimes (seconds)..... | 55 |
| Figure 30 - Conjunction and Disjunction..... | 56 |
| Figure 31 - Visual representation of method 2 runtimes (seconds)..... | 57 |
| Figure 32 - superSibling representation..... | 61 |

Abbreviation List

| | | |
|------|---|--|
| ADC | - | Automatic Discovery of Concepts |
| ASP | - | Answer Set Programming |
| BK | - | Background Knowledge |
| DHDB | - | Deductive Hierarchical Databases Clauses |
| FOL | - | First Order Logic |
| IBK | - | Interpreted Background Knowledge |
| IE | - | Inverse Entailment |
| ILP | - | Inductive Logic Programming |
| HRI | - | Hierarchical Relational Inference |
| HRL | - | Hierarchical Relational Learning |
| LP | - | Logic Programming |
| MIL | - | Meta Interpretative Learning |
| ML | - | Machine Learning |
| NRI | - | Neural Relational Inference |
| PI | - | Predicate Invention |
| RDR | - | Ripple Down Rules |
| RL | - | Relational Learning |

1 Introduction

1.1 Motivation

Nowadays, classical Machine Learning (ML) approaches fail to capture relational and hierarchical knowledge. As such, they are unable to find relations between two elements in a multi-dimensional way, either because they cannot process multi-dimensional data, or because they are under the assumption that the two datasets are isolated.

Hierarchical Relational Learning (HRL) is essentially an extension on Hierarchy induction for propositional logic, which tries to bypass the specified limitations. It will base itself not onto probabilities, but onto logic programming by reviewing information relevant to the elements and attempting to find connections between them.

Being able to learn Hierarchical Relational data from a set of elements brings a lot of benefits to further research into the given set. The immediate benefit that we obtain is the overall simplification of knowledge. If we can summarize higher-level terms by a combination of simpler terms, we are able to significantly decrease the complexity of our set, synthetizing them into a more compact structure.

While it might seem a minor benefit at first, it is important to understand that by doing this, we are also able to remove a lot of redundancy in the dataset, making it overall more organized, structured and overall easier to work with. This simplification could be quite significant as ILP systems are computationally intensive. Assuming that the knowledge is simplified as described above, it could be considered as an important pre-processing of the data before feeding it into more complex learning programs.

1.2 Objective

In this work, our objective is to be able to learn relations recursively from our initial knowledge. We want to be able to observe the basic facts and with them, create new rules which will themselves be used for the learning of higher-level relations. This should be done automatically in a recursive way, until the desired higher-level relations are obtained and

simplified. We also want to learn that information without the need for constant user definitions of what the higher-level relations should be, as it happens in most cases with the usage of predicate invention.

Most of current approaches fail to capture these relations without predefining some sort of support knowledge which will help guide their ILP systems into finding the desired relation, which is where our work hopes to innovate. By doing this, we can learn the relations that a given set of facts can represent, without the hindrance of a long setup for each and every higher-level relation we wish to learn.

1.3 Summary

In this thesis, we first begin by introducing relevant concepts and basic knowledge related to our work in chapter 2. These are useful to understand and contextualize our work, alongside state-of-the-art related research, which we present and discuss in chapter 3. In chapter 4, we discuss how we were able to create a C based program which accepts a file with a group of Prolog facts and is able to turn them into the objective of our work, which takes the form of learnt relations that are simplified by a single rule, each rule being comprised of two premises, with all rules being stored via usage of a list, which in turn acts as a sort of library. We test that program in different executions in chapter 5, observing its runtimes with different inputs. In chapter 6 we criticize and analyse our work in a more in-depth fashion, comparing it to other solutions currently available, as well as pointing out its limitations and strong points.

2 Background

This chapter covers basic knowledge and concepts that are required for the correct understanding of HRL, as well as its implication/relevance in the ML field. Each subsection will then cover either a field of study that is related in some way to our work, most of which are either used by, or are a branch of ML.

2.1 First Order Logic

First Order Logic (FOL) can be used as a way to represent knowledge in artificial intelligence. While propositional logic is perfectly capable of representing regular statements and facts, that is where the boundary lies. Should we need to represent something more than basic false/true propositions, like statements that are more complex, needing more expressive power, the First Order Logic (which is an extension to the Propositional Logic) would be a good solution to consider.

First Order Logic is also called Predicate Logic due to the way that it is structured. In an expression, there will be an interaction between a subject and a predicate. Such interaction can make use of functions, connectives, and quantifiers as to express the intended meaning. However, all the sentences can fall into two categories, depending on their structure: atomic and complex.

Atomic sentences are the most basic sentences, defined by N terms, which can be represented in a predicate($\text{term}_1, \text{term}_2, \dots, \text{term}_N$) fashion. More concretely, the sentence “Alice and Bob are siblings”, can be represented as $\text{siblings}(\text{alice}, \text{bob})$. Other examples might have a different number of terms but will always follow that same basic structure. On the other hand, complex sentences are a combination of atomic sentences while using connectives, in order to add another layer of meaning to a definition. For example, if we pick up the previous example and define P_1 as $\text{siblings}(\text{alice}, \text{bob})$, and afterwards define P_2 as $\text{siblings}(\text{bob}, \text{harriet})$, then a complex sentence could be $(P_1 \wedge P_2)$. However, the conjunction connective isn't the only one that allows the combination of atomic sentences to form complex ones. Amongst others, such connectives include \vee or \neg (which would represent disjunction and negation, respectively). Additionally, the universal quantifier \forall and

the existential quantifier \exists , as well as functions (which are used to evaluate objects) are also used in the formation of complex sentences.

Through the processing of information belonging to a statement in this fashion, we can convert the meaning of a sentence in a mathematical expression that computers are able to recognize, process, and work upon.

2.2 Logic Programming

Logic programming (LP) is a programming paradigm that allows programs to be expressive and flexible through the usage of formal logic, in the format of sets of sentences in logical form. Such sentences, or rules, express meaning through the usage of clauses. Assuming we are using Prolog's syntax (logic programming language), these rules follow a specific format,

$$\text{head}(a_1, a_2, \dots, a_n) \text{ :- } \text{body}_1(b_1, b_2, \dots, b_{n_1}), \text{body}_2(c_1, c_2, \dots, c_{n_2}), \dots, \text{body}_N(d_1, d_2, \dots, d_{n_N})$$

that can be read as “if $\text{body}_1(b_1, b_2, \dots, b_{n_1})$ and $\text{body}_2(c_1, c_2, \dots, c_{n_2})$ and and $\text{body}_N(d_1, d_2, \dots, d_{n_N})$ then $\text{head}(a_1, a_2, \dots, a_n)$ ”. The head of the rule is comprised by the statement that we wish to define, and consequently acts as a conclusion. The body of the rule contains the formulas that we will use as premises to test the preceding statement. One or more formulas can be present in the body of the rule, and the X indicates a sequence of Prolog terms. By using conjunction ($H \text{ :- } B_1, B_2$), disjunction ($H \text{ :- } B_1 ; B_2$) and other operators that will work as connectives, we can convey the specific meaning of a rule. More concretely if we have a rule such as:

$$\text{parent}(\text{laura}) \text{ :- } \text{mother}(\text{laura}) ; \text{father}(\text{laura}).$$

we can describe it less formally as “Laura is a parent if Laura is a mother, or if Laura is a father”.

It is also important to indicate that the body of an expression is optional in some cases. Should we wish to create an expression as a declarative statement, to later perform queries on it, we write a single rule predicate. The resulting expression is called a fact. This is usually done when we wish to introduce some sort of background knowledge (BK) for learning algorithms, where we can later execute queries on, where such facts are always true, and are formally known as axioms. The BK can be defined as a group of declarative and non-declarative statements. In ML, the BK, alongside examples, which are normally divided in two subsets: positive and negative, although some systems can take only the positive class or even multiple classes [1].

Now to put it more concretely, if we go back to the previous example, we create a fact:

```
mother(laura).
```

Afterwards, we perform a query:

```
?- parent(laura).
```

Assuming that we defined parent correctly, the result would be the program returning true. Since Laura is a mother, then according to the clause we had previously defined she would be a parent. This is a concept that will be important to be familiar with, as it will help understanding our work while also being a vital part on building databases.

2.3 Relational Machine Learning

Relational learning (RL) refers to learning in a context where there may be relationships between learning examples, or on their components due to having complex internal structures. As such, the “relational” can refer to the internal or external relational structure as there are no essential differences between these two cases, depending solely on the definition of example. An internal relational structure of an example consists of multiple components where there might exist relations between them, although there are no direct relationships that directly connect one example to another. External relations are cases

where each example has a relatively simple definition, for instance, a node in a graph where each example is linked to another via a relation. This is due to the fact that an example being either internal/external is attributed to the context and the definition of the example itself [2]. RL goes beyond the conventional analysis of entities in isolation, to instead analyse networks of interconnected entities. This is crucial in several world applications such as citation analysis and fraud detection, where the limited information about one entity is not very helpful, in contrast to the connection between several entities across different tables and attributes. So, by refusing to treat data as identical and independently distributed (as it has been assumed as such on propositional data), we are able to model more complex relational structures that are neither identical nor independent, which is precisely what we observe in the real-world applications [3].

2.3.1 Inductive Logic Programming

Inductive logic programming (ILP) is a form of machine learning which investigates the inductive construction of first-order clausal theories from examples and background knowledge. ILP is an intersection of ML and Logic Programming (LP), and its objective is to derive a hypothesis from a set of examples (both positive and negative) along with some known background knowledge.

The way that the ILP differs from regular deduction is that in regular deduction we are aware of the theory, and use resolution as an inference mechanism as to have answers to our queries. In ILP however, we have the examples and their proper descriptions, yet we don't have a general theory to describe them, in which case ILP algorithms may take advantage of inverse resolution where given the facts, we are able to induce the rules. ILP search can use bottom-up or top-down approaches. Bottom-up is when the search starts from the most specific clause that describes an example and is going to remove literals until we reach to the best, most general definition. Top-down works the opposite way, using the most general definition to reach the most specific one.

Due to its expressive and declarative nature, ILP can address a multitude of situations and problems that are associated with ML, such as the case of classification, regression, clustering, often expanding upon existing propositional machine learning systems [4].

2.3.1.1 Progol

Hypothesis construction is the core of any ILP system and, unsurprisingly, there are many different techniques for constructing hypotheses, including Inverse Resolution (IR), Relative Least General Generalisations (RLGG), Inverse Implication (II), and Inverse Entailment (IE). Progol [5] is an ILP system which combines IE with general-to-specific search through a refinement graph. By using mode declarations, Progol is able to derive the most specific clause within the mode language, which is used to guide a refinement-graph search. Progol is one of the most influential ILP system, one that inspired different approaches (Aleph [6], XHAIL [7], Atom [8]), which in turn inspired other ILP systems to be created (ILED, being an incremental version of the XHAIL algorithm).

To be able to derive its hypothesis, Progol needs to have a few specific components to work with, some of which we have already mentioned. First, we need BK, which is a group of facts and rules which Progol will refer to (similar to a library). That's the 'raw' knowledge which we will use to setup the system. Additionally, we will need to feed it with positive and negative examples, which in conjunction with the facts will help with the induction. We also need to add the modes, which will help our system in knowing what we want to learn, and the structure in which it will do so. We will need to add two kinds of 'mode's. The first one, 'modeh', is a mode declaration for head literals, and helps Progol learn the name of the concept/relation/predicate to be learned. The second one, 'modeb', is a mode declaration for body literals, and will indicate which predicates in the BK (with respective argument types and modes) are used to construct rules.

More concretely, if we take it a step further and want to learn 'parent' as being either a father or a mother of another person, we could define the following modes:

```
:- modeh(*,parent(+person,-person)).  
:- modeb(*,father(+person,-person)).  
:- modeb(*,mother(+person,-person)).
```

'+' signs represent input arguments, while '-' represent output arguments. 'modeh' will then teach Progol that the head literal we are looking for is called parent, and will be comprised by persons, with one acting as input and the other as output (person+ is parent of person-).

'modeb' will do something similar, but on the predicates present in the BK. It will then learn that it can use 'father' and 'mother' on the body, each of them having a person as input and another as output.

There is also a final detail, which is the 'determination'. The determination tells Prolog which predicates must be used to construct rules. In this case, it would be:

```
:- determination(parent/2,father/2).  
:- determination(parent/2,mother/2).
```

Which would tell Prolog that parent2/father2 and parent2/mother2 can be used to construct a definition for parent.

2.3.1.2 Aleph

Aleph [6] is an ILP system based on Prolog (having evolved to emulate some of the functionality of several other ILP systems) and is the latest implementation of the IE algorithm. It differs from its predecessor by having developed search strategies, such as randomized search that help improve its performance. It is implemented in Prolog and can be run with either YAP [9] or SWI [10].

The objective of Aleph is that given:

- A set of mode declarations M ,
- Background knowledge BK in the form of a normal program,
- E^+ positive examples represented as a set of facts,
- E^- negative examples represented as a set of facts.

It returns a normal hypothesis H such that:

- H is consistent with M ,
- $\forall e \in E^+, H \cup B \models e$ (i.e., is complete),
- $\forall e \in E^-, H \cup B \not\models e$ (i.e., is consistent).

To reach the desired hypothesis, Aleph follows a group of 4 simple steps. First, it selects a positive example to be generalized. If none exist, it stops; else it will proceed to the next step. From there, it constructs the most specific clause (called “bottom clause”) that entails the selected example and is consistent with the mode declarations. It searches for a clause that is more general than the previously constructed clause and that has the best score. Finally, it adds the clause to the current hypothesis and removes all the examples that were made redundant by it. [6]

2.3.2 Hierarchical Relational Learning

Hierarchical Relational Learning is the main topic on this work, and it is a subdivision of relational learning, with the goal of learning data in a multi-dimensional format from a set of given input. More specifically, we want to learn first-order rules in a hierarchical manner in a way that each level of search corresponds to learning new hierarchical concepts.

For instance, continuing with the previous family example, if we were to introduce proper knowledge so that we could learn that:

```
parent(A,B) :- mother(A,B).  
parent(A,B) :- father(A,B).
```

We could use that as our base knowledge, and from it learn a whole other layer of superior knowledge, such as for example:

```
grandparent(A,B) :- parent(A,C) , parent(C,B).
```

This will be the main objective of our work, to be able to learn such concepts, and be able to use it recursively as a base to other higher-level ones. While this example has been shown in Metagol’s page [11], it is a deprecated MIL system, and while it does learn grandparent, we need to feed it some substantial BK into it, which makes it ignore the relations that are in the middle (in this case, parent), only focusing instead on the higher relation (grandparent). This is precisely the issue that we want to address, by being able to learn the higher relation from the intermediate relations, which in turn are learnt from simple facts.

3 State of the Art

The analysis of the state of the art can be split onto three parts: the first one would be Andrew Cropper's work. He is a very interesting starting point in our search for related and past work on HRL as he is an influential researcher on the ML and ILP areas, having published several works in the described field, gave several conferences and talks about machine learning, inductive learning or one of his two ILP systems, as well as two theses. A good part of his past work was also made in partnership with S. H. Muggleton, which was previously mentioned as the creator of Progol, the ILP system that Aleph was based on (as so many others). More importantly, however, is the fact that the ILP area was first introduced by Muggleton in a paper, back in 1991. Another significant aspect is that a lot of his work is still quite recent.

The second part of our methodology is the search of related work through a query in Google Scholar. This query comprises the terms ["hierarchical" "first-order logic" "rule learning" -neural -probabilistic -robotic -"Hierarchical Pathology" -philosophy -Ontology] which yields about 58 results, from which we selected 7 to discuss. We considered only this small number of results relevant due to the fact that HRL being somewhat of a novelty in the RL area. It isn't as 'popular' or well defined as other areas such as predicate invention or logic programming. While there are a few examples where the analysis of hierarchies is observed in a learning context, there are very few works that approach HRL as specifically as we do, which makes our very specific query need to be satisfied by work that is tangentially related.

The first three terms are related terms to the core of this work. The three following terms that were removed are due to being either too specific or that pertain to a specific area of machine learning that isn't the focus of this work. The three final tags were removed due to being unrelated topics. An argument could be made that HRL is some sort of ontology, and that argument would be somewhat correct, however we aren't interested in the Ontology area itself.

The third part of our methodology is yet another query in Google Scholar, this time related with Predicate Invention. While it isn't specifically the objective of our work, it represents a part of it, and so it needs to be addressed in our research. For this query, we used ["predicate

invention" "hierarchical" "rule learning" -probabilistic -robotic] and obtained about 20 results, from which we selected 5. These were much easier to select than the previous 7, as shown by the ration of obtained results/selected results. In this query, we searched for work that is related to predicate invention, which is an area that is broader than HRL. We were able to obtain interesting results that allowed us to create a subsection dedicated to PI, introducing both its usability and major concerns.

In total, after selecting the ones that seemed to be relevant to our topic, the result was 2 works from Andrew Cropper and 12 from our queries, as described below.

The fourteen results which we decided to discuss were organized onto four different categories, each being important in its own way to our work, as well as the areas and main concerns related to it. The first subsection is about work that is related to automatic learning, or Hierarchical Learning, by either using graphs or higher order structures. The second subsection relates to the field of ILP itself, mostly observations about what the current state of the art is, as well as mentioning some interesting developments in the area. The third subsections will mostly analyse matters of complexity and optimization strategies regarding the execution of ILP based programs, and the final and fourth subsection will explore the predicate invention, as well as bring up some relevant issues and work currently done in the area.

Together, these four subsections aim to contextualize our work into the current state of the art, regarding the ILP area and its current advances. Relational learning is an ever-evolving area, where its flexibility allows for different methods, algorithms, programs, and techniques to be used to further expand and improve it in different ways, be it in matters related to its execution time, to how correct the extrapolated data is from these techniques, or what type of data can be used (and how much of this data is previously prepared, in the case of predicate invention).

3.1 Automatic and Hierarchical Learning

One of the objectives of our work is to be able to simplify a given group of relations, obtained through the analysis of facts. Synthesizing knowledge in such a manner has the advantage of turning the data simpler, more readable, and easier to work with in general. In “Learning

higher-order logic programs” [12], Cropper *et al.* work towards a similar goal, although instead of using first-order logic as we do, they extend it as to allow the usage of higher-order programs. The authors state that ILP, with its ability to learn first order programs, while being more expressive than traditional propositional programs, can still be worked on and expanded to allow the usage of such programs. Their initial data showed that an approach using meta-interpretative learning (MIL) would allow them to use higher-order programs, reducing the textual complexity required to express programs, which would in turn reduce the size of the hypothesis space and sample complexity. A simple overview about what is MIL and what can be accomplished with it can be found in Muggleton’s 2017 invited talk “Meta-Interpretive Learning: Achievements and Challenges” [13]. The procedure itself was very well structured, with the initial claim backed up by a simple example of how a Caesar’s cipher would be written in a significantly less complex way, essentially boosting performance. This was later proven true in the different experiments that they executed.

To reach a conclusion, they decided to test it in several ways, each one with several different algorithms. The experiments were spread across four domains (robot strategies, chess playing, list transformations and string decryption) and two new MIL systems (Metagol_{ho} and HEXMIL_{ho}). While we do not need to know the exact specifics of how the entire Metagol system works, we need to understand that there exists a `prove_aux` clause that tries to prove each atom in turn (atoms representing positive examples). Metagol_{ho} differs from Metagol by having a second `prove_aux/3` clause in its meta-interpreter which allows it to prove an atom by fetching a clause from the interpreted background knowledge (IBK) and proving it through meta-interpretation. Metagol_{ho}’s IBK differs from Metagol’s compiled background knowledge as the latter is proved deductively by calling Prolog. This simple change in the code allows for predicate invention, through the usage of conditions and functions, reducing the complexity of the rules that Metagol must learn. A similar approach was used on HEXMIL, although its inner workings are slightly different from Metagol. While Metagol searches for a proof using a meta-interpreter and SLD-resolution (Selective Linear Definite clause resolution), HEXMIL searches for proof by encoding the MIL problem as an Answer Set Programming (ASP) problem. ASP is a branch of logic programming meant to solve difficult (primarily NP-hard) search problems by modelling a representation, opposed to Prolog’s query derivation. ASP solvers employ efficient conflict propagation, which is important for detecting the derivability of negative examples

early during ASP search, making ASP implementations more efficient than Prolog implementations. To add support for higher-order definitions, it introduces a new ASP predicate called *ibk* which is responsible for encoding the higher-order atoms that occur in IBK, by encoding higher-order clauses as a mix of deduced atoms for first-order predicates and *ibk* atoms for those that involve predicates as arguments.

The first test was to teach a robot what kind of drink (tea or coffee) it needed to pour onto a set of cups. Regarding methodology, there would be a random number of cups and a set of positive and negative examples. At the end, the accuracy and the learning times would be noted down, resulting in the higher order variations of Metagol and HEXMIL to have significantly better results, with Metagol showing a much better scalability than HEXMIL.

The second test was based on chess, on the idea of advancing a black wall of pawns to the end of the board, as to attain promotion. The other pieces from the black set will not move and there would be no interference from the white pieces. Knowing this, and with the same methodology as the previous exercise (the difference being that instead of a random number of cups, it would be a random number of pawns), we reached similar conclusions, with Metagol's results being 100% accurate past the first two examples.

The third example was a more direct programming application, by attempting to learn a program that drops the last element from each sublist, given a list of lists. The methodology was similarly applied; however, the results were different from the past two. This time, while the Metagol results were consistent with what we have seen so far (by being vastly superior on the higher order version), the accuracy of the HEXMIL started near 100% with 2 training examples yet sank to 50% at 14. Similar poor results were seen on the learning times. Once again, these results were due to the poor scalability of ASP, and the inability to scale given more examples.

The fourth and final example revisited the encryption example that was presented in the introduction. The methodology was like the past examples, with the only variation being the random problem specific value that was selected, in this case to decide the size of the string. The results were even more extreme than in the past example, with Metagol_{ho} exceeding the regular Metagol version in both accuracy (reaching 100% at 6 examples) and learning

times. HEXMIL on the other hand, failed to deliver results (even on the most basic scenarios) under the allowed time.

The results from their work were pretty straightforward and supported the claim that compared to learning first-order programs, higher-order programs can improve learning performance.

The usage of Hierarchical techniques aren't, however, limited to Prolog based learning. While not exactly pertaining to the specific area of our work, Hierarchical Relational Inference (HRI), has a lot in common with HRL. More specifically, the approach that the authors of 2020's paper "Hierarchical Relational Inference" [14] took regarding the problem that they identified was worth a notable mention in this section of our work.

On the real world, the common-sense reasoning is based upon our knowledge of the objects that we encounter and their own specific interactions. The notions of abstract objects however, while having a somewhat intuitive concept to it, differ greatly in terms of supported behaviour. Different objects have different shapes, some are even deformable or not static. Some objects even have a complex behaviour, which can be independent in each of their parts, but behave differently as a whole, as is the case of an arm and fingers. To fix this issue, and to correctly identify these abstract differences, a new approach was based on a modelling of objects as hierarchies of parts, as to allow the distinction of different levels of abstraction consequently separating different hierarchies. They introduced a method that learns in an unsupervised fashion, able to learn from raw visual images as to discover objects, parts, as well as their relations, resulting in the correct identification of different levels of abstraction, improving at modelling synthetic and real-world videos.

What was quite interesting however, was that HRI extends Neural Relational Inference (NRI) as to allow the inference of hierarchical interaction graphs to simplify the modelling of the dynamics of more complex objects, as well as allowing a mechanism to apply NRI to raw visual images that infer part-based object representations spanning multiple levels of abstraction. The result is that HRI is a valid method for prediction of abstract objects, outperforming consistently synthetic and real-world physics predictions. There is a negative aspect, which is that since it groups parts into objects based on spatial proximity, it may be

suboptimal in case of severe occlusions, although it can be fixed by increasing the hierarchy depth.

While the application of hierarchical structures on MIL and HRI are certainly worth to mention, we do not want to lose sight of our goal. With that in mind, out of all the papers that were obtained from our query, this next one might be the most aligned with our work, as it entails automatic discovery of concepts, predicate invention and ILP.

The paper itself is called “Automatic discovery of relational concepts by an incremental graph-based representation” [15], and as the authors put it, automatic discovery of concepts has been an elusive area in machine learning. While it is our objective in this work to make it work with HRL, a similar goal was also achieved in a different manner through the usage of a system called Automatic Discovery of Concepts (ADC), that automatically discovers concepts in robotic domains through the usage of predicate invention. Predication invention is, as defined by Kok and Domingos [16], the creation of new symbols, together with formulas that define them in terms of the symbols in the data, referring to both the creation of new predicate symbols and new constant symbols.

The general concept for this idea is that an agent, using ADC, creates an incremental graph-based representation with the information that it gathers while exploring its environments. From that, it identifies sub-graphs, which will each be an instance of potential relational concepts. Similar sub-graphs will be grouped, and general concept definitions are induced with Inductive Logic Programming and predicate invention. They used different examples as to test the efficiency of their algorithm, and the results were positive, with it being able to learn the concepts in a satisfactory way.

They conclude their work by saying that automatic concept discovery has been a difficult task in machine learning, but they managed to contribute to that area by creating a new algorithm that successfully discovered relational concepts while exploring unknown definitions. Having reached this far however, there is still room for improvement, such as introducing intelligent exploration strategies, or produce definitions of concepts from unconnected sub-graphs.

Hierarchical and/or automatic learning are thus very desired and sought-after solutions, not only for their automatization, but for the capacity that they have to greatly simplify the relations or complexity of operations involved in future learning.

3.2 Flexibility and limitations of ILP

The best way to start this subsection is by first getting a good grasp about the current state of ILP. As such, “Inductive logic programming at 30: a new introduction” [17], isn’t as much as to prove a statement or theory, but instead acts more as an in-depth introduction to what is ILP, the main goals, how advanced the area is compared to when it was first introduced 30 years ago [18], as well as a comparison of several different ILP systems and a more profound analysis of four in specific (Aleph, TILDE, ASPAL, and Metagol). There are also comments on the application areas of ILP, limitations and future work.

It starts by presenting some basic concepts and definitions of what ILP is and compares it to traditional ML applications on three different examples: concept learning, string transformations and sorting. The conclusion at the end of the subsections was that ILP was a step ahead in terms of creating human readable hypothesis and have the capability of generalizing beyond the training data, being more data-efficient, while many other forms of ML are known quite the opposite, clearly illustrating the difference between ILP to most ML approaches. If we are to compare it further, we can see that the ILP has an obvious advantage compared to regular ML techniques (as it happens for tabled ones or decision tree learners) in matters of more complex representations, such as infinite relations by a single expression, more complex relational theories and algorithms, the capacity to reuse learned knowledge, as well as the innate ability to support relational data such as graphs.

The document further shows more information related to the basic building blocks of an ILP system, which are the learning setting (how to represent examples), representation language (how to represent BK and hypotheses), language bias (how to define the hypothesis space) and the search method (how to search the hypothesis space). They scrutinized the settings that an ILP system is built on and organized the information of several systems, arranging them in a table. It included several ILP systems, such as Progol, Aleph, Metagol and HEXMIL. With all these systems displayed, further descriptions were made on the following subsections where they approached the different kinds of hypotheses,

how to use the background knowledge effectively and its caveats, language bias (which include the mode declarations that are used by Aleph to indicate which predicate symbols may appear in a clause, how often, and their argument types) and the search method which include the top-down and bottom-up approach. Afterall, we cannot forget that in the end, ILP systems are tools made for a specific job in mind, each system having its strengths and weaknesses, and thus need to be chosen accordingly.

When comparing the multitudes of different ILP systems, they also compared different features, such as their noise handling capacity, if they were optimal, if they could handle infinite domains, if they were recursive and at last if they supported predicate invention, as shown in Table 1.

| System | Noise | Optimality | Infinite domains | Recursion | Predicate invention |
|---|-------|------------|------------------|------------------|---------------------|
| FOIL (Quinlan, 1990) | Yes | No | Yes | Partly | No |
| Progol (Muggleton, 1995) | No | Yes | Yes | Partly | No |
| TILDE (Blokceel & De Raedt, 1998) | Yes | No | Yes | No | No ¹⁴ |
| Aleph (Srinivasan, 2001) | No | Yes | Yes | Partly | No |
| XHAIL (Ray, 2009) | Yes | No | Yes | Partly | No |
| ASPAL (Corapi et al., 2011) | No | Yes | No | Yes | No |
| Atom (Ahlgren & Yuen, 2013) | Yes | No | Yes | Partly | No |
| ILASP (Law et al., 2014) | Yes | Yes | No ¹⁵ | Yes | Partly |
| LFIT (Inoue et al., 2014) | No | Yes | No | No ¹⁶ | No |
| Metagol (Muggleton et al., 2015) | No | Yes | Yes | Yes | Yes |
| ∂ ILP (Evans & Grefenstette, 2018) | Yes | Yes | No | Yes | Partly |
| HEXMIL (Kaminski et al., 2018) | No | Yes | No | Yes | Yes |
| Apperception (Evans et al., 2019) | Yes | Yes | No | Yes | Partly |
| Popper (Cropper & Morel, 2020) | No | Yes | Yes | Yes | No |

Table 1 - Simplified comparison of ILP systems

From these features, it is interesting to note that while the four's results in the table are somewhat spread out in terms of what the different ILP's can or can't support, almost all of them are unable to handle predicate invention. In a table with 14 ILP systems, only two can handle PI, while other three handle it only partially. This lack of success at integrating PI into ILP systems is attributed to three major reasons: the uncertainty of when to create a new symbol, how to invent a new symbol and how many arguments does it need to have, and a metric to evaluate the quality of a symbol, allowing us to either keep or discard invented

symbols. This lack of support is a bit concerning, considering the optimizations that PI can bring us, if done right.

Specifically, Aleph is a ILP system which is based on Progol, but its implementation is easier to use. While it uses a top-down approach to find the best hypothesis, it isn't pure as it adds a higher bound (the most general hypothesis) and a lower bound (most specific hypothesis) to the hypothesis space, making it very efficient at identifying relevant constant symbols that may appear in a hypothesis. However, since it learns its hypothesis through inverse entailment (IE), it struggles to learn recursive programs and optimal programs. Additionally, Aleph uses lots of parameters in its learning, which shape the resulting output. This heavily impacts the learning performance, and it produces results which are not desired if we aim for optimization. Still, despite its disadvantages, it has a sturdy and easily available implementation with good empirical performance, making it one of the most popular ILP systems available.

They conclude that while there have been major strides in the ILP area, predicate invention, and the usage of higher-order logic for hypotheses to name a few. Still, there is room for improvement, for example with language biases, probabilistic ILP and the capacity to read from raw data. All three of these would be impressive achievements, easily becoming a major breakthrough in not only ILP, but for the field of AI as a whole.

For instance, the application of inductive logic programming in data mining is something that can be quite profitable, especially when dealing with more complex structures such as graphs or multiple tables, which involve multiple relations. In "Foundations of Rule Learning", Fürnkranz *et al.* [19], the "Relational Features" section dedicates itself fully to the goal of going beyond simple relational analysis and applying it to more than a single data structure, which was one of the advantages of RL, as we had seen on the first section of this work. While it may not be related to learning data in a hierarchical manner, it is an interesting review about how to expand the level of dimensions that we are able to learn at once, as well as the relations between them.

Another great perk of ILP, is that while it is very flexible and powerful on its own, it can be used conjointly with other tools. In fact, in the introductory part of our work, we have mentioned the limitations of propositional logic, however that isn't to say that it isn't useful

while used with inductive logic. This specific work from Lavrač and Flach [20] aimed to use propositionalization as a transformation method. They mentioned the limitation of being unable to deal with nondeterminate local variables in the body of hypothesis clauses yet argue that it can be overcome by systematic first-order feature construction using a particular individual-centered feature bias.

They used LINUS, an ILP learner which induces hypotheses in the form of constrained deductive hierarchical database (DHDB) clauses. They employed a method as to effectively use background knowledge in learning both propositional and relational descriptions. Afterwards, they went one step further and overcame the DINUS (LINUS's successor) restriction of determinate literals. This was done by employing an individual-centered representation, which in turn would allow the LINUS hypothesis language to be extended in a way to learn nondeterminate DHDB clauses. Further actions to increase the efficiency of the algorithm were considered, such as the usage of the descriptive learner Tertius to generate only features that correlated sufficiently with the class attributes. Additionally, they concluded that there is a trade-off between how much effort a learner puts between the steps of the hypothesis generation process (rule construction, body construction, feature construction). Through alterations in one of the three, they would be able to solve complex relational learning tasks, although depending on the case, they would be somehow limited on their execution. This didn't seem to be an issue, since "concept learning and program synthesis are two very different tasks, which are probably not solvable with one and the same learning method" [20].

3.3 Complexity and optimization of ILP approaches

In the previous subsection we were able to get more familiarized with ILP and the multitude of ways that it can be used, however there are of course drawbacks. Putting it in a more concrete way, one of the issues of ILP is that while it is a very flexible machine learning technique which allows induction on first-order logic theories, it has the flaw of being relatively time-consuming. This will of course depend on the coverage test used by most ILP systems and how efficient their resolution decision procedure is; however, a large hypothesis search space will definitely be an issue on the most complex cases.

This was the problem brought up in Ferreira's work [21], problem which he proposed to solve through the usage of propositional logic-based inference. However, there was an issue regarding the lengthy formulas that propositionalization of first-order logic would create. Unfortunately, while he certainly brings up a few interesting issues and speculations on the viability of his offered solutions, there are no definitive answers as to whether propositional logic for ILP would be beneficial as to reduce the formulas' size.

While the attempt to decrease the hypothesis space wasn't entirely successful, it isn't a recent problem. In fact, in the Inductive Logic Programming book [22], there is a section (Extension of the Top-Down Data-Driven Strategy to ILP) where amidst the problem solving of the chapter, the idea of reducing the hypothesis space by covering a seed example was mentioned. We had seen previously that Aleph is one of the ILP systems that adds an upper and lower bound on the hypothesis space, effectively diminishing the time needed to find relevant symbols for the hypothesis. However, the Top-Down Data-Driven strategy, which was popularized by the AQ (Algorithm quasi-optimal A^q, with its initial version developed in 1969 by Ryszard S. Michalski) family, has not yet been transferred to ILP, since that if it would, the idea of reducing the hypothesis space by covering a seed example would make ILP systems such as Aleph to not benefit from the associated data-driven specialization operator.

This is troublesome, so they ended up presenting the data-driven strategy of AQ in the book and show how they managed to extend it to ILP, later evaluating an implementation of AQ in the system ProPal [22].

Another technique that attempts to fight this problem of having a big hypothesis space is the Top Directed Theory Derivation (TDTD). This method is used and discussed in LIN's dissertation [23], where they assert that it works due to the usage of the logic program T theory as a declarative bias that defines the search space. It differs from Inverse Entailment (as is the case of Aleph) as it derives the hypothesis deductively. It uses clauses in T theory to replace the hypothesized theory itself in the refutation for individual examples that make this forward/deductive computation feasible. Due to now benefitting from the deduction completeness, T no longer suffers from the incompleteness that might be found in IE-based methods.

The researcher used this knowledge and proceeded to create and implement a new ILP system for TDTD. This system would be able to efficiently learn multi-clauses problems correctly and in an efficient manner due to being an example-driven method, effectively bounding the search space. Furthermore, it does not suffer from redundancy and while efficient, it is also complete, being able to integrate both abduction and induction at the same phase, which results in both abductive and inductive hypothesis being able to be learnt at the same time.

3.4 Predicate Invention overview and analysis

When we are learning relations or concepts in certain domains, our capacity of acquiring new information depends on the available vocabulary that our domain possesses. By vocabulary, we mean the predicates, functions, and constant symbols, which will appear on the facts and the rules that will be used as background knowledge.

With that in mind, predicate invention is a method that allows to expand a given theoretical vocabulary to allow certain definitions based on observation of our predicates. While methods to achieve that goal vary between ILP systems and other techniques that are used in research, such predicates will be for the most part previously defined, with its contents either added at the same time of learning, or afterwards. It is important to note that while predicate invention is a powerful learning method, it is limited by how big the vocabulary can be augmented, and we need to know to some degree what to expect from the learning, we can't learn new definitions in a totally 'blind' way.

Stephan Kramer discussed higher-level learning from data in his paper [24] which was presented at IJCAI 2020. In it, he discussed about several symbolic higher-level representations, such as feature construction and constructive induction, predicate invention, and a few others, arguing that these approaches can benefit from each other to solve current issues in machine learning.

Kramer addresses the vocabulary issue by first discriminating three types of predicate invention methods. The first one is a *reformulation approach*, which as its name implies, introduces a new intermediate predicate by reformulating an already existing one. This is done to represent a theory in a more compact way and can be used as a way to either

optimize a theory or correct it should the theory fail. Opposed to this method, where there is too much information, *demand-driven* approaches are used in situations when there is not enough knowledge to correctly represent or learn a theory. The final method, called *clause-refinement* is used to make an over-general clause consistent by adding a literal that contains a new predicate. However, before the clause is changed and refined, there is a need to understand which clause of the theory was the one to blame for the instances that were covered incorrectly. For that end, different methods will use a different mix of knowledge at their disposal, such as positive and negative examples, or other clauses to discriminate between different instances.

After mentioning some example methods, the author identifies the excess of options for predicate invention, and a lack of options to assess the utility of candidates as prominent problems on current research. He also mentions that another big issue is the comprehensibility of new predicates, which is something that has been studied and discussed in the past, with some attempts at solving that issue being made. He specifically mentions Metagol as an enabler of ‘substantial progress’. In fact, there were attempts to create a general framework for predicate invention at a meta-level [25].

The matter related to the complexity of the predicates and how little comprehensible they can be is a matter that has been studied and is a big hindrance to both the development of the area, as well as minimizing the optimal learning process of the user. Suryanto and Compton [26] attempted to develop machine learning techniques which would speed up acquisition from an expert. They argued that once the expert shared some knowledge to the program, the program would generalise such knowledge to keep knowledge acquisition to a minimum. This process of generalization should be completely hidden from the expert. They were able to reach such a goal, minimizing the knowledge acquisition by up to 50%.

Such a goal was attained by using Ripple Down Rules (RDR), which allowed them to reduce the need for knowledge engineering. Due to having a lack of internal structure, RDR is unable to generate by itself intermediate conclusions, yet it is this same aspect which allows it to be quick and easy to build and maintain RDR systems. RDR is able to accept previously defined intermediate conclusions, yet that would defy the point of the user identifying reasons for a conclusion. The ideal solution (and consequently the author’s objective) is to use predicate invention to use in conjunction with RDR to generate intermediate

conclusions, while assessing their usefulness and removing the unwanted ones. They conclude by presenting a table of different examples, all of which having fewer rules than the user had to introduce.

Another interesting example from an attempt of understanding the comprehensibility of certain concepts related to machine learning can be observed in Muggleton's 2018 article "Ultra-Strong Machine Learning: comprehensibility of programs learned with ILP" [27]. Muggleton pointed the performance of comprehensibility of generated hypothesis as lacking, opposed to predictive accuracy due to the fact of the latter being more easily evaluated in the past, the first measure was mostly ignored which would end up favouring statistical over symbolic machine learning approaches.

The author then proceeded to do two human experiments as to discriminate what are the involved elements that affect our comprehension in matters related to machine learning. They observed that the comprehensibility was affected by both the complexity of the program, as well as anonymous predicate symbols. The participants couldn't learn the relational concepts on their own, although they could apply those same definitions when given by the ILP system. They then conclude that these findings point to the fact that there is a class of relation concepts that are hard to learn for humans, yet easy to understand if assisted by an abstract explanation.

Muggleton also worked on another issue related to predicate invention, which is the lack of theoretical results, caused (at least partially) due to a lack of theoretical framework for describing PI. The author then proceeds to give a small introduction to PI and its relation to ILP, then further detailing logical theorems related to both areas [28].

4 Hierarchy Discovery

In this chapter we address the process of designing a solution towards our objective, which is the recursive learning of higher-level knowledge, with the background knowledge we input at the start as a base. From these relations, we then learn further knowledge from them, until we learn all the possible relations, within our specified range.

This section will be split into three subsections, each of them encompassing a significant part of the designed code (module). This division will be made in accordance with the sub-problem that the respective module will be trying to solve.

For the initial setup of our work, the usage of Aleph and YAP was considered, yet we felt that the employment of Aleph as to create rules for YAP to be able to process was overdoing it, as our work is somewhat more limited than the broad, general usage that Aleph would allow, as it will be explained further down this section. With that in mind, we instead opted to use another language to create our own, simpler, and more specific analysis of relations, so that we could create a purpose-built program towards our goal. This was done in a way that it also allows modifications within the code as to process other knowledge than family trees (which was the example that we used from the beginning of our work).

At first, we considered either C or Java (Java in specific would have been very interesting to work with due to its object-oriented nature) yet ended up settling on C due to being faster overall, and much more flexible, attributes which are important for our work as we weren't sure at the beginning what we needed to reach our goal, or what kind of structures to use. Additionally, the fact that C is faster is a good thing all-round in most projects, but benefits us specially, as relational learning is quite heavy, complexity wise. The subsequent work was done in Linux (specifically Ubuntu 64-bit, with the usage of GCC compiler).

4.1 Knowledge Structure

While the decision of discarding Aleph and YAP gave us a lot more freedom and control on our work, it also made it that we have no initial knowledge processing, nor a structure that can hold all that information. This is what the first module is going to be about: obtaining the BK, creating a structure that is able to hold all of that information without loss, and in an easy enough way that we are able to iterate over it for post processing.

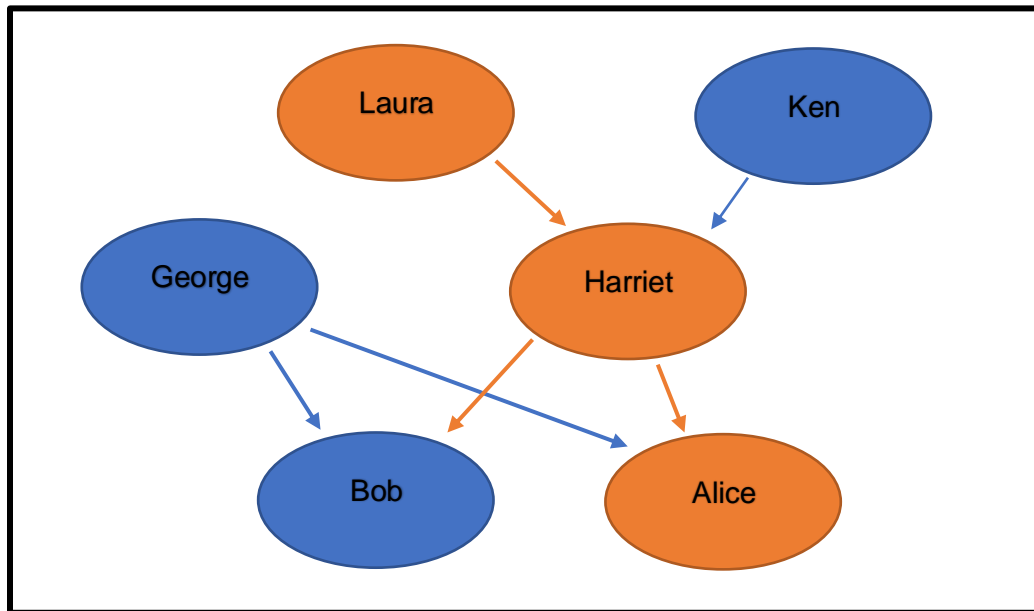


Figure 1 - Test Family

Figure 1 shows the 6-member family which was used to test and develop our work through its different stages. It is a static and concrete example, which allowed us more stability in terms of development and results. It is also the example that we use throughout this chapter to explain its functionalities. It represents a simple family tree via a directed graph, with blue arrows indicating that a node A is a father to a node B, or orange arrows indicating that a node C is mother of a node D. Ideally, we would like to group all our BK in such a structure, so that we can work over this 'condensed' form, instead of scattered bits of information.

Applying this design into code was a bit trickier, however. C doesn't have a native graph structure, as it has for Lists for instance. One option was to use community made libraries, but we instead preferred to create a custom structure as to give us more control and flexibility. A fundamental point is that we aren't aware of how much or how little the family is supposed to expand. In this case, we know it is six, but we need to have a structure that can

handle N nodes of family members, so we decided to create a list, due to its expanding potential. The list has its structure detailed in Figure 2.

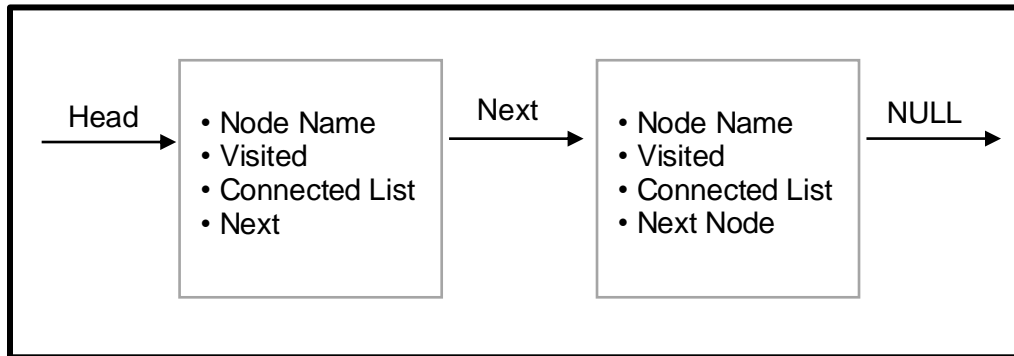


Figure 2 - Graph List

This list has 4 fields in each node. 'Node name' refers to the name of the family member, for example Laura. 'Visited' acts as a flag, which is needed on another module, as a way to keep track of progress. 'Next' is a pointer towards the next node of the list, and 'Connected List' acts as an adjacency list. The format of the adjacency list is displayed on Figure 3.

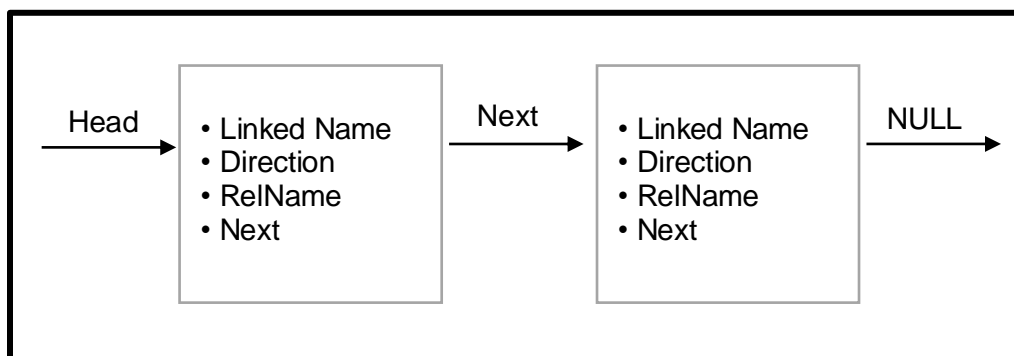


Figure 3 - Connected List

Similarly, this list also has 4 fields in each node. For the relationship of Laura with Harriet, for instance, we can define the following: 'Linked Name', which refers to the name of the node that is connected to, 'Harriet'. 'Direction' indicates whether the relationship is ingoing or outgoing. This is important as explained further in this chapter, being in this specific case 'Out'. 'RelName' is the name of the relationship, in this case being 'Mother'. 'Next' is a pointer to the next node.

With these Lists set up, we now have a solid structure to hold the BK from the family tree, without losing any information in the process. However, if we look closely at how the Graph List is defined, we notice that instead of the graph shown on Figure 1, we have instead (depending on the order that we process the BK) something similar to Figure 4.

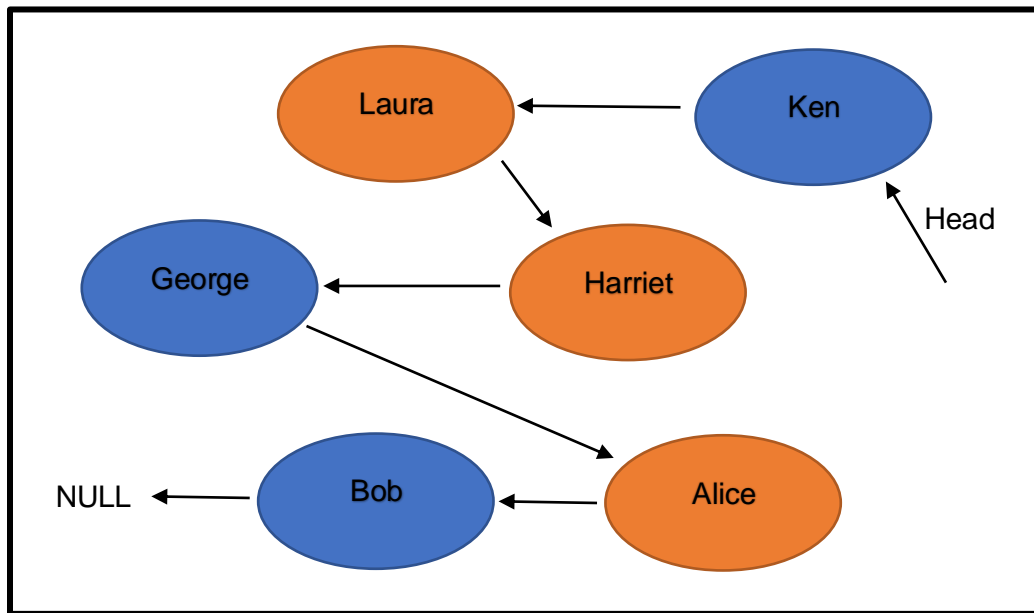


Figure 4 - Graph Order

It is important to point out that this is not an issue since each node has a list of their adjacent nodes, the Connected List. This makes it so that while the Graph List is a collection of nodes which in this case represent the family members, each node is responsible for being aware of their own relations to other nodes. This differs from regular graphs, where the information of a relation is on the relation itself, whereas here, all the information is instead stored on the nodes. This makes it easier for us to obtain all the relations of a node, by merely peeking into the Connected List. With a more concrete example, by examining the Connected List of the node 'Harriet', we see 2 relations: [Bob, Out, Mother] and [Alice, Out, Mother].

Another important feature that we need to mention for future reference is the fact that on the Group List, each node has a 'Direction' variable, being either Out or In. This was created initially to navigate back and forth between nodes, however another solution was soon found, and it became obsolete, until later was used for another purpose, which will be thoroughly described in the third subsection. For now, it is important to know that this In/Out variable serves the following purpose: for each Out-relation present in a node A, describing an outgoing relation from A to B, there will be an 'inverse' In-relation present in node B,

describing an ingoing relation from A to B. The relation itself is not an exact inverse of the Out relation (for example, the 'inverse' of a Mother, could be either a Daughter or a Son), and it is added to the Connect List of the node.

With that in mind, and if we pick up the previous example of the node 'Harriet', its Connected List has 4 nodes instead of the previous 2: [Bob, Out, Mother], [Alice, Out, Mother], [Laura, In, InvMother] and [Ken, In, InvFather]. While this way to represent relations might be a bit confusing and unintuitive at start, it brings us some advantages in the third subsection. The relations that we end up having aren't as straightforward as the ones in Figure 1, but nonetheless serve a similar purpose, as it allows the nodes to be more conscious of their surroundings, making it possible for them to consider all their relations (both outgoing and ingoing) when looking for higher level relations.

So far, we are able to tackle all our objectives for this module. We can obtain the BK without losses, hold that same BK in an organized and familiar way via the usage of a structure similar to a graph, which only leaves the issue of how to iterate over the different nodes of the graph. We considered two major choices on how to go about this, each with its own drawbacks. The first option would be, for each node in the Graph List, to add a pointer in all the Connected List nodes. That way, each node of the Graph List will not only be aware of the ingoing and outgoing relations that they have, but each of those relations will have a pointer to the other node they are related to. The second option would be to simply create a function which would iterate over the Graph List, looking for the specific node. This would imply that we would need to have some sort of Key to guarantee uniqueness, which we would use to find the specific node. Both options are viable on different contexts. Should we have a small graph, the second option would potentially be better, as an iteration over a small Graph List wouldn't affect much the overall performance of the program. With larger datasets however, a better choice might be the usage of a pointer towards the adjacent nodes. In theory, the program would have a heavier initial execution as it would need additional steps to setup the structure properly. However, we are trading off that bulky start by a more convenient and easier search mechanism later into the execution. The larger the final graph becomes, the more searches will need to be executed post setup for the learning of higher relations, and thus we would save up time by already having the pointers at hand instead of executing an iterative function repeatedly. For our example however, we decided to go with the second choice, as the test family has a low number of nodes.

We thus reached the conclusion of the first module. With all the information now able to be contained in our custom graph, with the capacity to iterate over it as we need, we are now ready to proceed into the next step of the execution: the extraction of relationship groupings.

4.2 Relationship Grouping Extraction

In the first module of our program, we are able to create a structure for a general representation of the background knowledge that we obtained. However, while it is useful to create an iterable graph to serve as our basis, we need to extract more particular information, specifically related to the relations. The names of the nodes, as well as most of the information regarding the ingoing and outgoing relations pertaining to them will be discarded, as what we are after right now, in this module, is the network of relations, and what unique groupings we can observe and extract from said network.

The relations that we obtained from the input of BK were all singular, as in they were only relations that linked two nodes via a singular relation (Mother/Father or their 'inverse' Daughter/Son). By linking these singular relations by their common nodes, we were then able to create the network that is the Graph List. Now that we have all that information grouped up, we want to extract all the possible combinations of non-singular relations, between certain limits which will be detailed more ahead.

4.2.1 Structure

To extract all the information that we need, we need to create specific structures. The fact that we don't know how many relation groupings we might have makes it so that we want once again a structure that we can expand upon. So, we decided to use a list for that end. This time, the structure will need to hold different nodes, each symbolizing a different relation grouping. We also need to store in each node the relations that make up the grouping, as well as the order that they appear in. Applying a similar design from the first module, the following structure was created:

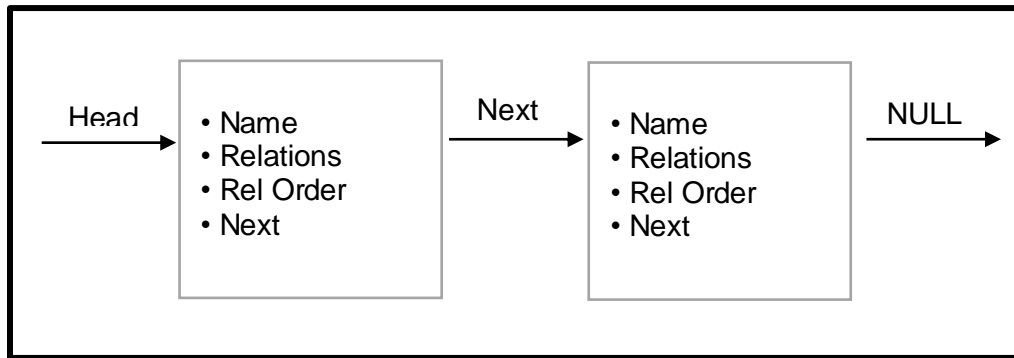


Figure 5 - Group List

Figure 5 represents the Group List, which is responsible of holding all the relationship groupings that we will encounter. Each node has a 'Name', which is generated upon its creation. The names are on a RelX format, with X being an incremental number, resulting in the first node being called Rel1, the second Rel2, until we learn all the N groupings of a Graph List, resulting in RelN being learnt. 'Relations' is a variable that shows the number of relations that are in a specific grouping. 'Next' is a pointer to the next node, while 'Rel Order' is a list that holds all the relations of the grouping, as well as their order. The structure is shown on Figure 6.

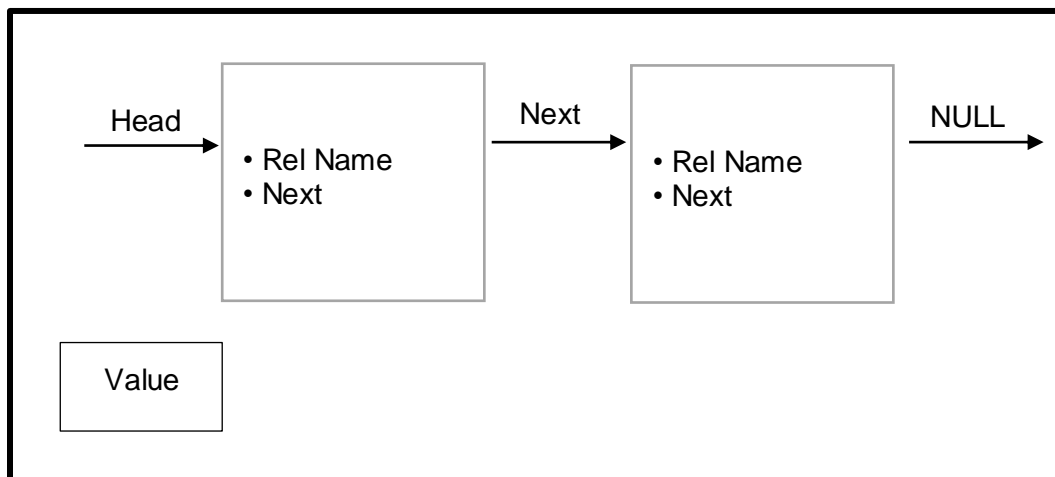


Figure 6 - Rel List

This time, the list itself has a linked 'Value' besides its list of nodes. This value is updated more ahead in the processing of the list, and on the third module, for the higher-level learning. It will be a flag to indicate if the content of a grouping is only made of 'regular' relations, only made of 'inverse' relations, or a mix of both. The nodes have a 'Rel Name', which is the name of the relation, and a 'Next' pointer, pointing towards the next node.

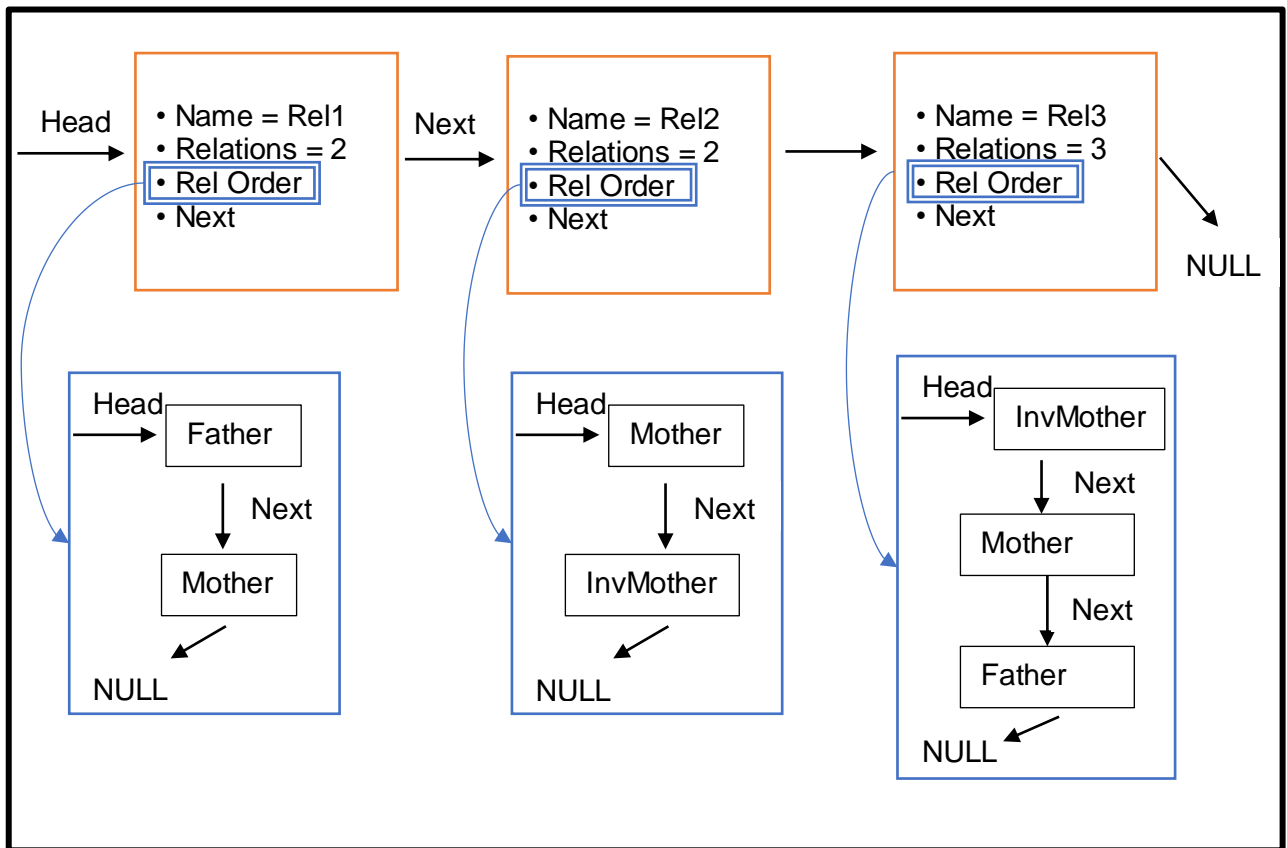


Figure 7 - Concrete Group List

Figure 7 helps us put all the pieces together, into a coherent structure: we have a Group List, which is responsible of holding all the relationship groupings that we will find on a given Graph List. A relationship grouping is a set of two or more successive relations found on the graph, resulting of an aggregation of multiple singular relations. The Group List saves all these groupings, without repetition. Each Group Node (orange blocks) generates a 'Name' for its newfound groupings (in this case, Rel1, Rel2 and Rel3) and saves the number of 'Relations' (respectively 2, 2 and 3 relations) that the grouping has. The relations as well as their order (Father->Mother for Rel1, Mother->InvMother for Rel2 and InvMother->Mother->Father for Rel3) are saved in a Rel List structure (blue blocks). The Rel List saves a set of nodes, each node having a 'Rel Name' in it (Rel Names being the relations names).

We can compare our objective in this module to combinatorial analysis. We want to get all the possible combinations of relations that we can observe on Figure 1, and save them on a Group List, with each node saving a single combination.

This set of structures is solid enough that it allows us to save all the important information that we need in order to process the Graph List and extract all of our desired information into an organized and simple format for us to understand, and later work on. Now that the inner workings of the structures have been clarified, we can now move on to the main code of the second module.

4.2.2 Execution

This subsection will now explain in detail how the code is executed through examples and pseudo code. The way that module 2 works is straightforward and doesn't need much setup. The function that we use to discover the relation groupings is called `discoveryFunction()`.

Before we get into details about how the function itself works, we need to take a step back and remind ourselves of our goal. We want to obtain all the relation groupings in a given Graph List, where the size of the groupings will be ≥ 2 up to a certain limit. The limit itself currently needs to be decided by the user. For our testing on the Figure 1 graph, we used an upper limit of 4. While the upper limit can be anything that the user desires, a good recommendation for a 'default' minimum value is the maximum number of steps which would take to reach from the higher node on the family tree, down to the bottom one. In Figure 1's case it would be 2, but we picked 4 to see how the program would handle hopping between different nodes that are not necessarily transitive (which in this case would be something different than a parent or grandparent, since the example that we are working with only has 3 generations throughout its 6 nodes).

The pseudo code for the relationship groupings is as follows. Assuming that we have an already setup Graph List called 'graphList' and an upper limit called 'maxScope' = 4, Figure 8 shows the code that is executed.

```

01  create a Group List called groupList;
02  create a Rel List called relList;
03  create a Group Node called groupNode;
04  create a char variable temp = "1";
05  create an int variable i = 2;
06
07  while i <= maxScope
08      groupNode = head of graphList;
09      while groupNode is not NULL
10          groupList = discoveryFunction(groupNode, i, groupList, relList, temp, graphList);
11          groupNode = groupNode->node;
12          i = i + 1;

```

Figure 8 - Module 2 initial execution

This block of code guarantees that we get all the relation groupings that we need. The inner While cycle makes sure that we iterate over all the nodes in the Group List, while the outer cycle makes sure that we execute that same piece of code for grouping size of 2, 3 and 4. This adds all the different groupings to the Group List in a sorted way, which is useful for later operations.

The code calls the `discoveryFunction()` a total of three times (for this example). This function is a recursive function, meaning that it repeatedly calls itself until the conditions to do so aren't met anymore. It works in a similar way as a cycle, but more convenient due to the way it executes in this scenario, sharing some of the structures, and creating new ones when needed.

To support this function, there is a small group of auxiliary functions which we will also mention. Such functions turn the code more legible, avoiding repeated instances of code and turning the `discoveryFunction()` significantly more compact.

The `discoveryFunction()` is defined as `discoveryFunction(node, currentCycle, groupList, relList, relName, familyList)`. 'node' is the current node we are iterating on, 'currentCycle' is the current grouping sizes we are searching for, 'groupList' is our end goal, slowly being worked upon. 'relList' is a temporary list which saves the group of relations we are exploring, 'relName' is the relation we are considering to add on the current step to the 'relList' and 'familyList' is a pointer to the initial Group List. The pseudo-code of the function is shown on Figure 9.

```

01  create a List Connect Node named tempConnectNode;
02  tempConnectNode = node->connected->head;
03  set node->visited to 1;
04
05  if relName is not "1"
06      if relList->head is NULL
07          then relList->head = newRelNode(relName)
08      else
09          create Rel Node called tempRelNode;
10          tempRelNode = relList->head;
11          while tempRelNode has next
12              tempRelNode = tempRelNode->next;
13          tempRelNode->next = newRelNode(relName);
14
15
16  while tempConnectNode is not NULL
17      if currentCycle > 0 and tempConnectNode hasn't been visited
18          groupList = discoveryFunction(tempConnectNode,    currentCycle-1,
groupList, relList, tempConnectNode->relName, familyList);
19      else if currentCycle = 0
20          if relList exists in groupList
21              set node->visited = 0;
22              remove last node from relList;
23              return groupList;
24          else
25              set node->visited = 0;
26              concatenate groupList with newGroupNode(relList);
27              remove last node from relList;
28              return groupList;
29      tempConnectNode = tempConnectNode->next;
30
31
32  set relList->head to NULL;
33  set node->visited to 0;
34  return groupList;

```

Figure 9 - Module 2 initial execution

On this function, we have 3 major parts of the code. Tackling them one at a time, we have: initialization of variables and setup of the relList (lines 01-13), the while cycle (lines 16-29), closure and return of the function (lines 32-34).

First, we create a variable called 'tempConnectNode'. This is a temporary node which we use to iterate over a node's adjacency list. Then, we proceed to mark the current node we are in right now as visited. The next block of code regards the relList. Initially, the 'relName'

is set as “1” from our initial execution. So, for now, we can safely ignore this part. Should the ‘relName’ be different than “1” (which happens for every single case after the first), depending on whether the ‘relList’ is empty or not, it either creates a new Rel Node and set it as ‘relList’->head, or creates a new Rel Node and adds it to the end of the ‘relList’.

The second part of this function deals with most of the processing that goes into analysing and managing the ‘groupList’ itself. First, it checks if we haven’t reached for the end of the adjacency list, as we iterate over it. If not, it tests whether ‘currentCycle’ is zero and if the adjacency node we plan to test hasn’t been visited yet. If both conditions are met, we call the function once more, with the adjacency node we are currently evaluating as ‘node’, decrease the ‘currentCycle’ by 1, and pass the relation’s name of the adjacency node as ‘relName’. We can consider the ‘currentCycle’ as some sort of step counter. If its value isn’t zero, we can call the function once more, guaranteeing that we can only “stretch” as much as the initial value of ‘currentCycle’ allows us to. Should it be zero instead, we evaluate the information that we currently have. We test whether the ‘relList’ that we have built up until now exists in ‘groupList’ via the usage of an auxiliary function. Should it exist, we want to ignore this path. We set the node we currently are as not visited; remove the last node we added to the ‘relList’ and return an unchanged ‘groupList’. However, if the ‘relList’ didn’t exist in the ‘groupList’, then we create a new Group Node from that ‘relList’ and add it to the end of the ‘groupList’. We make sure to set the current node we are in as not visited, remove the final node of the ‘relList’ and return an updated ‘groupList’. Finally, when the moment comes that we iterated over all the adjacency list of the original node, the cycle breaks. We set the ‘relList’ head to NULL, and the current node (which should be the initial one) to not visited before returning the ‘groupList’. We do this because the discoveryFunction() iterates over the adjacency list of a node. Of course, it visits other nodes, as to obtain the groupings of a certain size, yet all those groupings have the initial node as a start. In our main() function are two cycles, as we seen in Figure 8. The inner one is responsible to execute the discoveryFunction() for a certain grouping size on all of the nodes of the Group List. This means that if we don’t clear the ‘relList’, after the full execution of the first node, we will have a single relation in our ‘relList’, which is NOT what we want. It will get worse from there, as each call to the recursive function on our main() will add another relation to the list. Similarly, we set the initial node as not visited so that there are no remains/traces from previous executions.

This function uses an iterative deepening algorithm search, where its visual representation can be seen on Figure 10. If we start from the 'Ken' node (having set the visited at 1) while looking for groupings of size 2, we start by exploring our single adjacency for the node: 'Harriet'. We note the relation's name of the relation we used on our 'step', which would be 'Father'. Now on the node Harriet, we set it as visited and iterate over the node's adjacency list. Assuming that we see a relation to 'Ken' as the head of 'Harriet's Rel List, we ignore it, since we had set the node's 'visited' variable to 1. We then check the next adjacency, which could be to 'Laura'. Since it wasn't visited, we explore that adjacency, adding the relation's name 'InvMother' to the 'relList', which is currently ['Father',InvMother']. At this point, the number of steps will be zero, moment which we compare our current 'relList' and see if it exists in any node of the 'groupList'. Since it doesn't, we create a new node and add it to the list. We set the 'visited' of 'Laura' to 0 and remove the final node from the 'relList', returning it to the state it was when we first arrived at Harriet. We then repeat this process for 'Alice' and 'Bob', adding ['Father','Mother'] from Ken-Harriet-Alice, but not for Ken-Harriet-Bob, as it already exists in the 'groupList' from the previous step. This will conclude the cycle for 'Ken', yet the inner cycle of the main() function will repeat this process for all nodes. And once that is done, it will complete all of this process for a number of steps higher than the previous execution (assuming we haven't reached the max limit).

On Figure 10 we can see the example just described. Blue lines represent the relations which we can use to reach other nodes, green arrows represent an extension of the discovery function and red arrows represent the backtracking. Additionally, green arrows add the relation's name to the 'relList', set the 'visited' bit to 1 and decrease the 'currentCycle' by 1, while red arrows remove the relation's name from the 'relList', subtract the relation's name, set the 'visited' bit back to 0 and increase the 'currentCycle' by 1.

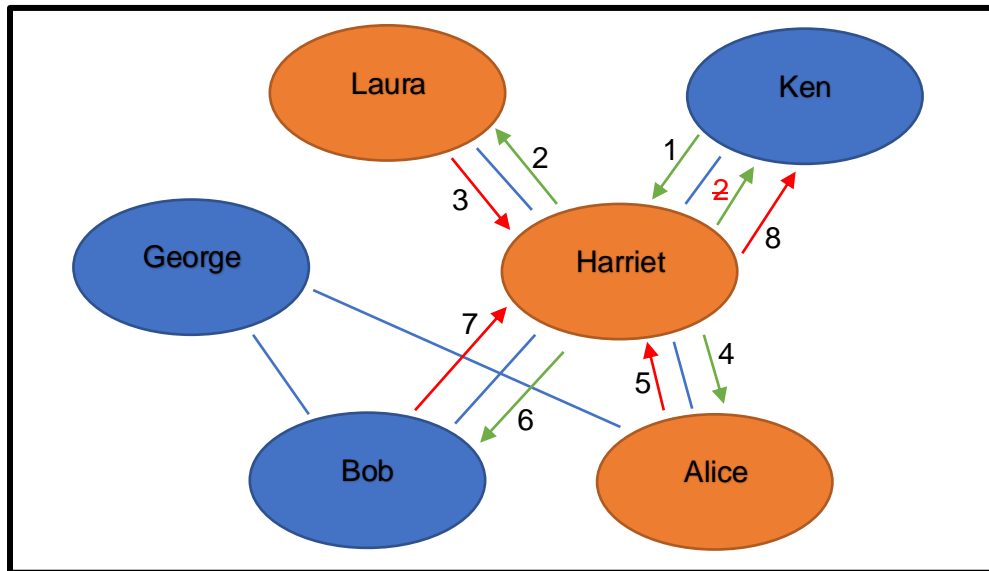


Figure 10 - Discovery Function Trace

Each movement of the function is numbered according to how it executes. The order by which the function visits adjacent nodes is entirely dependent to the order of which the nodes were added there in the first place. However, the order doesn't matter for our purpose as we end up visiting all possibilities, regardless of where we start.

As far as auxiliary functions go, there are a couple of worthy mentions. Other than functions that help in creating structures themselves, there is `testRelList()` which is responsible of iterating over the 'groupList' and discover if a given 'relList' is already present. The `concatGroupList()` is the function that adds a Group Node on the 'groupList' (setting it as head if the list is empty), while also making sure to name the 'groupNode' correctly, according to the number of the node.

There is also one final function, `copyRelList()`, which copies the content of the 'relList' that we use to keep our path. This function is called when we wish to create a new Group Node and obtain a new 'relList' for that node. We can't exactly copy the original Relation List, since we would copy the pointer to the address in the memory, and as the list would change as we would iterate over the Graph List, so would all the pointers in all the Group Nodes.

By the end of the execution of the second module of the code, we will have the structure represented in Figure 11.

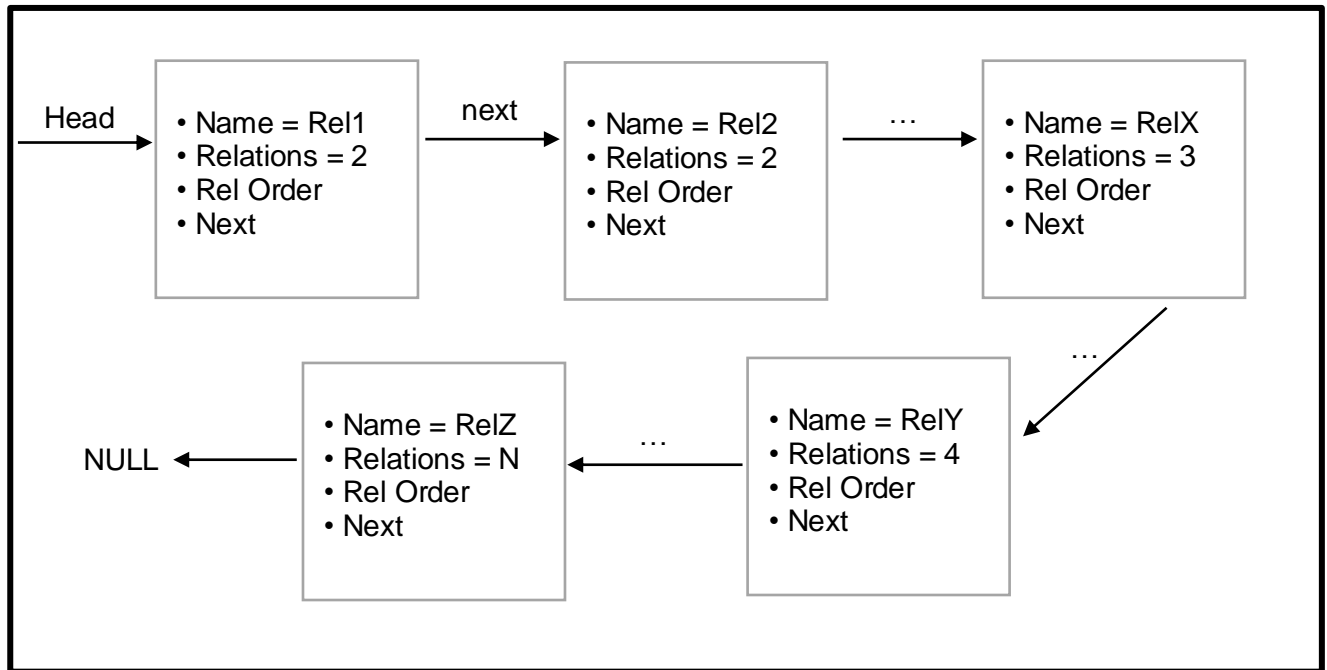


Figure 11 - Second Module Output

Assuming we executed the code with a starting 'currentCycle' of 2, and with an upper limit of N, we have as a result a 'groupList' which contains all of the different relationship groupings observable in the Graph List, in a binary fashion (this limitation will be explained more ahead). As seen in Figure 11, we have (X-1) groupings of 2 relations, (Y-X) groupings of 3 relations and so on, until we reach the final set of groupings, comprised of groupings with N relations.

The limitation that was mentioned on the previous paragraph, referred to the fact that due to how the search mechanism worked, we can only obtain relations which are in essence, paths between two nodes. Assuming a general familiar context, we can have a relationship where a node A is a parent of node B, and the only restriction for this to be true, is that there needs to be a relation connecting these two nodes (in this case could be either mother or father), and nothing else. If we try this idea on higher level concepts, such as grandparents, while we are executing more steps, the idea remains the same. We have a linear path all the way from the node A (which will be the grandparent) all the way to the node C, which will be the grandchildren. There will be some degree of transitivity to it, as we need to have parent(A,B) and parent(B,C) to have grandparent(A,C), however our search function will be able to get there with no issues due to jumping from node to node the way that we explained in Figure 10.

Now, let's have a more complex example. How would our program be able to identify siblings? We can identify a node A being a sibling to a node C, if $\text{parent}(B,A)$ and $\text{parent}(B,C)$ are both true. Here lies a problem to the identification of non-transitive relations. If our program can only extend to one node at a time, how will it be able to identify these two relations and thus learn the sibling relationship? The answer lies in the In/Out and 'inverse' relations that we mentioned previously on our first subsection. We will mention them in more detail on the third subsection, as the third module will be responsible for the higher-level learning, but for now, it suffices to say that non-transitive relations are able to be observed and learnt upon by our program.

If we think about the sibling's example, we can locate one easily on Figure 1, with 'George' being the father, and 'Bob' and 'Alice' being the children. In this case, if we consider George as a starting node, we can classify those three nodes as not only a graph, but also as a binary tree. And that's where the limitation of the program lies at. It can only learn new information from binary tree structures. When the term binary tree is used here, we do not mean the whole graph in itself, but rather, the starting node where the program executes the second module from. As we explained previously, the search mechanism works in a similar fashion as a depth search algorithm. That same algorithm will be repeated over each of the N individual nodes, which in essence, will be a depth search over N different binary trees.

To be more concrete about what our program can't handle, let's assume there is a term for a sibling relationship with specifically 3 children, called superSibling. It will be something akin to a triplet, but not necessarily born at the same time. $\text{superSibling}(A,B,C)$ is true if $\text{parent}(D,A)$, $\text{parent}(D,B)$, $\text{parent}(D,C)$ are all true. However, our program will only be able to process at most 2 out of 3 relations (depending on how the graph was built, it could either go ADB, ADC, BDA...but no matter what path it chooses, one relation will always be left out). While it is a valid concern, which will be addressed in chapter 5, it is enough for the moment being to solely learn from binary trees.

4.3 Common factor detection and hierarchical learning

On the first and second module we processed our BK and data in different ways that allowed us to obtain data structures that were important to set up our end goal. On our first module, we organize our BK into a graph, so that we can iterate over the set of nodes. On the second module, we take advantage of the relation 'network' and are able to obtain all the possible relation groupings of different sizes, all of them saved in a sorted list. For the third module, our objective finally aligns with this thesis' objective: Hierarchical Learning.

With all the pre-processing concluded, in this subsection we use the 'groupList' and its different groupings to build a library of different possible relations. Just like the previous modules, we start by creating a structure to hold the knowledge of the different rules that we will learn, and then analyse the code which allowed us to obtain the Hierarchical rules.

4.3.1 Structure

When thinking of how we wanted the structure to be built, we first had to think about how the rules themselves would look like. If we are to learn $\text{grandparent}(A,C)$, we will need to keep the information that $\text{grandparent}(A,C) :- \text{parent}(A,B), \text{parent}(B,C)$. And if we want to go one level above that, $\text{grandgrandparent}(A,C) :- \text{grandparent}(A,B), \text{parent}(B,C)$. Once again, we do find a binary pattern, but we can't discard the possibility that the structures we thought so far were only binary due to them being transitive. As we seen on the previous module, the siblings relationship also had two conditions, but what about an uncle, for example?

$\text{uncle}(A,D) :- \text{parent}(B,C), \text{parent}(C,D), \text{parent}(B,A)$. In other words, A will be uncle of D, if there is a node B which is grandparent of D, and that same node B is also parent of A. This makes perfect sense, and if we think about it further, the three initial parent conditions can be once again simplified to two. However, there is also a second way to think of this problem: A will be uncle of D, if A and C are siblings, and if C is parent of D. As for which of the two definitions will be the one that the program chooses, will of course depend on how the initial graph was created, but Figure 12 is able to sum up these two situations in a very clear way.

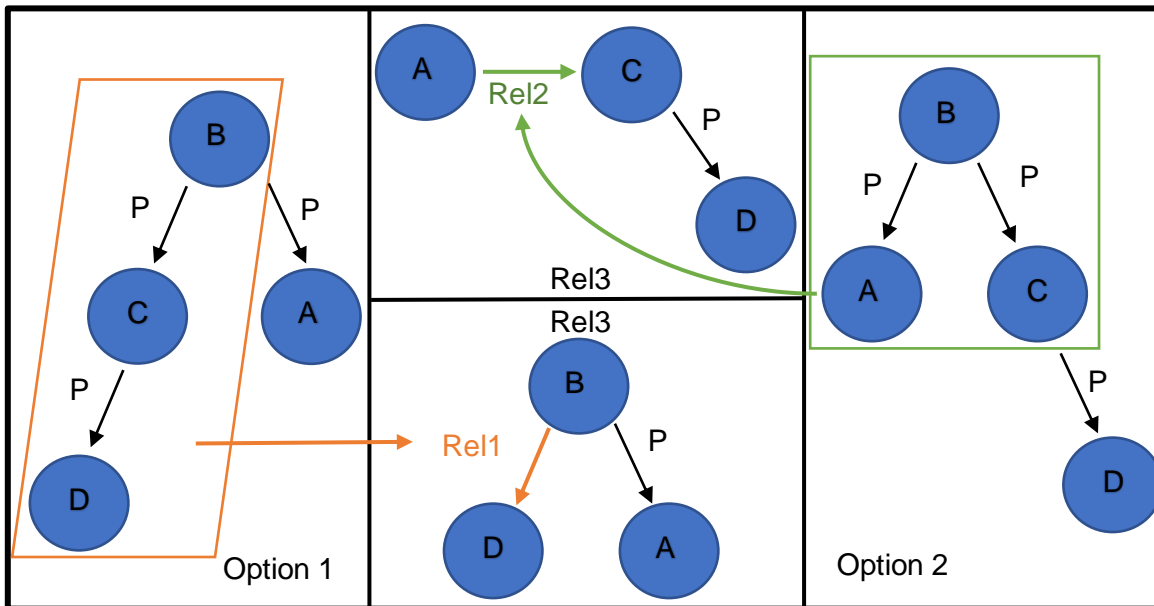


Figure 12 - Hierarchical Learning Alternatives

In this Figure, the nodes are individuals of a family, and P is a relation of parent (either mother or father). On both option 1 and 2 the family tree is the same, just mirrored. The yellow and green outlines represent the simplification of a grouping (of two relations) into a single relation, called Rel1 for Option 1 and Rel2 for option2. As mentioned on the previous paragraph, there are two ways of representing the uncle relation. The program might learn the grouping [Parent,Parent] on the left as Rel1 (Rel1 being the grandparent relation) and Rel3 (which will be the uncle relation) will be defined as Rel3:- Rel1,Parent (A uncle D, if B grandparent D, and if B parent A). However, if the program chooses to first learn the grouping [Parent,Parent] on the right as Rel2 (Rel2 being a brother relation), Rel3 will be defined as Rel3:- Rel2,Parent (A uncle D, if A sibling C, and if C parent D).

Depending on what relation the code learns first (Rel1 or Rel2), the resulting Rel3 will also be different. However, before moving on, there are two important points we need to make: first, this is not an issue, since that given a big enough graph, both scenarios will be added regardless. Second, this confirms that non transitive relations are not an issue for us to turn into the format RelX₁:- RelX₂, RelX₃.

Now knowing the kind of format that the structure needs to take, and what arguments to keep, we can proceed and create our list to hold the learnt relations:

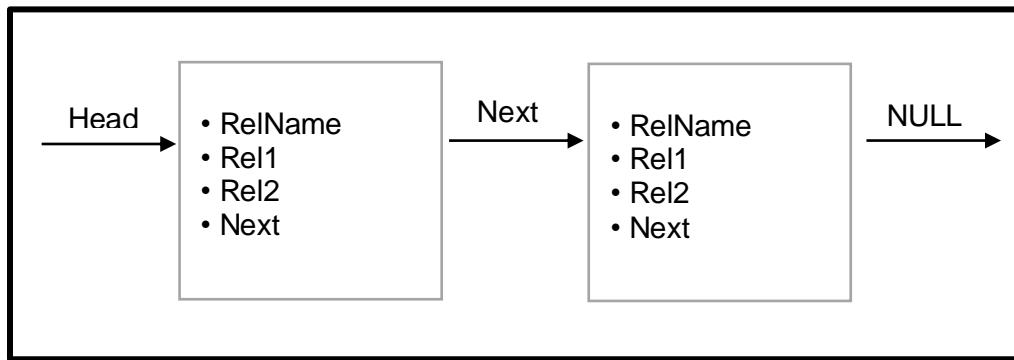


Figure 13 - Join List

This time, the structure detailed on Figure 13 is much more straightforward, and opposed to the other two modules, doesn't hold a list in each node. We save 'RelName' as the name of the learnt relation, in a format similar to how we learnt the ones in the Group List, however this time, the nodes follow the 'learnX' format, with X starting from 1 and increasing each node. 'Rel1' and 'Rel2' are the two relations that make up the new relation.

Now that we have a standardized format for all the relations that we want to learn, we now move into the execution section of the code.

4.3.2 Concepts and execution

We now have all the pieces that we need to progress into our final step, which is the writing of the function that learns relations recursively. So, our current objective is, from the Group List that we created, process those groupings of relations into rules of binary components, as we have mentioned previously. To that end, we need to split the execution part into two steps.

The first one is to try and find a common factor between the groupings of size two that we have. This is so that we can simplify the learning process. In our family's case, we learn a relation that would be parent, which was previously mentioned in this document. As a reminder:

`parent(A,B) :- mother(A,B) ; father(A,B).`

In other words, A is parent of B if A is mother of B, or if A is father of B. We can learn this by observing our groupings. We take the groupings in pairs and compare the first and second value of each grouping. Then, if we see similarities, we can simplify the common factor and thus create a new rule from this observation. More concretely, for this example we observe something as seen on Figure 14.

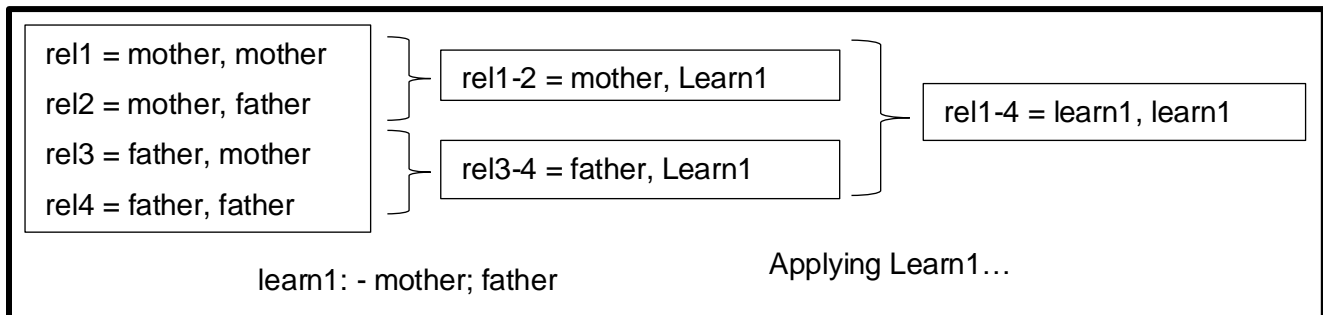


Figure 14 - Common Factor Analysis Concept

In the first step, we identify that mother (in rel1 and rel2) and father (in rel3 and rel4) are common elements. We select them, and create a new rule, called learn1 (which is the parent relation). From four relations, we now are down to two, however we can see that this time, we have once again a common factor. learn1 is the common factor, and the mother and father are already part of another relation: learn1 itself. So, from four relations, we can compress them into one, which will heavily simplify our relation learning.

There is, however, one small issue that we need to address. As we added the relations to the graph on the first module, we also added the 'inverse' ones, invFather and invMother. This is an issue as it could mess up with our analysis. If we consider the fact that we have 4 different singular relations (mother, father, invMother, invFather) instead of two, then we have more than 4 different groupings of size two. We have instead, 16 different combinations, and we are able to learn little from it. But once again, that's not a problem since father/mother aren't different than invFather/invMother.

While this is a bold statement, the addition of the 'inverse' relations is not a change that is made to the relations in the graph itself, as in their meaning isn't that different from a father/mother. As far as the learning process is concerned, father/mother are similar relations to invFather/invMother, because in the end, they are the same relation, linking the same nodes. The only thing that changes, is the direction and their name, which only happens as to allow us to learn nontransitive relations. To better understand what we mean, we need to take a look at Figure 15.

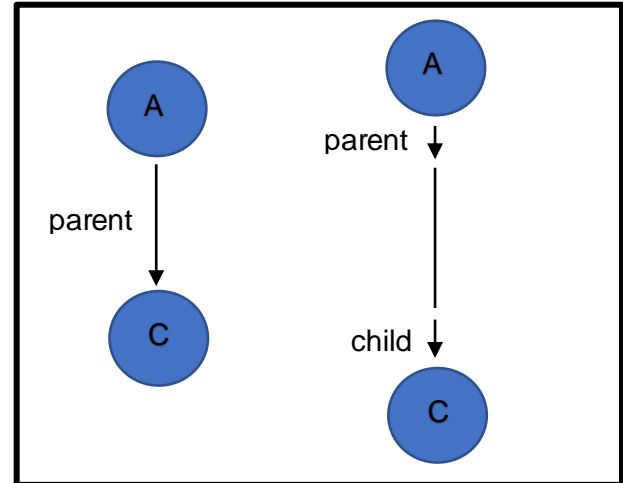


Figure 15 - Bridge Analogy

Instead of nodes, however, let's think of A and C as gates to castles, while what links them is a bridge. A and C share that same bridge. The bridge that connects both gates is the same. However, the terms 'leaving' and 'exiting' differ on where we are. So, for example, if we go on the AB direction, when we pass A gate, we are 'exiting' that gate, while when we reach B gate we are 'entering' that gate. Entering and leaving are two faces of the same coin. Now, if we are to apply the same logic on our nodes, from A's point of view, they are B's parent. And that's because they are on the outgoing perspective on the relationship. B, on the other hand, is the Child of A since they are on the incoming perspective of the relationship. This is the same situation as of the bridge example. The terms are different (within the code), but the bridge is there, and is the same for both nodes.

There are two reasons for this to happen. First, is due to implementation. There is no variable which represents a relation father/mother on our code as some sort of intermediary node between two nodes. The relation itself is part of a node, and an inverse version of that same relation is found on the linked node. The second reason is that, by implementing it this way, we can learn nontransitive relations. If we take once again the example of learning siblings'

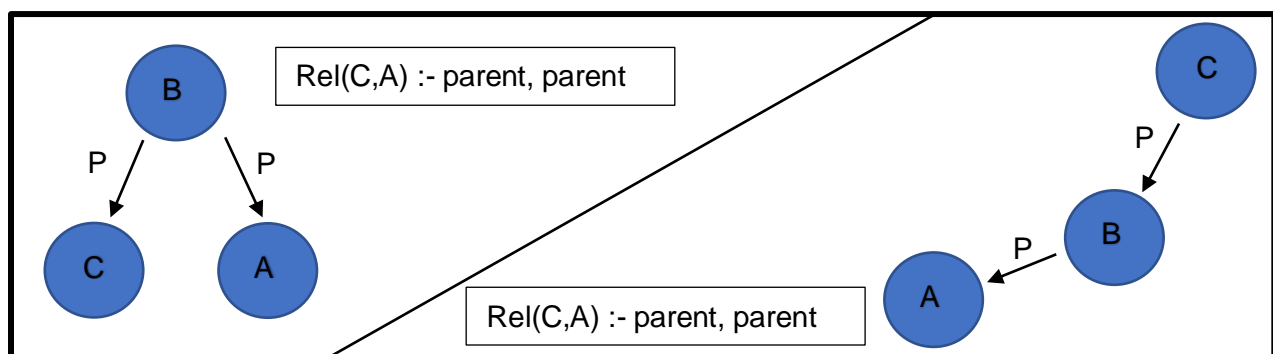


Figure 16 - Learning without perspective

relations, we conclude that to learn that A and C are siblings, we need to know that there is a B node such that B is parent of A, and B is parent of C. If we had implemented the relations in a unitary way, which no matter the direction, the relation's name would always be the same, we would end up with Figure 16's scenario. While the relation of C and A on the left is one of siblings, the second module would save that grouping the same way that it would save a grouping of a grandparent relation (example on the right, where C is A's grandparent).

On the other hand, if we are to add 'perspective' on the two nodes that are connected by the relationship, we end up getting very different results.

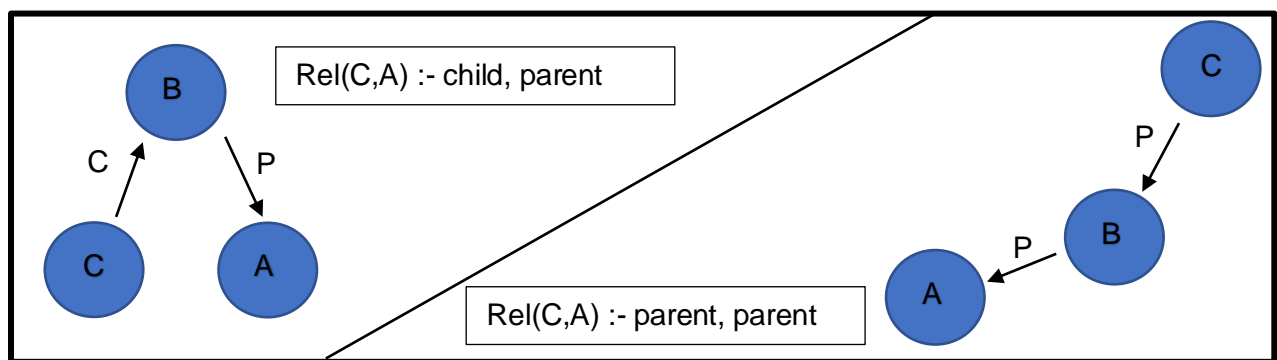


Figure 17 - Learning with perspective

As shown in Figure 17, both cases were processed into new relations on the second module of the code, assuming that they took the path C->B->A. We now know that the 'child' relation (which is for invFather/invMother what parent is to mother/father) means that there is a shared node for the two relations, allowing us to invert the relation name back to the original, and describe a siblings relationship, where we previously couldn't. While Figure 16's result was the same relation for both cases, in Figure 17 we now have two different relations added to the Group List.

Now that we explained why it is important for both mother/father, as well as invMother/invFather to exist, we need to fix the problem related to the analysis of 'inverse' relations. The solution is trivial. As we are building the Group List on module 2, when we create new group nodes, we made sure to keep a counter, which initializes on 0. Each mother/father that is added to its 'relList' increases the counter by 1. Each invMother/invFather decreases the counter by 1. Once the 'relList' is added to the group node, we compare the variable 'relations' from the group node (which keeps track of the size of its 'relList') to the counter. If the 'relations' is equal to the counter, that group node only

contains mother/father relations. If the 'relations' is equal to -counter, we know that the group node only contains invMother/invFather relations. Any other value means a mix of those relations. We then set the 'value' variable of the 'relList' as 1, -1 or 0 depending on these three cases. Then, we simply compare group nodes with the same 'value'. That guarantees that we can analyse common factors for mother/father, and their 'counterpart' invMother/invFather independently.

This first step is executed while the Join List is created, and the two first nodes are then added. The function that creates the Join List is createJoinList(groupList), where groupList is the groupList resulting from module 2. The code is as shown on Figure 18.

```
01  create Join List called joinList;
02  create Join Node called joinNode;
03  create Group Node called groupNode1;
04  create Group Node called groupNode2;
05  groupNode1 = groupList->head;
06  groupNode2 = groupNode1->next;
07
08  while groupNode1 exists and its grouping size is 2
09      while groupNode2 exists and its grouping size is 2
10          if groupNode1's value is different than 0 and it is equal to groupNode2's value
11              joinNode = commonFactorAnalysis(groupNode1,groupNode2);
12              if joinNode isn't NULL
13                  add joinNode to joinList
14              groupNode2 = groupNode2->next;
15          groupNode1 = groupNode1->next;
16          groupNode2 = groupNode1->next;
17  return joinList;
```

Figure 18 - Join List Creation

The common factor analysis is shown on Figure 19.

```

1    create Join Node called joinNode;
2    create Rel Node called relNode1;
3    create Rel Node called relNode2;
4    create a string called string1;
5    create a string called string2;
6
7    relNode1 = groupNode1->relOrder->head;
8    relNode2 = groupNode2->relOrder->head;
9
10   if relNode1->relName is the same as relNode2->relName
11       joinNode = createJoinNode(relNode1->next->relName, relNode2->next->relName);
12       return joinNode;
13   else
14       copy relNode1->relName into string1;
15       copy relNode2->relName into string2;
16       relNode1 = relNode1->next;
17       relNode2 = relNode2->next;
18       if relNode1->relName is the same as relNode2->relName
19           joinNode = createJoinNode(string1,string2);
20           return joinNode;
21   return NULL;

```

Figure 19 - Common Factor Analysis

Once that the createJoinList() is executed, we have two initial nodes in it. The order depends on the order of the relations from the second module, but we should have learn1 :- mother ; father and learn2 :- invMother ; invFather. There is a small detail that we need to be aware of. These two initial relations are a disjunction, while the other relations that we learn are be a conjunction. This happens because what we do with the common factor analysis is to point out that learn1 is either a mother or a father. With future relations, for example grandparent, we conclude that learn3 is a parent of a parent. Both need to be true; it is not an option as in the previous case.

Now that the first preparatory step is done, we still need to do one last thing before we execute the function that takes care of recursively learning the relations. We need to iterate over the Group List and swap every single mother/father and invMother/invFather' by learn1 and learn2. The function is trivial, only iterating over the Group List. On each 'groupNode' it iterates over the 'relList' and through an if statement, we identify which of the two cases we have in that Rel Node and simply redefine the 'relName'.

Once all the relation names are swapped, we simply need to create the function that automates all the learning process. The function itself is called `pandora()` and is supported by a group of auxiliary functions. Initially, we first observe the Group List. If we go back to Figure 11, we know that the output of the second module is a list of groupings, sorted on their grouping size. All the relations have been simplified as we just described, but that step didn't change the groupings size whatsoever. We create a pointer to the first node of the Group List and we iterate over it. On each Group Node, we create two pointers: one to the first node of its Rel List, and another to the node after that. We then call an auxiliary function to try and locate this pair of relations on the Join List nodes. If we are successful, the auxiliary function returns the Join Node's 'relName'. If the pairing doesn't exist however, `pandora()` creates a new Join Node in the Join List with the pair being used as 'rel1' and 'rel2'. When it runs those arguments once more, we have a match, called `learnX`. The function then proceeds to edit the 'relList' where it learnt the function. The first Rel Node is cut off, with the head of the Rel List pointing at the second node. We edit the 'relName' to be equal to `learnX`, and decrease the 'relations' of the Group Node by one, in accordance with the change of size that we made. Now, once `pandora()` executes its code once again, it confirms if 'relations' is bigger than 1. If it is, it continues learning the relations from the Group Node, until its size is only 1, with its relations being simplified down to only one relation.

All of these learned relations are saved on the Join List, which acts as a 'library' of sorts, while that same knowledge is applied, and at the same time extracted, from the Group List. The function `pandora(groupList, joinList)` has the known 'groupList' and the previously created 'joinList' as arguments. Its code is detailed on Figure 20.

```

01  create a Group Node called groupNode;
02  create a Rel Node called relNode;
03  create three strings called rel1, rel2, tempRel;
04  groupNode = groupList->head;
05
06  while groupNode isn't NULL
07      if the size of groupNode is 1
08          groupNode = groupNode ->next;
09      else
10          relNode = groupNode->relOrder->head;
11          copy relNode->relName into rel1;
12          relNode = relNode->next;
13          copy relNode->relName into rel2;
14          execute getJoin(joinList, rel1, rel2) and copy its output into tempRel;
15          if tempRel isn't NULL
16              create joinNode and add it to joinList;
17          else
18              run cutHead(groupNode->relOrder, tempRel);
19              decrease groupNode->relations by 1;
20  return joinList;

```

Figure 20 - Pandora Function

With `getJoin()` (line 14) being the function that checks on a given 'joinList' if both 'rel1' and 'rel2' are present in any node, and `cutHead()` (line 18) being the replacer function that removes the first node of a Rel List and swap the second node's relation by 'tempRel'.

As `pandora()` progresses through the Group List, it first learns from basic 2 sized groupings, then uses the relations from those groupings to learn on the groupings of size 3 and so on, until all of the Group List nodes have been simplified to a single relation each.

5 Results

In this chapter we present the results for different execution cases, indicating the facts that were used as knowledge, a rough calculation of their execution time, and we also decided to test all those cases with another way of iterating through the Graph List.

In chapter 4, we pointed out that we had considered two ways of executing our program, which were the call of an iterative function over the Graph List each time we wish to get the reference of a node, implying the usage of the node names as keys as way of search, or creating an adjacency pointer on the adjacency list instead of the name, which would point us directly at the target node. We start by observing the results on the initial method that we picked to iterate: the iterative function. All these tests were executed in a Virtual Machine, running Ubuntu 64-bit, with 10gb of RAM, and 60GB of storage. The program itself was compiled using GCC and executed through the Linux terminal. Table 2 and Table 3 display the execution time in seconds. Module 1, Module 2, and Module 3 refer to the three modules of the code, described in the fourth chapter, with the total runtime of the program in the final column. As for the relevant arguments used in the program, all executions were done with a maximum number of steps of 4.

5.1 Method 1 - Iterative Function

5.1.1 Input and output

The first test involves using the knowledge that we used to build the Graph List from Figure 1 as control. The facts were only 6, as seen on Figure 21.

| | |
|----------------------|-----------------------|
| Father(ken,harriet) | Mother(laura,harriet) |
| Father(george,alice) | Mother(harriet,alice) |
| Father(george,bob) | Mother(harriet,bob) |

Figure 21 - Control Input

The learnt relations from this input are depicted on Figure 22.

```
Learn: Learn1, arg1: Father , arg2: Mother
Learn: Learn2, arg1: InvFather , arg2: InvMother
Learn: Learn3, arg1: Learn1 , arg2: Learn2
Learn: Learn4, arg1: Learn1 , arg2: Learn1
Learn: Learn5, arg1: Learn2 , arg2: Learn2
Learn: Learn6, arg1: Learn3 , arg2: Learn1
Learn: Learn7, arg1: Learn3 , arg2: Learn2
Learn: Learn8, arg1: Learn7 , arg2: Learn2
```

Figure 22 - Control Output

With 'Learn' being the relation's name, 'arg1' being the first argument in the learnt relation, and 'arg2' being the second argument.

The second test involves using a slightly larger family, described by 18 facts, shown in Figure 23.

| | | |
|-----------------------|------------------------------|----------------------------|
| Father(ken,harriet) | Mother(alice,penelope) | Father(james,colin) |
| Father(george,alice) | Father(andrew,james) | Mother(christine,james) |
| Father(george,bob) | Father(andrew,jennifer) | Mother(christine,jennifer) |
| Mother(laura,harriet) | Father(christopher,arthur) | Mother(penelope,victoria) |
| Mother(harriet,alice) | Father(christopher,victoria) | Mother(victoria,charlotte) |
| Mother(harriet,bob) | Father(james,charlotte) | Mother(victoria,colin) |

Figure 23 - Control+ Input

With the output shown in Figure 24.

```

Learn: Learn1, arg1: Mother , arg2: Father
Learn: Learn2, arg1: InvFather , arg2: InvMother
Learn: Learn3, arg1: Learn1 , arg2: Learn2
Learn: Learn4, arg1: Learn1 , arg2: Learn1
Learn: Learn5, arg1: Learn2 , arg2: Learn2
Learn: Learn6, arg1: Learn3 , arg2: Learn1
Learn: Learn7, arg1: Learn3 , arg2: Learn2
Learn: Learn8, arg1: Learn4 , arg2: Learn2
Learn: Learn9, arg1: Learn4 , arg2: Learn1
Learn: Learn10, arg1: Learn5 , arg2: Learn1
Learn: Learn11, arg1: Learn5 , arg2: Learn2
Learn: Learn12, arg1: Learn6 , arg2: Learn1
Learn: Learn13, arg1: Learn7 , arg2: Learn2
Learn: Learn14, arg1: Learn8 , arg2: Learn1
Learn: Learn15, arg1: Learn10 , arg2: Learn2
Learn: Learn16, arg1: Learn7 , arg2: Learn1
Learn: Learn17, arg1: Learn8 , arg2: Learn2
Learn: Learn18, arg1: Learn6 , arg2: Learn2
Learn: Learn19, arg1: Learn11 , arg2: Learn1

```

Figure 24 - Control+ Output

The third test scenario involves removing a single relation from Control+ input, which acts as a 'bridge' between two sub families. These subfamilies will act as clusters, independent from one another, and the input to such a scenario is depicted on Figure 25.

| | | |
|-----------------------|------------------------------|----------------------------|
| Father(ken,harriet) | | Father(james,colin) |
| Father(george,alice) | Father(andrew,james) | Mother(christine,james) |
| Father(george,bob) | Father(andrew,jennifer) | Mother(christine,jennifer) |
| Mother(laura,harriet) | Father(christopher,arthur) | Mother(penelope,victoria) |
| Mother(harriet,alice) | Father(christopher,victoria) | Mother(victoria,charlotte) |
| Mother(harriet,bob) | Father(james,charlotte) | Mother(victoria,colin) |

Figure 25 - Control+ Clustered Input

The resulting learnt relations are shown on Figure 26.

```

Learn: Learn1, arg1: Father , arg2: Mother
Learn: Learn2, arg1: InvFather , arg2: InvMother
Learn: Learn3, arg1: Learn1 , arg2: Learn2
Learn: Learn4, arg1: Learn1 , arg2: Learn1
Learn: Learn5, arg1: Learn2 , arg2: Learn2
Learn: Learn6, arg1: Learn3 , arg2: Learn1
Learn: Learn7, arg1: Learn3 , arg2: Learn2
Learn: Learn8, arg1: Learn4 , arg2: Learn2
Learn: Learn9, arg1: Learn5 , arg2: Learn1
Learn: Learn10, arg1: Learn7 , arg2: Learn2
Learn: Learn11, arg1: Learn7 , arg2: Learn1
Learn: Learn12, arg1: Learn8 , arg2: Learn2
Learn: Learn13, arg1: Learn8 , arg2: Learn1
Learn: Learn14, arg1: Learn6 , arg2: Learn1
Learn: Learn15, arg1: Learn6 , arg2: Learn2
Learn: Learn16, arg1: Learn9 , arg2: Learn2

```

Figure 26 - Control+ Clustered Output

At last, the fourth test case was a larger family, with twice as many relations as from the Control+ case. The input is detailed on Figure 27.

| | | | |
|----------------------------|------------------------------|----------------------|-----------------------|
| Father(ken,harriet) | Father(christopher,victoria) | Father(don,mike) | Father(randy,blair) |
| Father(george,alice) | Father(james,charlotte) | Father(don,anne) | Father(ron,katarina) |
| Father(george,bob) | Father(james,colin) | Mother(rosie,randy) | Mother(katarina,ashe) |
| Mother(laura,harriet) | Mother(christine,james) | Mother(rosie,mike) | Mother(katarina,jax) |
| Mother(harriet,alice) | Mother(christine,jennifer) | Mother(rosie,anne) | Father(garen,ashe) |
| Mother(harriet,bob) | Mother(penelope,victoria) | Father(elmer,don) | Father(garen,jax) |
| Mother(alice,penelope) | Mother(victoria,charlotte) | Mother(mildred,don) | Father(jax,lux) |
| Father(andrew,james) | Mother(victoria,colin) | Mother(esther,rosie) | Mother(leona,lux) |
| Father(andrew,jennifer) | Father(colin,julie) | Mother(esther,ron) | Mother(ashe,annie) |
| Father(christopher,arthur) | Father(don,randy) | Mother(julie,esther) | Father(darius,annie) |

Figure 27 - Control+ Extended Input

From all these facts were obtained the results shown on Figure 28.

| | |
|--|--|
| Learn: Learn1, arg1: Mother , arg2: Father | Learn: Learn13, arg1: Learn7 , arg2: Learn2 |
| Learn: Learn2, arg1: InvFather , arg2: InvMother | Learn: Learn14, arg1: Learn8 , arg2: Learn1 |
| Learn: Learn3, arg1: Learn1 , arg2: Learn2 | Learn: Learn15, arg1: Learn10 , arg2: Learn2 |
| Learn: Learn4, arg1: Learn1 , arg2: Learn1 | Learn: Learn16, arg1: Learn7 , arg2: Learn1 |
| Learn: Learn5, arg1: Learn2 , arg2: Learn2 | Learn: Learn17, arg1: Learn8 , arg2: Learn2 |
| Learn: Learn6, arg1: Learn3 , arg2: Learn1 | Learn: Learn18, arg1: Learn6 , arg2: Learn2 |
| Learn: Learn7, arg1: Learn3 , arg2: Learn2 | Learn: Learn19, arg1: Learn11 , arg2: Learn1 |
| Learn: Learn8, arg1: Learn4 , arg2: Learn2 | Learn: Learn20, arg1: Learn9 , arg2: Learn1 |
| Learn: Learn9, arg1: Learn4 , arg2: Learn1 | Learn: Learn21, arg1: Learn9 , arg2: Learn2 |
| Learn: Learn10, arg1: Learn5 , arg2: Learn1 | Learn: Learn22, arg1: Learn11 , arg2: Learn2 |
| Learn: Learn11, arg1: Learn5 , arg2: Learn2 | Learn: Learn23, arg1: Learn10 , arg2: Learn1 |
| Learn: Learn12, arg1: Learn6 , arg2: Learn1 | |

Figure 28 - Control+ Extended Output

All these cases serve a specific purpose, which is described on the next section. As for a comparison of runtime between these four cases (as well as sub timers for each module), each case was executed 10 times, with their runtimes being averaged out in seconds, resulting in Table 2.

| | Module 1 | Module 2 | Module 3 | Total |
|-------------------------|-----------|-----------|-----------|-----------|
| Control | 0.0000846 | 0.0000861 | 0.0000255 | 0.0001962 |
| Control+ | 0.0001253 | 0.0006404 | 0.0001305 | 0.0008962 |
| Control+ Cluster | 0.0001145 | 0.0003981 | 0.0000706 | 0.0005832 |
| Control+ Ext. | 0.0001998 | 0.0017487 | 0.0001618 | 0.0021103 |

Table 2 - Method 1 Runtimes in seconds

5.1.2 Analysis

In the previous section, we showed a lot of information related to the different inputs that we used. In this section, we make sense of all that was shown, as well as analysing the runtimes accordingly. We will also use a visual representation of Table 2 (Figure 29), to better understand the values.

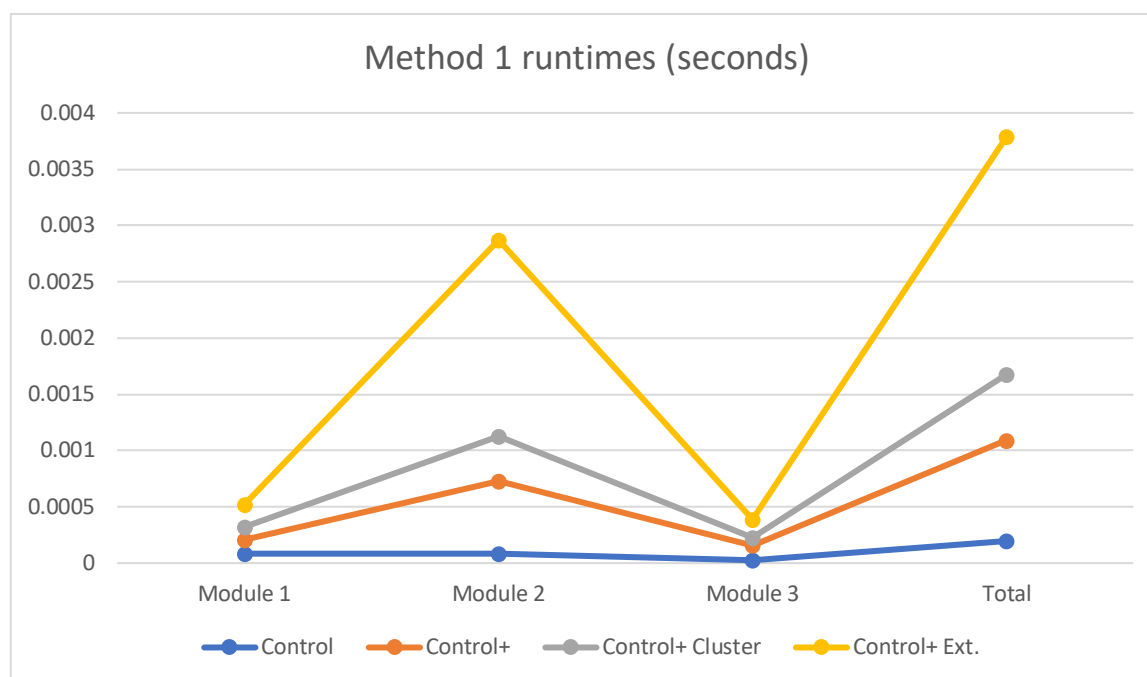


Figure 29 - Visual representation of method 1 runtimes (seconds)

The first example that we showed was meant to be used as a control. That was the family that we used throughout the development of our program, and the family that we used to explain our implementation, so it makes sense that we use it as a starting point, a reference to the other examples.

By observing its output, we see 'Learn1' and 'Learn2' that are disjunctive, as previously mentioned, acting as Parent and Child, respectively. The remaining relations are conjunctive, pertaining to all the relations that we were able to observe in that Graph List. So, for instance, 'Learn3' having 'Learn1' and 'Learn2' as arguments, can be written down mathematically as one of the two options shown on Figure 30.

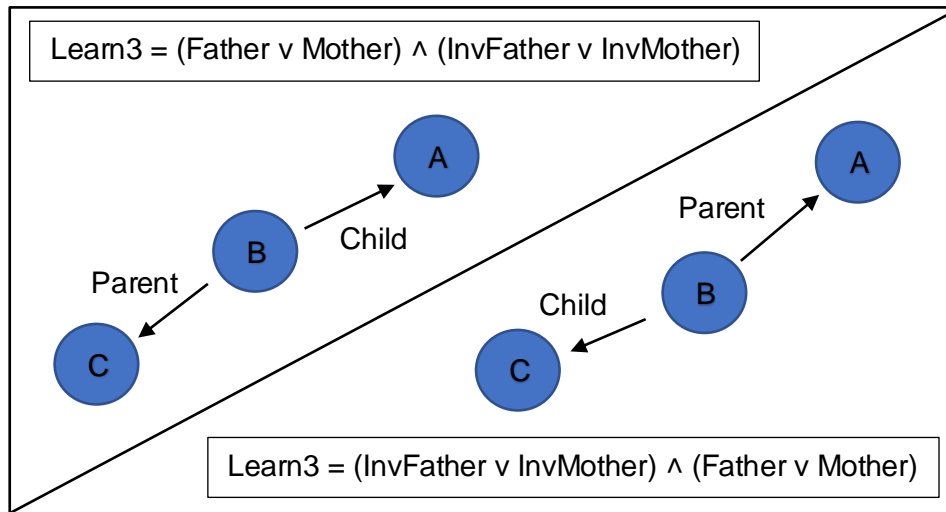


Figure 30 - Conjunction and Disjunction cases

Now, while at first glance we perceive these relations as different, they are one and the same, representing a parent (B) and two children (A,C). While the relation is the same in both cases, what changes is the order by which module 2 iterated over the nodes. For the upper left example, module 2 iterated CBA, while on the lower right example, it iterated as ABC. What is important to take from this example, is that not only these two relations show the same example, but we are able to represent it as a relation of parents. If we have a function that unfolds and prints the relations that were learnt, we could have (A Learn3 C) which would be converted into (A Learn1 Learn2 C) which in turn would notice an inverse term on Learn 2 and would understand that there is a node from which the left and right side of the expression branch out from, resulting in a (B Learn1 A ^ B Learn1 C). This would correspond as a sibling's relation.

The second example used, Control+ was an extension to the original Control test, as a way to observe how much the runtimes would increase, with three times as many facts. If we refer to Table 2, we can observe that while Module 1 wasn't affected so much, Module 2 and Module 3 run times increased close to eight and five-fold respectively. As expected, this iterable function approach scales terribly with the more facts that are added.

The third example was meant to test how well or how bad the program performed when the family wasn't a unified family. We removed a single fact, which split the Control+ family into two families. Once again, Module 1 wasn't much affected, while Module 2 and Module 3 run times were decreased by almost half. This makes sense, since that if we have smaller

clusters instead of a big family, there are quite a lot less paths that we can use. In fact, even the number of learnt rules are smaller.

The fourth and final example is an extension on Control+, doubling the number of facts that were given to the program. Once more, as we compare the different values, we observe that Module 1 has a very steady increase, while Module 2 and Module 3 abruptly increase once again, showing the poor scalability of this method.

5.2 Method 2 – Adjacency Pointers

The analysis of this second method, as well as the runtimes will be explained here, in a single section, opposed to the previous split of 2 sections for the presentation of method 1. This is simply because both the input and output were the same for all four tests, the only thing changing being the run times, as shown in Table 3 and Figure 31.

| | Module 1 | Module 2 | Module 3 | Total |
|-------------------------|-----------|-----------|-----------|-----------|
| Control | 0.0001049 | 0.0000615 | 0.0000290 | 0.0001954 |
| Control+ | 0.0001599 | 0.0002996 | 0.0000894 | 0.0005489 |
| Control+ Cluster | 0.0001532 | 0.0002257 | 0.0000612 | 0.0004401 |
| Control+ Ext. | 0.0002750 | 0.0008137 | 0.0001658 | 0.0012545 |

Table 3 - Method 2 Runtimes in seconds

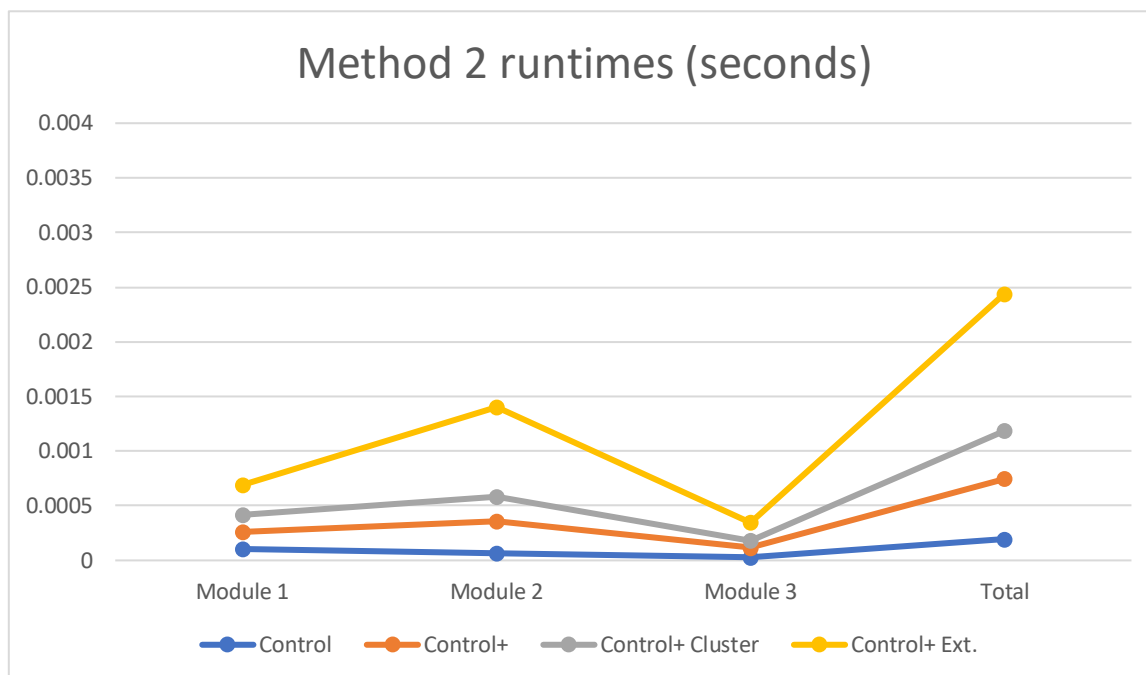


Figure 31 - Visual representation of method 2 runtimes (seconds)

Comparing the two tables and graphs, we can see some very interesting results. The first thing that we notice is that for the most part, all values from Method 2 seem be lower than those from Method 1. Now to discriminate in a more specific, and more example-oriented way:

In the Control test, we can observe that Module 1 and Module 3 run times are higher on method 2 than on method 1, opposed to Module 2, which decreased. This was expected, as we predicted that Module 1 would become 'bulkier' as it had to prepare a pointer for each adjacency (which if remember from our implementation, means two pointers for each adjacency). Module 2 would become lighter due to the ease which it would jump between nodes, instead of iterating through the full list to find the target node. Module 3's increase was minor, and from the 10 results collected, we could see that it was due to 2 anomalies on the runtimes, which lightly spiked the results (opposed to only one on the 10-set of Control test from Method 1). We believe that by running the program a higher number of times, this value will stabilize very closely to the value obtained in Method 1. However, despite all these changes, the total runtime of the test is almost the same for both methods.

On Control+ test, the differences in scalability between the two methods become more apparent. Module 1 continues to act as predicted, taking more time due to being more loaded on Method 2, but besides the first module, all values across the board decreased. The total value of $8,962 \times 10^{-4}s$ decreased to $5,489 \times 10^{-4}s$. We were able to see the difference immediately after only doubling the number of facts that were given to the program. Although as not as noticeable, we also observe the same pattern in Control+ Clustered, with Module 1 runtimes being the exception on the otherwise decrease in runtime.

On the final test, Module 1 runtime is still higher than on Method 1, however it increases steadily on this method as well. The remaining values are now vastly different from the ones obtained from the first method, and now clearly prove that Method 2 has much better scalability, as the results showed roughly a 41% decrease in execution time on the larger test that was covered.

6 Conclusion

Hierarchical Relational Learning is quite an intimidating topic to be working on. Despite the fact that the starting point and the objective of this task are distinctly described, there was no clear way about how to actually connect the dots. Of course, despite the complexity of the task, our work is nowhere near complete or extensive as the creation of Aleph, or other ILP systems. This is fine, as we are comparing projects of different magnitudes, both on the sense of the time it took for development, as well as their capabilities and end product. As previously stated, this work is only meant to be somewhat of an alternative to a small part of Aleph's capabilities, extending them for our own specific goals. Still, we hope that our contribution, however small, will be useful in the development of the ILP area.

The current literature for HRL is not as well developed as other ML areas are. Since we are trying to learn intermediate rules that were not declared previously, we need to include predicate invention in at least part of our work. However, we have seen that a lot of ILP systems have problems integrating predicate invention in their execution, with some only being partly able to do so. There are many concerns related to how and when should predicate invention be used, and how well we can evaluate such rules. The fact that it is done in conjunction with the induction of rules, some of which might be wrong or too redundant/broad to be useful, makes the result a bit uncertain, which we think is why this area is still so early in development.

To address this issue, we decided to take a step back. We discarded mechanisms that were too complex, avoided attempting to learn with uncertainty and instead created a program which would learn from knowledge that was undoubtedly true. We learn from the facts that are presented as base knowledge and don't try to induce or generalise new cases that we come across. We iterate over the network of relations that we built from the facts and archive all the relations that we can observe, then simplifying them by representing the higher-level ones with a mix of lower-level ones. While it is somewhat of a naïve approach, and sensitive to missing information (the Control+ compared to the Control+ Cluster proved that point), it learns the information that it is supposed to, potentially acting as some pre-processing for more complex ILP systems.

Regarding the structure of our work, the division of the code into modules also greatly helped for its development, as it allowed us to partition the main and more complex issues into smaller issues, which we were then able to tackle individually. A complex problem is, more often than not, just an aggregated of smaller problems. We only need to identify them and work on them properly. With that in mind, the critique of our work will also follow this similar module structure, as to address each module's issues, and then concluding by an overall conclusion of the program as a whole.

The first module is probably the simplest of the three as it takes care of receiving and structuring the BK into an iterable graph. There is little room for comments here, besides the optimization of how the knowledge gets obtained, and the two methods that we previously discussed for the iteration.

For the second module, there is quite a lot more to talk about, as this is where the base of the actual learning is. The learning will happen with whatever the Group List is able to extract from the BK. This potentially limits what we can learn on the next module so it's important to discuss about the limitations of this step.

As previously discussed, we observed that the way that the Group List obtains knowledge is through a depth like search in a collection of binary trees (each of them having a node from the Graph List as top node) and is thus limited by relations with at most two nodes involved (independently if the relations involved are transitive or not). If we observe the superSiblings example once more, it's very clear that we can't obtain the full knowledge of the whole relation at once on a Group List node due to reasons we explained on chapter 4. However, even if we were able to extend our program to allow the learning of relationships with 3 related nodes, we would face on the third module a similar issue that we faced on Figure 16 and Figure 17.

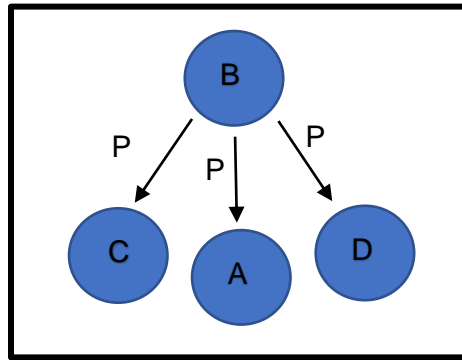


Figure 32 - superSibling representation

More concretely, if we were to define SuperSibling as is shown on Figure 32, we would need a definition similar to $\text{superSibling}(A,C,D)$, where we ignore at most one node (in this case it would be node B, which will serve as an undefined node that would link all three nodes from the rule, similar to how node B was 'ignored' in Figure 16's example) as to make the connection happen. Depending on our solution, we could ignore node B and have two 2-sized relations (CBA, ABD) like how we did previously, or ignore none and have all three relations defined 'properly' (BC, BA, BD). However, no matter which option we pick, we need to increase the number of nodes that is saved as information in our structures by at least one, or completely change the way the code works. But this is not the single issue that we have related to this situation. If we increase the number of nodes needed for a relation to 4, we once again bump into this problem.

A possible fix for this issue in future work could be done by a rework of the inner workings of the code with this issue specifically in mind. Taking Figure 32 as example, if we were to execute the search on node B, we could send a 'seeker' to each of the outgoing relations. From each of these seekers, they would observe the relations (similar to how our second module currently looks for groupings) first by backtracking back to B, and then exploring other outgoing relations from B. In our case we would have [CBA,CBD] for seeker C, [ABC,ABD] for seeker A, and [DBA, DBC] for seeker D. We would then join the results, taking into consideration to remove duplicates with the same nodes on the extremities (ABC = CBA), which would result in [CBA,CBD,ABD]. Now, we can notice that CBD is transitive, as (CBA,ABD) = CBD. However, that's more of an optimization issue, about how compact and non-redundant the rules are. This can be easily fixed by detecting transitivity in the given groupings. The conclusion that we want to take from this is that it might be possible by the solution described above, although computationally more expensive, which is to be

expected. If the increase in complexity isn't an issue however, another option could be the implementation of a hypergraph. This would imply rewriting most of the code as well as the structures themselves, but if we managed to efficiently use hypernodes, we would be able to process the `superSiblings()` problem, since that our limitation by using graphs was the edges of the graph only connecting two nodes. By contrast, hypergraphs have hyperedges which allow several nodes to be connected by the same connection, removing that barrier.

The third module is relatively simple to analyse, although not as simple as the first one. Being mostly the module that is in charge of processing the results of module 2, there is only a point or two where it can 'fail'. First, the `commonFactorAnalysis()` is slightly tailored to the examples at hand and assumes there are only two types of facts (mother/father). Should it not be the case, it could prove to be an issue. If there is only one fact the function is not needed. If there are more than two however, the function needs to be adapted to process the three, or four different results, which could prove to be a problem, as it would significantly increase the complexity of this function. On the other hand, it would also significantly compress the learnt rules, avoiding much redundancy.

The other issue is described on Figure 12, which is depending on how the Graph List is built, the Group List learns relations in a different way. As we mentioned previously, it isn't an issue in graphs that are sufficiently large, as both the case on the left and the case on the right are found. On smaller examples, what we end up learning can be more limited. This is, however, not really an issue as both ways to build the relation will end up building the exact same relation in different ways.

Overall, our work results in a simple and lightweight program. While it can't learn as much as ILP systems, its simplicity has the advantage of not dawdling with probabilities, as we are extracting relations from the BK, and not trying to create/define them based on probabilistic models or previously prepared definitions. In fact, since we partially integrate predicate invention on the final module in an automatic way (without the user needing to create the predicates beforehand) it allows a much bigger automatization of the learning process. Of course, it suffers more from other probabilistic approaches from lack of knowledge, since we cannot learn from a relationship grouping that is not present on the Group List.

While the names of the output learnt relations only differ in the number following “learn”, we can simply extract the learnt knowledge and swap the terms in a post analysis of the results as a way to make it more human-friendly.

In matters of how much we can generalise the code for BK’s other than a family representation, while there is one or another term that need to be swapped out depending on our BK and initial terms, there is virtually no setup needed, opposed to more complicated programs. Additionally, we have seen that, for now, it can only learn relations in a binary fashion, choking when dealing with relations that have 3 or more nodes to consider at the same time. This peculiarity however can be addressed in future work, hopefully expanding it to non-binary relations. A final positive note about our program is that it can learn from families that have clusters of nodes. Of course, there is a limit to how many relation groupings we can obtain from small clusters, but nonetheless, with the way that the program was implemented, it is possible to do it.

7 Bibliography

- [1] S. H. Muggleton and C. Hocquette, "Machine Discovery of Comprehensible Strategies for Simple Games Using Meta-interpretive Learning," 25 04 2019. [Online]. Available: <https://link.springer.com/article/10.1007/s00354-019-00054-2>. [Accessed 18 12 2021].
- [2] H. Blockeel and J. Struyf, "Encyclopedia of Machine Learning," 2010. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-30164-8_719. [Accessed 08 01 2021].
- [3] H. Khosravi and B. Bina, "A Survey on Statistical Relational Learning," 05 2010. [Online]. Available: <https://www.cs.ubc.ca/~hkhosrav/pub/survey.pdf>. [Accessed 06 01 2021].
- [4] L. D. Raedt, "Encyclopedia of Machine Learning," 2010. [Online]. Available: https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-30164-8_396. [Accessed 08 01 2021].
- [5] S. Muggleton, "Inverse entailment and prolog," 12 1995. [Online]. Available: <https://link.springer.com/article/10.1007/BF03037227>. [Accessed 23 09 2021].
- [6] A. Srinivasan, "Aleph Manual," 13 03 2007. [Online]. Available: www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html. [Accessed 06 01 2021].
- [7] O. Ray, "Nonmonotonic abductive inductive learning," 09 2009. [Online]. Available: www.researchgate.net/publication/220429997. [Accessed 29 09 2021].
- [8] J. Ahlgren and S. Y. Yuen, "Efficient Program Synthesis Using Constraint Satisfaction," 12 2013. [Online]. Available: jmlr.csail.mit.edu/papers/volume14/ahlgren13a/ahlgren13a.pdf. [Accessed 29 09 2021].
- [9] V. S. Costa, R. Rocha and L. Damas, "The YAP Prolog system," 11 2011. [Online]. Available: www.researchgate.net/publication/48210071_The_YAP_Prolog_system. [Accessed 29 09 2021].
- [10] J. W. "SWI-Prolog," [Online]. Available: <https://www.swi-prolog.org/>. [Accessed 29 09 2021].

- [11] A. Cropper and S. Muggleton, "Metagol System," 2016. [Online]. Available: <https://github.com/metagol/metagol>. [Accessed 23 09 2021].
- [12] A. Cropper, R. Morel and S. H. Muggleton, "Learning higher-order logic programs," 25 07 2020. [Online]. Available: <https://arxiv.org/abs/1907.10953>. [Accessed 12 01 2021].
- [13] S. Muggleton, "Meta-Interpretive Learning: Achievements and Challenges," 14 06 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-61252-2_1. [Accessed 12 01 2021].
- [14] A. Stanić, S. v. Steenkiste and J. Schmidhuber, "Hierarchical Relational Inference," 7 10 2020. [Online]. Available: <https://arxiv.org/abs/2010.03635>. [Accessed 15 01 2021].
- [15] A. C.Tenorio-González and E. F. Morales, "Automatic discovery of relational concepts by an incremental graph-based representation," 09 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0921889016303542>. [Accessed 15 01 2021].
- [16] S. Kok and P. Domingos, "Statistical Predicate Invention," 06 2007. [Online]. Available: <https://homes.cs.washington.edu/~pedrod/papers/mlc07.pdf>. [Accessed 15 01 2021].
- [17] A. Cropper and S. Dumančić, "Inductive logic programming at 30: a new introduction," 13 10 2020. [Online]. Available: <https://arxiv.org/abs/2008.07912>. [Accessed 06 01 2021].
- [18] S. Muggleton, "Inductive logic programming," 02 1991. [Online]. Available: <https://link.springer.com/article/10.1007%2FBF03037089>. [Accessed 14 01 2021].
- [19] J. Fümkrantz, D. Gamberger and N. Lavrač, "Foundations of Rule Learning," 27 9 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-75197-7_5. [Accessed 15 01 2021].
- [20] N. Lavrač and P. A. Flach, "An extended transformation approach to inductive logic programming," 04 2000. [Online]. Available: <https://dl.acm.org/doi/10.1145/383779.383781>. [Accessed 16 01 2021].
- [21] H. Ferreira, "Propositional Based Inductive Logic Programming: Reduced Encoding of Hypotheses," [Online]. Available: https://paginas.fe.up.pt/~prodei/dsie09/docs/hugoferreira/prop-relation-final_b.pdf. [Accessed 16 01 2021].

- [22] S. Muggleton, R. Otero and A. Tamaddoni-Nezhad, "Inductive Logic Programming," 24-27 08 2006. [Online]. Available: <https://link.springer.com/book/10.1007/978-3-540-73847-3>. [Accessed 16 01 2021].
- [23] D. LIN, "Efficient, Complete and Declarative Search in Inductive Logic programming," 09 2009. [Online]. Available: http://ilp.doc.ic.ac.uk/mcTopLog/lin_mscThesis.pdf. [Accessed 16 01 2021].
- [24] S. Kramer, "A Brief History of Learning Symbolic Higher-Level Representations from Data (And a Curious Look Forward)," 2020. [Online]. Available: <https://www.ijcai.org/proceedings/2020/678>. [Accessed 24 09 2021].
- [25] D. Athakravi, K. Broda and A. Russo, "Predicate Invention in Inductive Logic," January 2012. [Online]. Available: www.researchgate.net/publication/286102994. [Accessed 24 09 2021].
- [26] H. Suryanto and P. Compton, "Invented Predicates to Reduce," 03 2003. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.4511>. [Accessed 24 09 2021].
- [27] S. Muggleton et al, "Ultra-Strong Machine Learning: comprehensibility of programs learned with ILP," 07 05 2018. [Online]. Available: <https://link.springer.com/article/10.1007/s10994-018-5707-3>. [Accessed 24 09 2021].
- [28] S. Muggleton, "Predicate invention and utilization," 27 04 2007. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/09528139408953784>. [Accessed 24 09 2021].